14th International Conference on Interactive Theorem Proving

ITP 2023, July 31 to August 4, 2023, Białystok, Poland

Edited by

Adam Naumowicz

René Thiemann

Editors

Adam Naumowicz
University of Białystok, Poland
adamn@math.uwb.edu.pl

René Thiemann
University of Innsbruck, Austria
rene.thiemann@uibk.ac.at

ACM Classification 2012
Theory of computation → Interactive proof systems; Theory of computation → Higher order logic; Software and its engineering → Formal methods; Theory of computation → Program reasoning; Computing methodologies → Theorem proving algorithms

ISBN 978-3-95977-284-6

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-284-6.

Publication date
July, 2023

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.2023.0

ISBN 978-3-95977-284-6 **ISSN 1868-8969** **https://www.dagstuhl.de/lipics**

LIPIcs – Leibniz International Proceedings in Informatics

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

Editorial Board

- Luca Aceto (*Chair*, Reykjavik University, IS and Gran Sasso Science Institute, IT)
- Christel Baier (TU Dresden, DE)
- Mikolaj Bojanczyk (University of Warsaw, PL)
- Roberto Di Cosmo (Inria and Université de Paris, FR)
- Faith Ellen (University of Toronto, CA)
- Javier Esparza (TU München, DE)
- Daniel Král' (Masaryk University - Brno, CZ)
- Meena Mahajan (Institute of Mathematical Sciences, Chennai, IN)
- Anca Muscholl (University of Bordeaux, FR)
- Chih-Hao Luke Ong (University of Oxford, GB and 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)

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

Preface
Adam Naumowicz and René Thiemann 0:ix

Invited Talks

Formalisation of Additive Combinatorics in Isabelle/HOL
Angeliki Koutsoukou-Argyraki 1:1–1:2

Interactive and Automated Proofs in Modal Separation Logic
Robbert Krebbers 2:1–2:1

Regular Papers

A Formal Analysis of RANKING
Mohammad Abdulaziz and Christoph Madlener 3:1–3:18

Fast, Verified Computation for Candle
Oskar Abrahamsson and Magnus O. Myreen 4:1–4:17

Formalizing Functions as Processes
Beniamino Accattoli, Horace Blanc, and Claudio Sacerdoti Coen 5:1–5:21

An Elementary Formal Proof of the Group Law on Weierstrass Elliptic Curves in Any Characteristic
David Kurniadi Angdinata and Junyan Xu 6:1–6:19

A Proof-Producing Compiler for Blockchain Applications
Jeremy Avigad, Lior Goldberg, David Levit, Yoav Seginer, and Alon Titelman 7:1–7:19

No Unification Variable Left Behind: Fully Grounding Type Inference for the HDM System
Roger Bosman, Georgios Karachalias, and Tom Schrijvers 8:1–8:18

Automated Theorem Proving for Metamath
Mario Carneiro, Chad E. Brown, and Josef Urban 9:1–9:19

Reimplementing Mizar in Rust
Mario Carneiro 10:1–10:18

Now It Compiles!: Certified Automatic Repair of Uncompilable Protocols
Luís Cruz-Filipe and Fabrizio Montesi 11:1–11:19

Lessons for Interactive Theorem Proving Researchers from a Survey of Coq Users
Ana de Almeida Borges, Annalí Casanueva Artís, Jean-Rémy Falleri, Emilio Jesús Gallego Arias, Érik Martin-Dorel, Karl Palmskog, Alexander Serebrenik, and Théo Zimmermann 12:1–12:18

Formalizing Norm Extensions and Applications to Number Theory
María Inés de Frutos-Fernández 13:1–13:18

14th International Conference on Interactive Theorem Proving (ITP 2023).
Editors: Adam Naumowicz and René Thiemann
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Tealeaves: Structured Monads for Generic First-Order Abstract Syntax Infrastructure
Lawrence Dunn, Val Tannen, and Steve Zdancewic 14:1–14:20

Closure Properties of General Grammars – Formally Verified
Martin Dvorak and Jasmin Blanchette 15:1–15:16

Formalising Yoneda Ext in Univalent Foundations
Jarl G. Taxerås Flaten 16:1–16:17

LISA – A Modern Proof System
Simon Guilloud, Sankalp Gambhir, and Viktor Kunčak 17:1–17:19

Semantic Foundations of Higher-Order Probabilistic Programs in Isabelle/HOL
Michikazu Hirata, Yasuhiko Minamide, and Tetsuya Sato 18:1–18:18

MizAR 60 for Mizar 50
Jan Jakubův, Karel Chvalovský, Zarathustra Goertzel, Cezary Kaliszyk, Mirek Olšák, Bartosz Piotrowski, Stephan Schulz, Martin Suda, and Josef Urban . 19:1–19:22

Constructive Final Semantics of Finite Bags
Philipp Joram and Niccolò Veltri 20:1–20:19

Proof Pearl: Faithful Computation and Extraction of μ-Recursive Algorithms in Coq
Dominique Larchey-Wendling and Jean-François Monin 21:1–21:17

Group Cohomology in the Lean Community Library
Amelia Livingston 22:1–22:17

A Formalisation of Gallagher's Ergodic Theorem
Oliver Nash 23:1–23:16

An Extensible User Interface for Lean 4
Wojciech Nawrocki, Edward W. Ayers, and Gabriel Ebner 24:1–24:20

Bel-Games: A Formal Theory of Games of Incomplete Information Based on Belief Functions in the Coq Proof Assistant
Pierre Pomeret-Coquot, Hélène Fargier, and Érik Martin-Dorel 25:1–25:19

Proof Repair Infrastructure for Supervised Models: Building a Large Proof Repair Dataset
Tom Reichel, R. Wesley Henderson, Andrew Touchet, Andrew Gardner, and Talia Ringer 26:1–26:20

POSIX Lexing with Bitcoded Derivatives
Chengsong Tan and Christian Urban 27:1–27:18

A Sound and Complete Projection for Global Types
Dawit Tirore, Jesper Bengtson, and Marco Carbone 28:1–28:19

Real-Time Double-Ended Queue Verified (Proof Pearl)
Balazs Toth and Tobias Nipkow 29:1–29:18

Certifying Higher-Order Polynomial Interpretations
Niels van der Weide, Deivid Vale, and Cynthia Kop 30:1–30:20

Slice Nondeterminism
Niels F. W. Voorneveld 31:1–31:19

Foundational Verification of Stateful P4 Packet Processing
Qinshi Wang, Mengying Pan, Shengyi Wang, Ryan Doenges, Lennart Beringer, and Andrew W. Appel 32:1–32:20

Dependently Sorted Theorem Proving for Mathematical Foundations
Yiming Xu and Michael Norrish 33:1–33:18

Formalizing Results on Directed Sets in Isabelle/HOL (Proof Pearl)
Akihisa Yamada and Jérémy Dubut 34:1–34:13

Formalising the *Proj* Construction in Lean
Jujian Zhang 35:1–35:17

Short Papers

Fermat's Last Theorem for Regular Primes
Alex J. Best, Christopher Birkbeck, Riccardo Brasca, and Eric Rodriguez Boidi ... 36:1–36:8

Implementing More Explicit Definitional Expansions in Mizar
Adam Grabowski and Artur Korniłowicz 37:1–37:8

Formalizing Almost Development Closed Critical Pairs
Christina Kohl and Aart Middeldorp 38:1–38:8

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 back 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 Białystok in Poland. It is hosted by the Mizar group of the University of Białystok. 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 and Haifa 2022; 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 78 submissions (70 regular papers and 8 short papers). Each paper was systematically reviewed by at least three program committee members or appointed external reviewers, as a result of which the PC winnowed down the selection to be presented at the conference: 36 papers (33 regular papers and 3 short papers). We thank the authors of both accepted and rejected papers for their submissions, as well as the PC members and external reviewers for their invaluable work.

As well as all the regular papers, we are very pleased to have invited talks by Angeliki Koutsoukou-Argyraki (University of Cambridge) and Robbert Krebbers (Radboud University Nijmegen). The present volume collects all the accepted papers contributed to the conference as well as abstracts of the two invited talks. This is the fourth time that the ITP proceedings are published in the LIPIcs series. We thank all those 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 throughout.

Adam Naumowicz and René Thiemann

14th International Conference on Interactive Theorem Proving (ITP 2023).
Editors: Adam Naumowicz and René Thiemann
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Organisation

Programme Chairs

Adam Naumowicz	University of Białystok, Poland
René Thiemann	University of Innsbruck, Austria

Programme Committee

Andreas Abel	Jesús Aransay	Jeremy Avigad
Mauricio Ayala-Rincon	Christoph Benzmüller	Jasmin Blanchette
Sandrine Blazy	Sylvie Boldo	Cyril Cohen
Liron Cohen	Luis Cruz-Filipe	Ruben Gamboa
Jason Gross	John Harrison	Hugo Herbelin
Cezary Kaliszyk	Chantal Keller	Peter Lammich
Andreas Lochbihler	Marco Maggesi	Assia Mahboubi
Magnus O. Myreen	Cláudia Nalon	Tobias Nipkow
Michael Norrish	John O'Leary	Lawrence Paulson
Andrei Popescu	Bas Spitters	Josef Urban
Makarius Wenzel	Freek Wiedijk	Akihisa Yamada

Local Organisation

Czesław Byliński
Adam Grabowski
Artur Korniłowicz
Roman Matuszewski
Karol Pąk

External Reviewers

Rajashree Agrawal	Johannes Åman Pohjola	Thaynara Arielly de Lima
Alexander Best	Timothy Bourke	Mario Carneiro
Felix Cherubini	Vikraman Choudhury	Stefan Ciobaca
Evelyne Contejean	Sander Dahmen	Jérémy Dubut
Manuel Eberl	Andres Erbsen	Daniil Frumin
David Fuenmayor	Lorenzo Gheri	Daniel Gratzer
Ariel Grunfeld	Roberto Guanciale	Stepan Holub
Matthias Hutzler	Jules Jacobs	Jacques-Henri Jourdan
Ohad Kammar	Dominik Kirst	Bram Kohlen
Amélie Ledein	Milan Lopuhaä-Zwakenberg	Aart Middeldorp
Houda Mouhcine	Julian Parsert	Bartosz Piotrowski
Nicolas Pouillard	Ivan Prokić	Pierre-Marie Pédrot
Robert Rubbens	Joshua Schneider	Carlos Simpson
Matthieu Sozeau	Christoph Sprenger	Runzhou Tao
Dmitriy Traytel	Daniel Ventura	Yuting Wang

Formalisation of Additive Combinatorics in Isabelle/HOL

Angeliki Koutsoukou-Argyraki
University of Cambridge, UK

Abstract

In this talk, I will present an overview of recent formalisations, in the interactive theorem prover Isabelle/HOL, of significant theorems in additive combinatorics, an area of combinatorial number theory. The formalisations of these theorems were the first in any proof assistant to my knowledge. For each of these theorems, I will discuss selected aspects of the formalisation process, focussing on observations on our treatment of certain mathematical arguments when translated into Isabelle/HOL and our overall formalisation experience with Isabelle/HOL for this area of mathematics.

2012 ACM Subject Classification Mathematics of computing → Combinatorics; Theory of computation → Logic and verification

Keywords and phrases Additive combinatorics, additive number theory, combinatorial number theory, formalisation of mathematics, interactive theorem proving, proof assistants, Isabelle/HOL

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.1

Category Invited Talk

Funding Angeliki Koutsoukou-Argyraki is funded by the ERC Advanced Grant ALEXANDRIA (Project GA 742178, European Research Council) led by Lawrence C. Paulson (University of Cambridge, Department of Computer Science and Technology). The Cambridge Mathematics Placements (CMP) Programme has been supporting and (partially) funding summer internships contributing to (ongoing) formalisations: Mantas Bakšys (2022); three more internships to contribute to Isabelle/HOL formalisations of material in a related area to be supported in 2023.

1 Summary

Additive combinatorics studies the properties of sumsets of subsets of groups, often employing proof techniques from other mathematical areas. In 2022 I initiated a line of formalisations of results in this area of mathematics using Isabelle/HOL [11], one of my main goals being the formalisation of advanced course material from the Cambridge Mathematical Tripos. My collaborators and I achieved the formalisation of a number of profound theorems in this area. A first project involved the formalisation of a proof of the Plünnecke–Ruzsa Inequality [9], an inequality giving information on the size (cardinality) of sumsets (and difference sets) of finite subsets of an abelian group. To this end, Lawrence Paulson and I, building on an algebra library by Clemens Ballarin [2], introduced the basics of sumset theory in Isabelle/HOL including basic results such as the Ruzsa Triangle Inequality [9]. Our source was the set of the 2022 lecture notes by Timothy Gowers for Part III of the Cambridge Mathematical Tripos [5]. Building on our formalisation of the basics [9] and again following [5], Lawrence Paulson and I went on to formalise Khovanskii's Theorem [8], which attests that for all sufficiently large n, the cardinality of the n-iterated sumset of a finite subset of an abelian group is polynomial in n. Continuing to follow [5], Mantas Bakšys, Chelsea Edmonds and I, formalised the Balog–Szemerédi–Gowers Theorem [7, 6], a profound result which played a central role in Gowers's proof deriving the first effective bounds for Szemerédi's Theorem. The Balog–Szemerédi–Gowers Theorem attests that every finite subset (of given additive energy) of an abelian group must contain a large subset whose sumset (difference set) is small,

© Angeliki Koutsoukou-Argyraki;
licensed under Creative Commons License CC-BY 4.0

and gives bounds on these cardinalities depending on the given additive energy. The proof is of great mathematical interest in itself given that it involves an interplay between graph theory, probability theory and additive combinatorics. This interplay made the formalisation process more rich and technically challenging, and was handled by an appropriate use of locales, Isabelle's module system. To treat the graph-theoretic aspects of the proof, we made use of a new, more general undirected graph theory library by Chelsea Edmonds [4]. Another subsequent formalisation project, this time involving proofs of purely combinatorial and algebraic flavour, was the formalisation of Kneser's Theorem (following a paper by Matt DeVos [3]) and the Cauchy–Davenport Theorem as its corollary by Mantas Bakšys and myself [1]. Both theorems give information on various estimates on the cardinality of sumsets of finite subsets of abelian groups under certain conditions. Lastly, I will very briefly comment on a new line of ongoing formalisation work that I initiated, currently in progress by my students from the Computer Science Department and my interns from the Mathematics Department at Cambridge: formalising material in additive number theory, a related research area involving combinatorial tools. In particular, this line of work involves material related to Waring's problem and follows Nathanson's book [10].

References

1 Mantas Bakšys and Angeliki Koutsoukou-Argyraki. Kneser's Theorem and the Cauchy–Davenport Theorem. *Archive of Formal Proofs*, November 2022. Formal proof development. URL: `https://isa-afp.org/entries/Kneser_Cauchy_Davenport.html`.

2 Clemens Ballarin. A Case Study in Basic Algebra. *Archive of Formal Proofs*, August 2019. Formal proof development. URL: `https://isa-afp.org/entries/Jacobson_Basic_Algebra.html`.

3 Matt DeVos. A Short Proof of Kneser's Addition Theorem for Abelian Groups. In *Springer Proceedings in Mathematics and Statistics, vol 101*, pages 39–41, New York, NY, USA, 2014. Springer New York. `doi:10.1007/978-1-4939-1601-6_3`.

4 Chelsea Edmonds. Undirected Graph Theory. *Archive of Formal Proofs*, September 2022. Formal proof development. URL: `https://isa-afp.org/entries/Undirected_Graph_Theory.html`.

5 Timothy Gowers. *Introduction to Additive Combinatorics*. Online course notes for Part III of the Mathematical Tripos, University of Cambridge, 2022.

6 Angeliki Koutsoukou-Argyraki, Mantas Bakšys, and Chelsea Edmonds. The Balog–Szemerédi–Gowers Theorem. *Archive of Formal Proofs*, November 2022. Formal proof development. URL: `https://isa-afp.org/entries/Balog_Szemeredi_Gowers.html`.

7 Angeliki Koutsoukou-Argyraki, Mantas Bakšys, and Chelsea Edmonds. A Formalisation of the Balog–Szemerédi–Gowers Theorem in Isabelle/HOL. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, Boston, MA, USA*, pages 225–238, New York, NY, USA, 2023. Association for Computing Machinery. `doi:10.1145/3573105.3575680`.

8 Angeliki Koutsoukou-Argyraki and Lawrence C. Paulson. Khovanskii's Theorem. *Archive of Formal Proofs*, September 2022. Formal proof development. URL: `https://isa-afp.org/entries/Khovanskii_Theorem.html`.

9 Angeliki Koutsoukou-Argyraki and Lawrence C. Paulson. The Plünnecke–Ruzsa Inequality. *Archive of Formal Proofs*, May 2022. Formal proof development. URL: `https://isa-afp.org/entries/Pluennecke_Ruzsa_Inequality.html`.

10 Melvyn B. Nathanson. *Additive Number Theory: The Classical Bases*. Springer-Verlag New York, 1996.

11 Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL, A Proof Assistant for Higher-Order Logic*. Springer-Verlag Berlin Heidelberg, 2002. Updated online tutorial on `https://isabelle.in.tum.de/dist/Isabelle/doc/tutorial.pdf`.

Interactive and Automated Proofs in Modal Separation Logic

Robbert Krebbers
Radboud University, Nijmegen, The Netherlands

Abstract

In program verification, it is common to embed a high-level object logic into the meta logic of a proof assistant to hide low-level aspects of the verification. To verify imperative and concurrent programs, separation logic hides explicit reasoning about heaps and pointer disjointness. To verify programs with cyclic features such as modules or higher-order state, modal logic provides modalities to hide explicit reasoning about step-indices that are used to stratify recursion.

The meta logic of proof assistants such as Coq is well suited to embed high-level object logics and prove their soundness. However, proof assistants such as Coq do not have native infrastructure to facilitate proofs in embedded logics – their proof contexts and built-in tactics for interactive and automated proofs are tailored to the connectives of the meta logic, and do not extend to those of the object logic. This results in proofs that are at a too low level of abstraction because they are cluttered with bookkeeping code related to manipulating the object logic.

In this talk I will describe our work in the Iris project to address this problem – first for interactive proofs, and then for semi-automated proofs. The *Iris Proof Mode* provides high-level tactics for interactive proofs in higher-order concurrent separation logic with modalities. Recent work on *RefinedC* and *Diaframe* have built on top of the Iris Proof Mode to obtain proof automation for low-level C programs and fine-grained concurrent programs.

2012 ACM Subject Classification Theory of computation → Separation logic; Theory of computation → Automated reasoning; Theory of computation → Program verification

Keywords and phrases Program Verification, Separation Logic, Step-Indexing, Modal Logic, Interactive Theorem Proving, Proof Automation, Iris, Coq

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.2

Category Invited Talk

Acknowledgements I thank my coauthors of the Iris Proof Mode (POPL'17, ICFP'18), RefinedC (PLDI'21), and Diaframe (PLDI'22, PLDI'23, OOPSLA'23) papers: Lars Birkedal, Arthur Charguéraud, Łukasz Czajka, Derek Dreyer, Deepak Garg, Herman Geuvers, Jacques-Henri Jourdan, Ralf Jung, Jan-Oliver Kaiser, Rodolphe Lepigre, Kayvan Memarian, Ike Mulder, Michael Sammler, Joseph Tassarotti, and Amin Timany. I thank all contributors to the Iris project.

© Robbert Krebbers;
licensed under Creative Commons License CC-BY 4.0

A Formal Analysis of RANKING

Mohammad Abdulaziz
King's College London, UK
Technische Universität München, Germany

Christoph Madlener
Technische Universität München, Germany

Abstract

We describe a formal correctness proof of RANKING, an online algorithm for online bipartite matching. An outcome of our formalisation is that it shows that there is a gap in all combinatorial proofs of the algorithm. Filling that gap constituted the majority of the effort which went into this work. This is despite the algorithm being one of the most studied algorithms and a central result in theoretical computer science. This gap is an example of difficulties in formalising graphical arguments which are ubiquitous in the theory of computing.

2012 ACM Subject Classification Theory of computation; Mathematics of computing

Keywords and phrases Matching Theory, Formalized Mathematics, Online Algorithms

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.3

Related Version *Full Version*: `https://arxiv.org/abs/2302.13747`

Funding *Mohammad Abdulaziz*: Part of this work was funded by DFG Koselleck Grant NI 491/16-1.

Acknowledgements We thank the anonymous reviewers whose comments helped improve the text and the formal proofs. We also thank Kurt Mehlhorn for his insightful comments.

1 Introduction

Matching is a classical problem in computer science, operations research, graph theory, and combinatorial optimisation. In short, in this problem, given an undirected graph, one tries to compute a subset of the edges of this graph, s.t. no two edges are incident on the same vertex. This subset is usually optimised w.r.t. a given objective, e.g. matching cardinality, sum of weights of edges in the matching, etc. An important special case of matching problems is maximum cardinality matching in bipartite graphs. It is one of the first problems to be addressed in combinatorial optimistation, where, for instance, the Hungarian method was invented in 1955 to solve it in the edge-weighted setting [14]. The online version of that problem, i.e. the version in which one of the parties of the graphs arrive online, one vertex at a time, along with its incident edges, has received special attention. This is because the problem can model many economic situations, most-notably Google's Adwords market [15].

The most basic version of online bipartite matching is the one where vertices and edges have no weights. That problem was studied by Karp, Vazirani, and Vazirani (henceforth, KVV) [13], where they devised the so-called RANKING algorithm. In that paper, KVV showed that their algorithm can solve the online problem with a *competitive ratio*, i.e. the average case ratio of the online algorithm's solution quality compared to the best offline algorithm, of $1-1/e$. They also showed that this ratio is the best possible for any randomised online bipartite matching algorithm. The analysis of the RANKING algorithm has been continuously studied, where authors have mainly tried to simplify the algorithm's original correctness proof, i.e. the proof that it achieves a $1-1/e$ competitive ratio [9, 3, 4, 6, 19, 16].


© Mohammad Abdulaziz and Christoph Madlener;
licensed under Creative Commons License CC-BY 4.0

This is because the algorithm's analysis, which can be divided into a probabilistic and a combinatorial part, is considered to be "extremely difficult" [18] by the algorithms community, despite the algorithm itself being very simple.

In this paper we formalise an analysis of the algorithm by Birnbaum and Mathieu [3] (henceforth, BM) in Isabelle/HOL [17]. BM claim to present the first simple proof of the algorithm's competitive ratio. Indeed, the paper's title is "Online bipartite matching made simple", and it is the last attempt at a simple combinatorial proof for the algorithm, as later works focused on primal-dual analyses of the algorithm.

Our most striking finding is that there is a "gap" in the proof, where there was one lemma whose proof was "a simple structural observation" by the authors. Formalising the proof of this lemma constitutes the majority of the effort that went into the work we describe here as well as the majority of the volume of the formal proof scripts. There are also other interesting aspects, from a formalisation perspective, of that proof. For instance, it combines graph theoretic, probabilistic, and graphical arguments. It also requires modelling and reasoning about online algorithms.

The rest of the paper is structured as follows. We first describe the algorithm and how we model it in Isabelle/HOL. Then we discuss the probabilistic part of the proof and its formalisation. We then discuss the combinatorial part of the proof, where we describe the main findings of this work, namely,

1. the first complete proof that covers the gap in the proof by BM, as well as other combinatorial proofs of the algorithm, and
2. a significantly simpler proof of a lemma needed by BM to facilitate the algorithm's probabilistic analysis.

Lastly, we discuss a part of the proof usually glossed over by other authors, which is lifting the analysis to obtain an asymptotic statement on the competitive ratio.

Isabelle/HOL. Isabelle/HOL [17] is a theorem prover based on Higher-Order Logic. Roughly speaking, Higher-Order Logic can be seen as a combination of functional programming with logic. Isabelle's syntax is a variation of Standard ML combined with (almost) standard mathematical notation. Function application is written prefix, and functions are usually curried (i.e., function f applied to arguments $x_1 \ \ldots \ x_n$ is written as $f \ x_1 \ \ldots \ x_n$ instead of the standard notation $f(x_1, \ \ldots \ , x_n)$). In Isabelle/HOL, *SOME* is the Hilbert choice, and *THE* is the definite description operator.

Availability. Our formalisation is in the Archive of Formal Proofs (www.isa-afp.org). Throughout the paper, and in the appendix, we added excerpts from the formalisation representing important definitions and theorem statements to aid in linking the informal description in the paper and the formal proof scripts.

2 Basic Definitions and Notation

We denote a list of elements as $[x_1, x_2, \ldots, x_n]$. In the rest of this paper, we only consider lists with distinct elements. We say element x_i has rank i[1] in the list $[x_1, x_2, \ldots, x_i, \ldots, x_n]$. We overload the membership, subset, union and intersection set operations to lists. For a list vs, of length n, and an element $v \in vs$, let, for $1 \leq i \leq n$, $vs[v \mapsto i]$ denote the list which results from inserting v into vs where v has been removed s.t. its rank is exactly i. Also, let

[1] In the formalisation we use index, which is the same as the rank minus one.

$vs(v)$ denote the rank of v in vs and $vs[i]$ the element of rank i in vs. For a list vs, $v\#vs$ denotes the list vs but with the vertex v appended to its head. A permutation of a finite set s is a list whose elements are exactly the elements of s.

An edge is a set of vertices with size 2. A graph $\mathcal{G}$ is a set of edges. The set of vertices of a graph $\mathcal{G}$, denoted by $\mathcal{V}(\mathcal{G})$, is $\bigcup_{e\in\mathcal{G}} e$. For a vertex v, $N_{\mathcal{G}}(v)$ denotes $\{u \mid \{v,u\} \in \mathcal{G}\}$. We say a graph $\mathcal{G}$ is bipartite w.r.t. to two sets of vertices V and U (henceforth, the left and right party) iff

1. $\mathcal{V}(\mathcal{G}) \subseteq V \cup U$,
2. for any $\{v,u\} \in \mathcal{G}$, we have that $\{v,u\} \not\subseteq V$ and $\{v,u\} \not\subseteq U$.

A set of edges $\mathcal{M}$ is a matching iff $\forall e \neq e' \in \mathcal{M}.\ e \cap e' = \emptyset$. For a matching $\mathcal{M}$ and a vertex v, if there is u s.t. $\{v,u\} \in \mathcal{M}$, we say u is the partner of v, denoted by $\mathcal{M}(v)$. We use $\mathcal{G} - E$ to denote the edges in $\mathcal{G}$ that are not in E, and, for a set of vertices V, $\mathcal{G} \setminus V$ denotes $\mathcal{G} \cap \{e \mid e \cap V = \emptyset\}$, i.e. the graph with edges incident to vertices in V removed.

In many cases, a matching is a subset of a graph, in which case we call it a matching w.r.t. the graph. For a graph $\mathcal{G}$, a matching $\mathcal{M}$ w.r.t $\mathcal{G}$ is a maximum cardinality matching, aka maximum matching, w.r.t. $\mathcal{G}$ iff for any matching $\mathcal{M}'$ w.r.t. $\mathcal{G}$, we have that $|\mathcal{M}'| \leq |\mathcal{M}|$. A matching $\mathcal{M}$ w.r.t. $\mathcal{G}$ is a perfect matching w.r.t. $\mathcal{G}$ iff $\mathcal{V}(\mathcal{M}) = \mathcal{V}(\mathcal{G})$. A matching $\mathcal{M}$ w.r.t. $\mathcal{G}$ is a maximal matching w.r.t. $\mathcal{G}$ iff $\forall e \in \mathcal{G}.\ e \cap \mathcal{V}(\mathcal{M}) \neq \emptyset$.

A discrete probability space P is defined by a countable sample space Ω_P and a probability mass function (PMF) $\mathbb{P}_P : \Omega_P \to [0,1]$ assigning a probability to each sample, where $\sum_{\omega\in\Omega_P} \mathbb{P}_P(\omega) = 1$. The PMF is lifted naturally to events (sets of samples) as $\mathbb{P}_P(E) = \sum_{\omega\in E} \mathbb{P}_P(E)$ for $E \subseteq \Omega_P$. The expectation of a random variable $X : \Omega_P \to \mathbb{R}$ is denoted $\mathbb{E}_{\omega\sim P}[X(\omega)]$. For a set B and a non-empty, finite subset $A \subseteq B$, $\mathcal{U}_B(A)$ is the discrete uniform distribution, i.e. $\Omega_{\mathcal{U}_B(A)} = B$ and $\mathbb{P}_{\mathcal{U}_B(A)}(a) = \frac{1}{|A|}$ if $a \in A$ and $\mathbb{P}_{\mathcal{U}(A)}(b) = 0$ if $b \notin A$. If $A = B$ we simply write $\mathcal{U}(A)$ for $\mathcal{U}_A(A)$.

We model randomised algorithms as probability distributions over the results of the algorithm. The Giry Monad [8] allows to compose random experiments in an elegant manner and is used to express randomised algorithms. The return operator gives a distribution which places probability 1 on a single sample ω, i.e. $\mathbb{P}_{\mathsf{return}(\omega)}(x)$ is 1, if $x = \omega$, and 0, otherwise.

Composition of experiments is achieved via the bind operator (written infix as $\gg\!=$). Intuitively, $P \gg\!= Q$ randomly chooses a sample ω according to P and then returns a value chosen randomly according to the distribution $Q(\omega)$. For additional legibility, we use Haskell-like **do**-notation for bind and return. This notation can be desugared recursively as follows:

$$\textbf{do}\{\ x \leftarrow P;\ \mathit{stmts}\ \} \equiv P \gg\!= (\lambda x.\ \mathit{stmts}).$$

In Isabelle/HOL, we base our work on a simple formalisation of undirected graphs by Abdulaziz et al. [1], which was introduced in the context of the verification of Edmonds' blossom matching algorithm. We use this formalisation because of its simplicity, and the fact that it has a rich library on matchings and other related notions, as we will discuss later. However, we will not further discuss the merits of this representation as it is outside of the scope of this work. Interested readers should consult the original paper [1].

Probability theory in Isabelle/HOL is based on a general formalisation of measure theory by Hölzl [10]. In the formalisation, $\mathcal{U}(A)$ is denoted *pmf_of_set* A, and return is denoted *return_pmf*. The meanings of other Isabelle/HOL notations used in the rest of the paper should be self-explanatory.

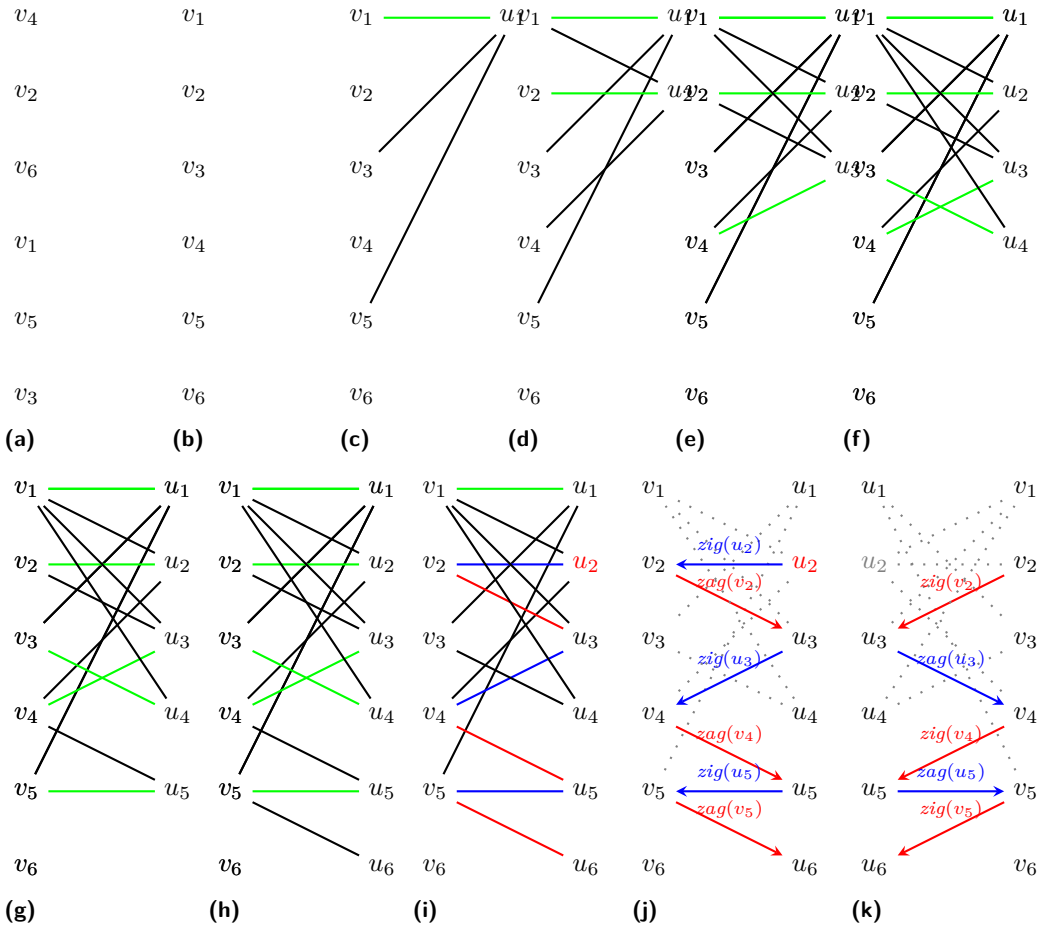


Figure 1 The steps of computing a matching using *online-match*, and what happens when an online vertex is removed from the input.

3 RANKING

Given a bipartite graph $\mathcal{G}$, whose left and right parties are V and U, the ranking algorithm takes as an offline input V, and a sequence π as an online input, where vertices, along with their adjacent edges, arrive one-by-one. As an example, consider Fig. 1a, showing a graph whose left party, i.e. the offline vertices, is $\{v_4, v_2, v_6, v_1, v_5, v_3\}$. The right party, i.e. the online vertices, arrive in the order $[u_1, u_2, u_3, u_4, u_5]$. The first step in the algorithm is that it randomly permutes the offline input. In our example, this is shown in Fig. 1b. Then, vertices from the right party of the graph arrive one-by-one. The most important thing to note is that, for every arriving vertex u, the algorithm adds the edge connecting u and the offline unmatched vertex with the minimum rank, if any such edge exists. In our example, we have the ranking $[v_1, v_2, v_3, v_4, v_5, v_6]$, of the offline vertices. Fig. 1c shows the state of the matching after the arrival of u_1: it has three edges connecting it to the offline vertices v_1, v_3, and v_5. The edge connecting it to v_1 is added to the matching, as it is unmatched and has the lowest rank among them. Then, the other vertices on the online side arrive based on the order given earlier, and the matching is updated, as shown in Fig. 1d-1g, and the final matching computed by the algorithm is the one represented by the green edges in Fig. 1h.

Algorithm 1 Pseudo-code of *RANKING*.

function *online-match(G, π, σ)* **begin**
  $\mathcal{M} \leftarrow \emptyset$
  for *every arriving vertex u in π* **do**
    if $\exists v \in (N_{\mathcal{G}}(u) - \mathcal{V}(\mathcal{M}))$ **then** $\mathcal{M} \leftarrow \mathcal{M} \cup \{\{\mathsf{argmin}_{v \in (N_{\mathcal{G}}(u) - \mathcal{V}(\mathcal{M}))} \sigma(v), u\}\}$
  return $\mathcal{M}$
end
function *RANKING(G, π)* **begin**
  $\sigma \leftarrow$ a random permutation of V
  return *online-match*$(\mathcal{G}, \pi, \sigma)$
end

Listing 1: Modelling *RANKING* in Isabelle/HOL.

```
fun step :: "'a graph ⇒ 'a ⇒ 'a list ⇒ 'a graph ⇒ 'a graph" where
  "step _ _ [] M = M"
| "step G u (v#vs) M = (
      if v ∉ Vs M ∧ u ∉ Vs M ∧ {u,v} ∈ G
      then insert {u,v} M
      else step G u vs M
    )"

fun online_match' :: "'a graph ⇒ 'a list ⇒ 'a list ⇒ 'a graph ⇒ 'a graph" where
  "online_match' _ [] _ M = M"
| "online_match' G (u#us) σ M = online_match' G us σ (step G u σ M)"

abbreviation "online_match G π σ ≡ online_match' G π σ {}"

definition "ranking ≡
  do {
    σ ← pmf_of_set (permutations_of_set V);
    return_pmf (online_match G π σ)
  }"
```

As should be clear by now, the algorithm's description and, accordingly, modeling is a simple task. The pseudo-code is in Algorithm 1. In Isabelle/HOL, we model the algorithm as shown in Listing 1. The first two functions are recursive on lists. The first function, *step*, is recursive on the list of the offline vertices, where, given a graph G, a vertex u from the online side, the list of offline vertices, and the matching, it adds to the matching the first edge it finds that connects u and an offline vertex v. The function does the recursion on the list, assuming the list is ordered according to the ranking of the offline vertices, with the head of the list being the vertex with the lowest rank. The second function, *online_match'*, is recursive on the list of online vertices, where the list is ordered according to the arrival order of those vertices, where the head of the list is the earliest arriving vertex. For each vertex in the list, *online_match'* tries to match it to an offline vertex using *step*. The other main function, *ranking*, chooses a permutation of the offline vertices and passes it to *online-match*.

We note that we avoid devising an involved way to model and reason about online computation, and only model it simply as a list of inputs and a step function that operates on each online input. This is because the algorithm description itself is simple. The primary focus of our work here is the formalisation of the correctness argument, the mathematical part of which is the main challenge.

3.1 Competitive Ratio of *RANKING*

The goal of this work is to formalise the analysis of *RANKING*'s competitiveness. In general, for online algorithms solving optimisation problems, the analysis focuses on the quality of their outputs in comparison with the quality of the output of the best offline algorithm, i.e. an algorithm which has access to the entire input before it starts computing its output. The outcome of such an analysis is referred to as the *competitive ratio* of the respective online algorithm. In the case of bipartite matchings, the best offline algorithms, like the Hopcroft-Karp algorithm [11], can compute maximum cardinality matching for bipartite graphs. Thus, for *RANKING*, the natural way to analyse it is by showing that the size of the matching it computes maintains a certain ratio if compared to the size of the maximum matching of the input graph. Furthermore, since *RANKING* is a randomised algorithm, it is natural that this relationship is in expectation. More precisely, for *RANKING*, we have the following relation, which was first shown by KVV: for any given graph and arrival orders, the ratio between the expected size of the matching computed by *RANKING* and the size of the maximum matching is $1 - 1/e$. The expectation ranges over the different permutations of the offline side.

4 Competitiveness for Bipartite Graphs with Perfect Matchings

In the following, let $\mathcal{G}$ be a bipartite graph w.r.t. V and U, s.t. $\mathcal{M}$ is a perfect matching w.r.t. $\mathcal{G}$, and $|\mathcal{M}| = n$. Let π be an arrival order for U and let $\mathcal{S}(A)$ denote the set of all permutations of a finite set A.[2]

The algorithm can be modelled as the following Giry monad

$$\mathit{RANKING}(\mathcal{G}, \pi) \equiv \textbf{do } \{\ \sigma \leftarrow \mathcal{U}(\mathcal{S}(V));\ \mathsf{return}\ \mathit{online\text{-}match}(\mathcal{G}, \pi, \sigma)\ \}.$$

In the following, we describe our formal proof of the analysis of the competitive ratio for instances with perfect matching. This formal proof closely follows the one by BM. However, we highlight the differences to the original one as they arise.

We need the following lemma ([3, Lemma 5]) before the main result can be shown.

▶ **Lemma 1.** *Let x_t denote the probability over the random permutations of V that the vertex of rank t is matched by the algorithm, for $1 \leq t \leq n$. Then $1 - x_t \leq (1/n) \sum_{1 \leq s \leq t} x_s$.*

Let $v \in V$ be the vertex of rank t for some fixed permutation σ of V. The intuition behind this bound is that v only remains unmatched if its partner $\mathcal{M}(v)$ in the perfect matching is matched to a vertex ranked lower in π. Since v is a random vertex (when drawing a permutation), so is $\mathcal{M}(v)$. The right-hand-side is supposed to be the probability that $\mathcal{M}(v)$ is matched to a vertex arriving before v (since the sum is the expected number of vertices matched to vertices of rank at most t). This intuitive idea does not work due to the dependence of $\mathcal{M}(v)$ and the set of vertices matched to vertices of rank at most t. The correct argument avoids this dependence. However, this requires a stronger statement on what happens with $\mathcal{M}(v)$ if v stays unmatched, captured in the following lemma ([3, Lemma 4]), whose proof we discuss in the next section.

▶ **Lemma 2.** *Let $v \in V$, u denote $\mathcal{M}(v)$, and $\sigma \in \mathcal{S}(V)$. If v is not matched by online-match$(\mathcal{G}, \sigma, \pi)$ to u, then, for all $1 \leq i \leq n$, u is matched by online-match$(\mathcal{G}, \sigma[v \mapsto i], \pi)$ to a $v_i \in V$ s.t. $\sigma[v \mapsto i](v_i) \leq \sigma(v)$.*

[2] In the formalisation $\mathcal{S}(A)$ is written *permutations_of_set A*.

$\mathbb{I}'_t \equiv$ **do** {
$\sigma \leftarrow \mathcal{U}(\mathcal{S}(V))$;
$v \leftarrow \mathcal{U}(V)$;
let $R = \mathit{online\text{-}match}(\mathcal{G}, \pi, \sigma[v \mapsto t])$;
return $(v \in \mathcal{V}(R))$
}

(a) In addition to a random permutation $\sigma \in \mathcal{S}(V)$, a random vertex $v \in V$ is drawn and moved to rank t.

$\mathbb{I}''_t \equiv$ **do** {
$\sigma \leftarrow \mathcal{U}(\mathcal{S}(V))$;
$v \leftarrow \mathcal{U}(V)$;
let $R = \mathit{online\text{-}match}(\mathcal{G}, \pi, \sigma)$;
return $(\mathcal{M}(v) \in \mathcal{V}(R) \wedge \sigma(R(\mathcal{M}(v)) \leq t))$
}

(b) Distribution describing the probability that the partner $\mathcal{M}(v) \in U$ of a random vertex $v \in V$ is matched to a vertex of rank at most t.

Figure 2 Two Bernoulli distributions used in the proof of Lemma 1.

Before presenting the proof of Lemma 1, we need to consider how to formally define x_t. It cannot be stated as a probability in the distribution $\mathit{RANKING}(\mathcal{G}, \pi)$. There is no way to refer to the "vertex of rank t in the permutation σ" since $\mathit{RANKING}(\mathcal{G}, \pi)$ is a distribution over subgraphs of $\mathcal{G}$ and the random permutations used to obtain them are not accessible. The solution is to explicitly define the Bernoulli distribution capturing the notion of the vertex of rank t being matched.

$$\mathbb{I}_t \equiv \textbf{do } \{ \ \sigma \leftarrow \mathcal{U}(\mathcal{S}(V)); \ \textbf{let } R = \mathit{online\text{-}match}(\mathcal{G}, \pi, \sigma); \ \mathsf{return}\ (\sigma[t] \in \mathcal{V}(R)) \ \}$$

Then, $1 - x_t$ corresponds to the probability $\mathbb{P}_{\mathbb{I}_t}(\mathsf{False})$.

A key step to achieve the independence of the involved events revolves around not only drawing a random permutation, but also drawing a random vertex and moving it to rank t. This is reflected in the distribution $\mathbb{I}'_t$, given in Fig. 2a. This deceptively simple change ensures the independence of the drawn permutation, i.e. σ, and the actual partner in the perfect matching of the vertex of rank t, i.e. $\mathcal{M}(\sigma[v \mapsto t][t])$ which is the same as $\mathcal{M}(v)$. There is an aspect that is glossed over in the original proof and is intuitively clear: simply drawing a random permutation uniformly at random and the modified way where a random vertex is put at rank t are equivalent. This is shown explicitly in the formal proof.

The final distribution we present here, $\mathbb{I}''_t$ in Fig. 2b, captures the probability that the partner $\mathcal{M}(v)$ of a random $v \in V$ is matched to a vertex of rank at most t.

Proof of Lemma 1. The first step follows from the fact that the permutation σ, in both $\mathbb{I}_t$ and $\mathbb{I}'_t$, and the vertex v are all drawn from uniform distributions.

$$\mathbb{P}_{\mathbb{I}_t}(\mathsf{False}) = \mathbb{P}_{\mathbb{I}'_t}(\mathsf{False})$$

By Lemma 2, if $v \in V$ is unmatched in $\mathit{online\text{-}match}(\mathcal{G}, \pi, \sigma[v \mapsto t])$, then, $\mathcal{M}(v)$ is matched to a vertex of rank at most t in $\mathit{online\text{-}match}(\mathcal{G}, \pi, \sigma)$ (by using $\sigma[v \mapsto t][v \mapsto \sigma(v)] = \sigma$).

$$\leq \mathbb{P}_{\mathbb{I}''_t}(\mathsf{True})$$

Then, the process of drawing a random $v \in V$ and considering $\mathcal{M}(v)$ in $\mathbb{I}''_t$ can be replaced with drawing a random $u \in U$ directly, using the bijection induced by $\mathcal{M}$. This describes the probability that a random $u \in U$ is matched to a vertex of rank at most t. That probability, in turn, is exactly the expected size of the set of online vertices matched to vertices of rank at most t. Formally, these two steps are performed by defining two more Bernoulli distributions capturing the involved concepts. Their definitions are omitted here. Let $\mathbb{I}^*_t$ be the distribution for the set of online vertices matched to vertices of rank at most t.

$$= \frac{1}{n} \mathbb{E}_{O \sim \mathbb{I}^*_t}[|O|]$$

The final step is to express the expected size of the set of online vertices matched to vertices of rank at most t as a sum of the probabilities that the offline vertices of rank up to t are matched. This completes the argument.

$$= \frac{1}{n} \sum_{s=1}^{t} \mathbb{P}_{\mathbb{I}_s}(\mathsf{True}) \qquad \blacktriangleleft$$

Then, we proceed to the main result of this section.

▶ **Theorem 1.** *The competitive ratio of RANKING for instances with a perfect matching of size n is at least $1 - (1 - \frac{1}{n+1})^n$, i.e. $1 - (1 - \frac{1}{n+1})^n \leq \frac{\mathbb{E}_{R \sim RANKING(\mathcal{G},\pi)}[|R|]}{n}$.*

Proof. The expected size of the matching produced by $RANKING(\mathcal{G}, \pi)$ can be rewritten as a sum of the probabilities of the vertices of some rank getting matched.

$$\frac{\mathbb{E}_{R \sim RANKING(\mathcal{G},\pi)}[|R|]}{n} = \frac{1}{n} \sum_{s=1}^{n} \mathbb{P}_{\mathbb{I}_s}(\mathsf{True})$$

The bound obtained on $\mathbb{P}_{\mathbb{I}_s}(\mathsf{False})$ for $1 \leq s \leq n$ in Lemma 1 can be used to bound the sum. This requires a fact on sums provable by induction on n, followed by algebraic manipulation.

$$\geq \frac{1}{n} \sum_{s=1}^{n} \left(1 - \frac{1}{n+1}\right)^s$$

More algebraic manipulation yields the final result.

$$= 1 - \left(1 - \frac{1}{n+1}\right)^n \qquad \blacktriangleleft$$

5 Lifting the Competitiveness to General Bipartite Graphs

Until now, we have shown that *RANKING* satisfies the desired competitive ratio for graphs with a perfect matching. Also, until now, our formalisation closely follows BM's proof. However, in all previous graph-theoretic expositions of the correctness proof of this algorithm [9, 3, 13], as opposed to linear programming-based expositions [4, 6, 19], the authors would stop at the current point, stating, or implicitly assuming, that it is obvious to see how the analysis of *RANKING* for bipartite graphs with perfect matchings extends to general bipartite graphs. The central argument is as follows: it is easy to see that, for a fixed permutation of the offline vertices, if we remove a vertex from a bipartite graph that does not occur in a maximum matching of that graph, then *online-match* will compute a matching that is either one edge smaller or of the same size as the matching *online-match* would compute, given the original graph.

Indeed, BM, who are the authors who give the most detailed account of the graph-theoretic correctness proof of this algorithm, state, as a proof for this fact [3, Lemma 2], that "it is an easy structural observation". In a sense they are correct: in our example, illustrated in Fig. 1, if we remove u_2, it is easy to see that *online-match*'s output size will be only one less than on the original graph. This is because all the matching edges will "cascade" down. This is illustrated in Fig. 1i, showing the blue edges being replaced with the red edges. In this section we mainly formalise this argument. We also formalise another easier, but no less crucial, graph-theoretic part of the proof by BM [3, Lemma 4]. This lemma is used in the probabilistic part of the proof, as stated earlier. In our formalisation we significantly simplified the proof. Before we do so, however, we introduce some necessary background and notions related to paths.

5.1 Alternating Paths, Augmenting Paths, and Berge's Lemma

A list of vertices $[v_1, v_2, \ldots, v_n]$ is a path w.r.t. a graph $\mathcal{G}$ iff $\{v_i, v_{i+1}\} \in \mathcal{G}$ for $1 \leq i < n$. Note: a path $[v_1, v_2, \ldots v_n]$ is always a simple path as we only consider distinct lists. A list of vertices $[v_1, v_2, \ldots, v_n]$ is an alternating path w.r.t. a set of edges E iff for some E'

1. $E' = E$ or $E' \cap E = \emptyset$,
2. $\{v_i, v_{i+1}\} \in E'$ holds for all even numbers i, where $1 \leq i < n$, and
3. $\{v_i, v_{i+1}\} \notin E'$ holds for all odd numbers i, where $1 \leq i \leq n$.

We call a list of vertices $[v_1, v_2, \ldots, v_n]$ an augmenting path w.r.t. a matching $\mathcal{M}$ iff $[v_1, v_2, \ldots, v_n]$ is an alternating path w.r.t. $\mathcal{M}$ and $v_1, v_n \notin \mathcal{V}(\mathcal{M})$. If $\mathcal{M}$ is a matching w.r.t. a graph $\mathcal{G}$, we call the path an augmenting path w.r.t. to the pair $\langle \mathcal{G}, \mathcal{M} \rangle$. Also, for two sets s and t, $s \oplus t$ denotes the symmetric difference of the two sets.

A central result in matching theory is Berge's lemma, which gives an algorithmically useful characterisation of a maximum cardinality matching.

▶ **Theorem 2** (Berge's Lemma). *For a graph $\mathcal{G}$, a matching $\mathcal{M}$ is maximum w.r.t. $\mathcal{G}$ iff there is not an augmenting path γ w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$.*

We use a formalisation of the above concepts and Berge's Lemma by Abdulaziz et al. [1].

5.2 *online-match*'s Behaviour after Removing a Vertex

Now that we have all the necessary machinery, we can discuss the formalisation of the correctness of *RANKING* for general bipartite graphs. The central claim to show is stated in the following lemma, which is a restatement of Lemma 2 in BM's paper. It states what happens to the result of *online-match* when a vertex is removed from the graph.

▶ **Lemma 3.** *Let $\mathcal{G}$ be a bipartite graph w.r.t. the lists σ and π. Consider a vertex $u \in \pi$. Let $\mathcal{H}$ be $\mathcal{G} \setminus \{u\}$. We have that either online-match$(\mathcal{G}, \pi, \sigma)$ = online-match$(\mathcal{H}, \pi, \sigma)$ or online-match$(\mathcal{G}, \pi, \sigma) \oplus$ online-match$(\mathcal{H}, \pi, \sigma)$ can be ordered into an alternating path w.r.t. online-match$(\mathcal{G}, \pi, \sigma)$ and online-match$(\mathcal{H}, \pi, \sigma)$, and that path starts at u.*

The above lemma was never proved by any of the previous expositions of the combinatorial argument for the algorithm's correctness. BM's exposition is an exception, where there is at least a graphical example, showing what happens when we remove a vertex before running *online-match*. A version of that graphical argument can be seen in Fig. 1. Fig. 1h shows the matching computed by the algorithm on the original graph, and Fig. 1i shows the difference in the computed matching if a vertex from the online side of the graph is removed.[3] As shown, when the vertex is removed, the matched edges "cascade downwards", where the original matching edges, shown in blue, are replaced with the red edges. The lemma states that the symmetric difference between the two computed matchings is always an alternating path, w.r.t. both the old and the new matchings, if there is any difference. When looking at the graphical illustration this is obvious. However, when formalising that argument, many challenges manifest themselves.

The first challenge is the characterisation of the path that constitutes the difference between the two matchings. This characterisation has to, among other things, make formal proofs by induction manageable. To do so, we had to formulate this characterisation *not*

[3] The lemma above is stated for an online vertex being removed, while in the formalisation an offline vertex is removed. This highlights an important concept in many of the proofs: the interchangeability of the offline and online vertices for fixed orders σ and π.

Listing 2: Formalising *shifts-to* in Isabelle/HOL

```
definition "shifts-to G M u v v' π σ ≡
 u ∈ set π ∧ v' ∈ set σ ∧ index σ v < index σ v' ∧ {u,v'} ∈ G
 ∧ (∄u'. index π u' < index π u ∧ {u',v'} ∈ M) ∧
 (∀v''. (index σ v < index σ v'' ∧ index σ v'' < index σ v')
 ⟶ ({u,v''} ∉ G ∨ (∃u'. index π u' < index π u ∧ {u',v''} ∈ M)))
```

Listing 3: Formalising zig-zag in Isabelle/HOL.

```
function zig :: "'a graph ⇒ 'a graph ⇒ 'a ⇒ 'a list ⇒ 'a list ⇒ 'a list"
and zag :: "'a graph ⇒ 'a graph ⇒ 'a ⇒ 'a list ⇒ 'a list ⇒ 'a list" where
  proper-zig: "zig G M v π σ = v # (
                    if ∃u. {u,v} ∈ M
                    then zag G M (THE u. {u,v} ∈ M) π σ
                    else [])" if "matching M"
| no-matching-zig: "zig - M v - - - = [v]" if "¬matching M"

| proper-zag: "zag G M u π σ =  u # (if ∃v. {u,v} ∈ M
                      then
                      (let v = THE v. {u,v} ∈ M in (
                        if ∃v'. shifts-to G M u v v' π σ
                        then zig G M (THE v'. shifts-to G M u v v' π σ) π σ
                        else [])
                      )
                      else []
                     )" if "matching M"
| no-matching-zag: "zag - M v - - - = [v]" if "¬matching M"
```

recursively on the given bipartite graph, i.e. the given bipartite graph should not change across different recursive calls. Otherwise, proving anything about the path would involve a complicated induction on the given bipartite graph.

To define that path, we first introduce a concept relating two vertices on the online side. We state v *shifts-to* v' iff

1. v occurs before v' in the offline permutation σ,
2. v is matched to some u,
3. v' is not matched to any vertex that occurs before u in π, and
4. any vertex $v'' \in N_{\mathcal{G}}(u)$ occurring between v and v' in σ is matched by *online-match* to a vertex occurring before u in the arrival order π.

Intuitively, this means that, if v is removed from the graph, then v' would be matched to u by *online-match*. Our formalisation of this definition can be found in Listing 2. Note: the omitted arguments in the text, $\mathcal{G}$, $\mathcal{M}$, π, σ, and u are usually clear from the context.

Now that we are done with the definition of *shifts-to*, we are ready to describe our characterisation of the path whose edges form the symmetric difference of the two matchings computed by *online-match*. We characterise it using the following functions:

$$zig(\mathcal{G},\mathcal{M},v,\pi,\sigma) \equiv \begin{cases} v \# zag(\mathcal{G},\mathcal{M},u,\pi,\sigma) & \text{if } \{v,u\} \in \mathcal{M} \\ [v] & \text{otherwise} \end{cases}$$

$$zag(\mathcal{G},\mathcal{M},u,\pi,\sigma) \equiv \begin{cases} u \# zig(\mathcal{G},\mathcal{M},v',\pi,\sigma) & \text{if } \{v,u\} \in \mathcal{M}, \text{ for some } v, \text{ and } v \textit{ shifts-to } v' \\ [u] & \text{otherwise} \end{cases}$$

As the names of the functions indicate, the path zig-zags between the online and the offline sides of the graph, going down the online ordering. This is indicated in Fig. 1j. The formalisation of *zig-zag* is given in Listing 3. Note that the formalisation has extra cases for when the second argument is not a matching: this is to ensure termination, which is not straightforward, as the definite descriptions are not well-defined in these cases. The

Listing 4: Formalising the specification of *online-match*'s output in Isabelle/HOL.

```
definition ranking−matching :: "'a graph ⇒ 'a graph ⇒ 'a list ⇒ 'a list ⇒ bool" where
  "ranking−matching G M π σ ≡ graph−matching G M ∧
    bipartite G (set π) (set σ) ∧ maximal−matching G M ∧
    (∀u v v'. ({u,v} ∈ M ∧ {u,v'} ∈ G ∧ index σ v' < index σ v) ⟶
      (∃u'. {u',v'} ∈ M ∧ index π u' < index π u)) ∧
    (∀u v u'. ({u,v} ∈ M ∧ {u',v} ∈ G ∧ index π u' < index π u) ⟶
      (∃v'. {u',v'} ∈ M ∧ index σ v' < index σ v))"
```

termination relation encodes the intuition that, while zig-zagging, the path also goes down the ordering of online vertices. More formally, because this is a mutually recursive function, we have to provide an order that relates the argument passed to recursive calls of *zag* from *zig* and the other way around. For evaluating $zig(\mathcal{G}, \mathcal{M}, v, \pi, \sigma)$, we need a call to $zag(\mathcal{G}, \mathcal{M}, u, \pi, \sigma)$, in which case the relation holds iff v and u satisfy

1. $\{v, u\} \in$ *online-match*$(\mathcal{G}, \pi, \sigma)$ and
2. if there is v', s.t. v *shifts-to* v', then $\sigma(v) < \sigma(v')$.

For evaluating $zag(\mathcal{G}, \mathcal{M}, u, \pi, \sigma)$, we need a call to $zig(\mathcal{G}, \mathcal{M}, v', \pi, \sigma)$, in which case the relation holds iff u and v' satisfy

1. there is v s.t. $\{v, u\} \in$ *online-match*$(\mathcal{G}, \pi, \sigma)$ and
2. v *shifts-to* v' and $\sigma(v) < \sigma(v')$.

Another challenge for formalising the proof of Lemma 3 is devising a non-recursive characterisation of the properties of the matching computed by *online-match*, which would be enough for proving the lemma, yet more abstract than the actual specification of the algorithm. This characterisation can be intuitively described as follows: $\mathcal{M}$ is a *ranking-matching* w.r.t. $\mathcal{G}$, σ, and π iff

1. $\mathcal{G}$ is bipartite w.r.t. σ and π,
2. $\mathcal{M}$ is a maximal matching w.r.t. $\mathcal{G}$,
3. every vertex from $u \in \pi$ is matched to the unmatched vertex from σ at u's arrival, to which it is connected, with the lowest rank in σ, and
4. no vertex from σ "refuses" to be matched.

The formal specification is given in Listing 4. It should be clear that the following properties hold for *ranking-matching*.

▶ **Proposition 1.** *If $\mathcal{G}$ be a bipartite graph w.r.t. σ and π, then*

1. *online-match$(\mathcal{G}, \pi, \sigma)$ is a ranking-matching w.r.t. $\mathcal{G}$, σ, and π,*
2. *if $\mathcal{M}$ is a ranking-matching w.r.t. $\mathcal{G}$, σ, and π, then it is a ranking-matching w.r.t. $\mathcal{G}$, π, and σ, and*
3. *if $\mathcal{M}$ and $\mathcal{M}'$ are both ranking-matchings w.r.t. $\mathcal{G}$, σ, and π, then $\mathcal{M} = \mathcal{M}'$.*

This specification of *online-match* makes our proofs about *online-match* much simpler, as it allows us to gloss over many of the computational details of the algorithm. In particular, it allows us to avoid nested inductions, especially when using the I.H. of Lemma 3.

Now that we have characterised the difference between the matchings computed by *online-match* before and after removing a vertex, as well as the main properties satisfied by matchings computed by *online-match*, we are ready to present the proof that the competitiveness for bipartite graphs with perfect matchings lifts to general bipartite graphs. There are two main ideas to our proof. The first one is that we show that the output of *zig*, for some online vertex u, which is matched to an offline vertex v, stays the same when offline vertices are removed from the graph and the matching, if those offline vertices are all ranked

lower than v. Graphically, this is clear. For instance, in Fig. 1j, if we remove the vertex v_1 from the graph and the matching, the result of *zig* applied to u_2, w.r.t. to the modified graph and matching, will be the same as its output w.r.t. the old graph and matching.

▶ **Lemma 4.** *Let $\mathcal{G}$ be a bipartite graph w.r.t. σ and π. Consider a vertex $u \in \pi$, s.t. there is v, where $\{v, u\} \in \mathcal{M}$. Consider a set of vertices $U' \subseteq \pi$, s.t. for all $u' \in U'$ we have that $\pi(u') < \pi(u)$. Let $\mathcal{M}$ be a ranking-matching w.r.t. $\mathcal{G}$, π, and σ. We have that $zig(\mathcal{G}, \mathcal{M}, u, \sigma, \pi) = zig(\mathcal{G} \setminus U', \mathcal{M} \setminus U', u, \sigma, \pi)$ and $zag(\mathcal{G}, \mathcal{M}, v, \sigma, \pi) = zag(\mathcal{G} \setminus U', \mathcal{M} \setminus U', v, \pi, \sigma)$.*

We do not prove this lemma here: the proof depends on an involved case analysis of the behaviour of *shifts-to*, and we describe below similar case analyses, which convey the difficulty of translating such obvious graphical arguments into proofs. Interested readers, however, should refer to the accompanying formal proof.

The second idea is that we exploit the symmetry between the online and the offline vertices. This is encoded in the following relationship between *zig* and *zag*.

▶ **Lemma 5.** *Let $\mathcal{G}$ be a bipartite graph w.r.t. σ and π. Consider a vertex $u \in \pi$. Let $\mathcal{H}$ be $\mathcal{G} \setminus \{u\}$. Let $\mathcal{M}$ be a ranking-matching w.r.t. $\mathcal{G}$, π, and σ, and $\mathcal{M}'$ be a ranking-matching w.r.t. $\mathcal{H}$, σ, and π. Let v be a vertex s.t. $\{v, u\} \in \mathcal{M}$. We have that $zig(\mathcal{H}, \mathcal{M}', v, \pi, \sigma) = zag(\mathcal{G}, \mathcal{M}, v, \sigma, \pi)$.*

Before we discuss the proof, we first show a graphical argument of why the lemma holds. Fig. 1j and 1k show an example of how *zig* and *zag* would return the same list of vertices if invoked on the same vertex once on the offline side, and another time on the online side. In the first configuration, $zag(\mathcal{G}, \mathcal{M}, v_2, \sigma, \pi)$ chooses u_3, because in $\mathcal{M}$, we have that v_2 is matched to u_2, and u_2 *shifts-to* u_3. Then the rest of the recursive calls proceed as shown in the figure. When the online and offline sides are flipped, as shown in Fig. 1k, $zig(\mathcal{H}, \mathcal{M}', v_2, \pi, \sigma)$, where $\mathcal{H}$ denotes $\mathcal{G} \setminus \{u_2\}$, will also choose u_3 because, this time, it will be matched to v_2 in $\mathcal{M}'$, which is a *ranking-matching* for $\mathcal{H}$. As we will see in the proof, this graphical argument is much shorter than the corresponding textual proof, let alone the formal proof.

Proof. Our proof is by strong induction on the index of v. Let all the variable names in the I.H. be barred, e.g. the graph is $\overline{\mathcal{G}}$. Our proof is done by case analysis. We consider 3 cases:

1. we have vertices u', v', s.t. $\{v, u'\} \in \mathcal{M}'$ and $\{u', v'\} \in \mathcal{M}$,
2. we have a vertex u', s.t. $\{v, u'\} \in \mathcal{M}'$ and there is no v' s.t. $\{u', v'\} \in \mathcal{M}$, and
3. there is no vertex u', s.t. $\{v, u'\} \in \mathcal{M}'$.

We focus on the first case, as that is the one where we employ the I.H. To apply the I.H., we use the following assignments of the quantified variables. $\overline{\mathcal{G}} \mapsto \mathcal{G} \setminus \{u, v\}$, $\overline{\pi} \mapsto \pi$, $\overline{\sigma} \mapsto \sigma$, $\overline{u} \mapsto u'$, $\overline{v} \mapsto v'$, $\overline{\mathcal{M}} \mapsto \mathcal{M} \setminus \{u, v\}$, and $\overline{\mathcal{M}'} \mapsto \mathcal{M}' \setminus \{v, u'\}$. From the I.H., we get $zig(\overline{\mathcal{H}}, \overline{\mathcal{M}}', \overline{v}, \overline{\pi}, \overline{\sigma}) = zag(\overline{\mathcal{G}}, \overline{\mathcal{M}}, \overline{u}, \overline{\sigma}, \overline{\pi})$. This proof is then finished by Lemma 4. ◀

We are now ready to prove a lemma that immediately implies Lemma 3.

▶ **Lemma 6.** *Let $\mathcal{G}$ be a bipartite graph w.r.t. σ and π. Consider a vertex $u \in \pi$. Let $\mathcal{H}$ be $\mathcal{G} \setminus \{u\}$. Let $\mathcal{M}$ be a ranking-matching w.r.t. $\mathcal{G}$, σ, and π, and $\mathcal{M}'$ be a ranking-matching w.r.t. $\mathcal{H}$, σ, and π. We have that $\mathcal{M} \oplus \mathcal{M}' = zig(\mathcal{G}, \mathcal{M}, u, \sigma, \pi)$[4] or $\mathcal{M} = \mathcal{M}'$.*

[4] We abuse the notation: although $zig(\mathcal{G}, \mathcal{M}, u, \sigma, \pi)$ is the list of vertices in the path, we use it here to denote the edges in the path.

Proof. Our proof is by strong induction on $|\mathcal{G}|$. Again, let all the variable names in the I.H. be barred. We consider two cases, either $u \notin \mathcal{V}(\mathcal{M})$ or $u \in \mathcal{V}(\mathcal{M})$. In the former case, the lemma follows immediately, since *online-match* will compute the same matching.

For the second case, we instantiate the I.H. as follows: $\overline{\mathcal{G}} \mapsto \mathcal{G} \setminus \{u\}$, $\overline{\mathcal{M}} \mapsto \mathcal{M}'$, $\overline{\mathcal{M}'} \mapsto \mathcal{M} \setminus \{v, u\}$, $\overline{\pi} \mapsto \sigma$, $\overline{\sigma} \mapsto \pi$, and $\overline{u} \mapsto v$, where v is some vertex s.t. $\{v, u\} \in \mathcal{M}$, which must exist since $u \in \mathcal{V}(\mathcal{M})$.[5] To show that the I.H. is usable in this case, we need to show that:

1. $\overline{\mathcal{M}}$ is a *ranking-matching* w.r.t. $\overline{\mathcal{G}}$, $\overline{\pi}$, and $\overline{\sigma}$, and
2. $\overline{\mathcal{M}'}$ is a *ranking-matching* w.r.t. $\overline{\mathcal{H}}$, $\overline{\pi}$, and $\overline{\sigma}$

. The first requirement follows from the assumption that $\mathcal{M}'$ is *ranking-matching* w.r.t. $\mathcal{H}$, σ, and π, and the fact that *ranking-matching* is commutative w.r.t. the left and right parties of the given graph. The second requirement follows from a property of *ranking-matching*, which we do not prove here, stating that for any $\mathcal{M}$ that is a *ranking-matching* w.r.t. $\mathcal{G}$, σ, and π, and for any $e \in \mathcal{M}$, $\mathcal{M} - \{e\}$ is a *ranking-matching* w.r.t. $\mathcal{G} \setminus e$, σ, and π.

Then, from the I.H. and since we know that $v \in \mathcal{V}(\mathcal{M})$, we have that either

1. $\overline{\mathcal{M}} = \overline{\mathcal{M}}'$ or
2. $\overline{\mathcal{M}} \oplus \overline{\mathcal{M}'} = \mathit{zig}(\overline{\mathcal{G}}, \overline{\mathcal{M}}, \overline{u}, \overline{\sigma}, \overline{\pi})$.

In the former case, we have that $\mathcal{M}' = \mathcal{M} \setminus \{u, v\}$, so v was not matched to anything in the graph, after removing u. This means that there is no u' for v s.t. u *shifts-to* u', which means that $\mathit{zig}(\mathcal{G}, \mathcal{M}, u, \sigma, \pi) = [u, v]$. From that, we have the lemma proved for this case, since $\mathcal{M} \oplus \mathcal{M}' = \{v, u\}$.

In the second case, we have that $\overline{\mathcal{M}} \oplus \overline{\mathcal{M}'} = \mathit{zig}(\mathcal{G} \setminus \{u\}, \mathcal{M}', v, \pi, \sigma)$. From Lemma 5, we have $\mathit{zig}(\mathcal{G} \setminus \{v\}, \mathcal{M}', v, \pi, \sigma) = \mathit{zag}(\mathcal{G}, \mathcal{M}, v, \sigma, \pi)$. From the definition of *zig* and since $\{u, v\} \in \mathcal{M}$, the lemma follows for this case. ◀

Proof of Lemma 3. Lemma 3 follows immediately from Lemma 6 and from Proposition 1. ◀

5.3 Finishing the Proof

The next step in our proof is to generalise the previous analysis to address the case when the removed vertex is from the offline side of the graph. Although this is not considered by any of the previous expositions, this generalisation is crucial for proving the competitive ratio for general bipartite graphs, i.e. graphs that do not have a perfect matching.

▶ **Lemma 7.** *Let $\mathcal{G}$ be a bipartite graph w.r.t. σ and π. Consider a vertex $v \in \sigma$. Let $\mathcal{H}$ be $\mathcal{G} \setminus \{v\}$. Let $\mathcal{M}$ be a ranking-matching w.r.t. $\mathcal{G}$, σ, and π, and $\mathcal{M}'$ be a ranking-matching w.r.t. $\mathcal{H}$, σ, and π. We have that $\mathcal{M} \oplus \mathcal{M}' = \mathit{zig}(\mathcal{G}, \mathcal{M}, v, \pi, \sigma)$ or $\mathcal{M} = \mathcal{M}'$.*

The proof of this lemma is very similar to that of Lemma 3. However, we are able to reuse all our lemmas that exploit the symmetry of the offline and online sides of the graphs, so there is not much redundancy in our proofs.

Until now, we have primarily focused on the *structural* difference between matchings computed by *online-match* before and after removing a vertex from the original graph. The next step in the proof is to use that to reason about the competitiveness of *online-match* for general bipartite graphs. The first step is proving the following lemma.

[5] The instantiation of $\overline{\mathcal{H}}$ follows implicitly from the other instantiations.

Listing 5: Formalising the specification of *make-perfect*'s output in Isabelle/HOL.

```
function make_perfect_matching :: "'a graph ⇒ 'a graph ⇒ 'a graph" where
  "make_perfect_matching G M = (
    if (∃x. x ∈ Vs G ∧ x ∉ Vs M)
    then make_perfect_matching (G \ {SOME x. x ∈ Vs G ∧ x ∉ Vs M}) M
    else G
  )
  " if "finite G"
| "make_perfect_matching G M = G" if "infinite G"
```

▶ **Lemma 8.** *Let $\mathcal{G}$ be a bipartite graph w.r.t. σ and π. Consider a vertex x. Let $\mathcal{H}$ be $\mathcal{G} \setminus \{x\}$. Let $\mathcal{M}$ be a ranking-matching w.r.t. $\mathcal{G}$, σ, and π, and $\mathcal{M}'$ be a ranking-matching w.r.t. $\mathcal{H}$, σ, and π. We have that $|\mathcal{M}'| \leq |\mathcal{M}|$.*

Proof. Our proof is by case analysis. The first case is when $x \notin \mathcal{V}(\mathcal{M})$. In this case we will have that $\mathcal{M} = \mathcal{M}'$, which finishes our proof.

The second case is when $x \in \mathcal{V}(\mathcal{M})$. In this case, we have two sub-cases: either $x \in \pi$ or $x \in \sigma$. We only describe the first case here and the second is symmetric. Our proof is by contradiction, i.e. assuming $|\mathcal{M}'| > |\mathcal{M}|$. From Lemma 6, we have that $\mathcal{M} \oplus \mathcal{M}' = \mathit{zig}(\mathcal{G}, \mathcal{M}, u, \sigma, \pi)$. Also note that, from Berge's lemma, we will have that a subsequence of $\mathit{zig}(\mathcal{G}, \mathcal{M}, u, \sigma, \pi)$ is an augmenting path w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$. We know from the definition of an augmenting path that both its first and last vertices are not in the matching it augments. Accordingly, we have that the first and last vertices of that subsequence of $\mathit{zig}(\mathcal{G}, \mathcal{M}, u, \sigma, \pi)$ are not in $\mathcal{M}$. This is a contradiction, because all vertices in $\mathit{zig}(\mathcal{G}, \mathcal{M}, u, \sigma, \pi)$, except possibly the last one, are in $\mathcal{V}(\mathcal{M})$. ◀

Lastly, we show that, given a bipartite graph $\mathcal{G}$ and a maximum cardinality matching $\mathcal{M}$ for that graph, we can recursively remove the vertices that do not occur in $\mathcal{M}$. To do that we define a recursive function, *make-perfect*, to remove these vertices and then prove the following lemma by computation induction, using the computation induction principle corresponding to *make-perfect*. Listing 5 shows the formalisation of that function.

▶ **Lemma 9.** *Let $\mathcal{G}$ be a bipartite graph w.r.t. σ and π. Let $\mathcal{M}$ be a ranking-matching w.r.t. $\mathcal{G}$, σ, and π, and $\mathcal{M}'$ be a ranking-matching w.r.t. make-perfect$(\mathcal{G}, \mathcal{M})$, σ, and π. We have that $|\mathcal{M}'| \leq |\mathcal{M}|$.*

This last lemma leads to the final theorem below.

▶ **Theorem 3.** *Let $\mathcal{G}$ be a bipartite graph w.r.t. σ and π. Let $\mathcal{M}$ be a maximum cardinality matching for $\mathcal{G}$. We have that $1 - (1 - \frac{1}{|\mathcal{M}|+1})^{|\mathcal{M}|} \leq \mathbb{E}_{R \sim RANKING(\mathcal{G},\pi)}[|R|]/|\mathcal{M}|$.*

Proof. This follows immediately from Lemma 9, Theorem 1, and the fact that the size of a maximum cardinality matching for *make-perfect*$(\mathcal{G}, \mathcal{M})$ is the same as the size of $\mathcal{M}$, if $\mathcal{M}$ is a maximum cardinality matching for $\mathcal{G}$. ◀

5.4 Proving Lemma 2

Until now we have not discussed how we formalised Lemma 2 – we believe it better fits here as its proof is a combinatorial argument. Graphically, Fig. 3 shows some instances of Lemma 2 for $v = v_3$ and $\mathcal{M}(v_3) = u_1$. No matter where v_3 is put, u_1 is always matched to a vertex of rank at most 3. BM prove this Lemma by stating that the difference, if any, between the matchings computed by *online-match* before and after moving the offline vertex is also

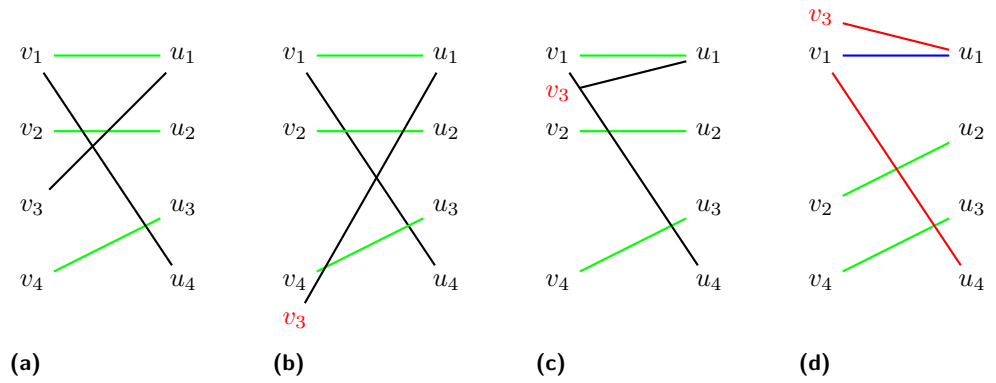


Figure 3 Illustrating Lemma 2, where $v = v_3$, and $\mathcal{M}(v_3) = u_1$. Initially (3a), v_3 is unmatched. Moving it further down in the ranking (3b) does not change the partner of u_1. Moving v_3 up in the ranking can either (3c) also leave u_1 untouched, or (3d) change the partner of u_1.

Listing 6: The formalisation of Theorem 4

```
abbreviation matching-instance-nat :: "nat ⇒ (nat × nat) graph" where
  "matching-instance-nat n ≡ {{(0,k),(Suc 0,k)} |k. k < n}"

definition ranking-instances-nat :: "nat ⇒ (nat × nat) graph set" where
  "ranking-instances-nat n ≡ {G. max-card-matching G (matching-instance-nat n) ∧
     finite G ∧ G ⊆ {{(0,k),(Suc 0,l)} |k l. k < 2*n ∧ l < 2*n}}"

definition arrival-orders :: "(nat × nat) graph ⇒ (nat × nat) list set" where
  "arrival-orders G ≡ permutations-of-set {(Suc 0,l) |l. ∃k. {(0,k),(Suc 0,l)} ∈ G}"

definition offline-vertices :: "(nat × nat) graph ⇒ (nat × nat) set" where
  "offline-vertices G ≡ {(0, k) |k. ∃l. {(0, k),(Suc 0, l)} ∈ G}"

definition comp-ratio-nat where
  "comp-ratio-nat n ≡
     Min {Min {measure-pmf.expectation
                 (wf-ranking.ranking-prob G π (offline-vertices G)) card
                    / card (matching-instance-nat n)
                  |π. π ∈ arrival-orders G}
                     | G. G ∈ ranking-instances-nat n}"

theorem comp-ratio-limit':
  assumes "convergent comp-ratio-nat"
  shows "1 − exp(−1) ≤ (lim comp-ratio-nat)"
```

an alternating path, where the ranks of the offline vertices traversed by that path increase. Again, like other combinatorial parts of the analysis, graphically this is clearly evident: Fig. 3d shows the difference between *online-match*$(\mathcal{G}, \pi, \sigma)$ and *online-match*$(\mathcal{G}, \pi, \sigma[v_3 \mapsto 1])$. The blue edge was removed from the original matching, and the two red edges are added instead. The three edges form an alternating path w.r.t. to the original matching.

However, to formalise this argument would be as difficult as for Lemma 3. Indeed, we found out that there is no reason to construct the entire difference between the two matchings just to reason about the rank of the vertex v_i to which u is matched in *online-match*$(\mathcal{G}, \pi, \sigma[v \mapsto i])$. With this approach, the lemma follows almost immediately from the specification *ranking-matching*. Hence, the formal proof is much shorter than BM's approach.

6 The Competitive Ratio in the Limit

BM claim that the competitive ratio tends to $1-1/e$ if the matching's size tends to infinity. The main complication of showing that is to show that the competitive ratio converges, which they do not address at all. We formalised the following.

▶ **Theorem 4.** *Let* $\mathcal{M}_n$ *denote* $\{\{(0,k),(1,k)\} \mid 1 \leq k \leq n\}$. *Let* Γ_n *denote graphs in the power set of* $\{\{(0,k),(1,l)\} \mid 1 \leq k,l \leq 2n\}$ *and that have* $\mathcal{M}_n$ *as a maximum cardinality matching. Let* π_n *denote* $\mathcal{S}(\{(1,k) \mid 1 \leq k \leq 2n\})$. *If* $\mathcal{Q}_n$ *converges, then* $\mathcal{Q}_n$ *tends to* $1-1/e$ *as* n *tends to* ∞, *where* $\mathcal{Q}_n$ *denotes* $\mathsf{min}_{(\mathcal{G},\pi)\in\Gamma_n\times\pi_n}\mathbb{E}_{R\sim RANKING(\mathcal{G},\pi)}[|R|]/|\mathcal{M}_n|$.

We only prove the limit for a specific set of bipartite graphs, namely, Γ_n. We conjecture that Γ_n is isomorphic to the set of all bipartite graphs with maximum cardinality matchings of size n. Despite it being trivial, it was impressive that the part of the proof of this lemma which pertains to arithmetic manipulation was almost completely automated using Eberl's tool [5]. The other part of the proof was to show that Γ_n is finite, which was tedious.

The more interesting part would be to show that $\mathcal{Q}_n$ converges. In BM, they do not prove that, yet they do not have it as an assumption in their theorem statement. One way to show that this assumption holds is to use the theorem by KVV showing that no online algorithm for bipartite matching has a better competitive ratio that $1-1/e$. However, formalising that theorem is beyond the scope of our project.

7 Discussion

KVV's paper on online bipartite matching was a seminal result in the theory of online algorithms and matching. Its interesting theoretical properties, together with the emergence of online matching markets have inspired a lot of generalisations to other settings, e.g. for weighted vertices [2], online bipartite b-matching [12], the AdWords market [15], which models the multi-billion dollars industry online advertising industry, and general graphs [7], which models applications like ride-sharing. All of this means an improved understanding of the theory of online-matching, and especially RANKING, is of great interest.

Indeed, as stated earlier, multiple authors studied the analysis of RANKING. We mention here the most relevant five approaches:

1. Goel and A. Mehta [9], tried to simplify the proof and fill in a "hole" in KVV's original proof, in particular in the proof of Lemma 6 in KVV's original paper,
2. Birnbaum and C. Mathieu [3] also provided a simple, primarily combinatorial, proof for RANKING,
3. Devanur, Jain, and Kleinberg [4] whose main contribution was to model the algorithm as a primal-dual algorithm, in an attempt to unify the approaches for analysing the unweighted, vertex-weighted, and the AdWords problem,
4. Eden, Feldman, Fiat, and Segal [6], who tried to simplify the proof by using approaches from theory of economics, and finally
5. Vazirani [19], who tried to simplify the proof of RANKING, in an attempt to use RANKING, or a generalisation of it, to solve AdWords.

However, despite all of these attempts, the proof of RANKING's correctness is still considered difficult to understand, e.g. Vazirani's latest trial to generalize it had a critical non-obvious flaw in the combinatorial part of the analysis [19], which took months of reviewing to find out.

We believe this formalisation serves two purposes. First, it is yet another attempt to improve the understanding of this algorithm's analysis. From that perspective, our work achieved two things.

1. It further clarified the complexity of the combinatorial argument underlying the analysis of this algorithm by providing a detailed proof for how one could generalise the competitiveness of the algorithm from bipartite graphs with perfect matchings to general bipartite graphs. We note that this part of the analysis is analogous to the "no-surpassing

property" in Vazirani's work [19], which is where his attempt to generalise RANKING to AdWords fell apart, further confirming our findings regarding the complexity of this part of the analysis.

2. We significantly simplified the analysis of the consequences of changing the ranking of an offline vertex.

Another outcome of this project is interesting from a formalisation perspective. It further confirmed the previously reported observation that it is particularly hard to formalise graphical or geometric arguments and concepts. E.g. verbally, let alone formally, encoding the intuition behind *shifts-to*, which is a primarily graphical concept, is extremely cumbersome. We hypothesise that this is an inherent complexity in graphical concepts and arguments which manifests itself when the graphical argument is put into prose.

One point which we believe would particularly benefit from further study is that of modelling online computation. In its full generality, online computation is computation where the algorithm has access only to parts of the input, which arrive serially, but not the whole input. The way we model our algorithm is ad-hoc and does not capture that essence of online computation in its full generality. It remains an interesting question how one can model online computation, more generally. In addition to the theoretical interest, a satisfactory answer to that question is essential if one is to show that the competitive ratio of RANKING is optimal for online algorithms, which is a main result of KVV.

References

1 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.

2 Gagan Aggarwal, Gagan Goel, Chinmay Karande, and Aranyak Mehta. Online Vertex-Weighted Bipartite Matching and Single-bid Budgeted Allocations. In *The 22nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2011.

3 Benjamin E. Birnbaum and Claire Mathieu. On-line bipartite matching made simple. *SIGACT News*, 2008.

4 Nikhil R. Devanur, Kamal Jain, and Robert D. Kleinberg. Randomized Primal-Dual Analysis of RANKING for Online Bipartite Matching. In *The 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, January 2013.

5 Manuel Eberl. Verified Real Asymptotics in Isabelle/HOL. In *The International Symposium on Symbolic and Algebraic Computation (ISSAC)*, 2019.

6 Alon Eden, Michal Feldman, Amos Fiat, and Kineret Segal. An Economics-Based Analysis of RANKING for Online Bipartite Matching. In *Symposium on Simplicity in Algorithms (SOSA)*, January 2021.

7 Buddhima Gamlath, Michael Kapralov, Andreas Maggiori, Ola Svensson, and David Wajc. Online Matching with General Arrivals, April 2019. `arXiv:1904.08255`.

8 Michele Giry. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis*, 1982.

9 Gagan Goel and Aranyak Mehta. Online budgeted matching in random input models with applications to Adwords. In *The 19th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2008.

10 Johannes Hölzl. *Construction and Stochastic Applications of Measure Spaces in Higher-Order Logic*. PhD thesis, Technical University Munich, 2013.

11 John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM J. Comput.*, 1973.

12 Bala Kalyanasundaram and Kirk Pruhs. An optimal deterministic algorithm for online b-matching. *Theor. Comput. Sci.*, 2000.

13 R. M. Karp, U. V. Vazirani, and V. V. Vazirani. An optimal algorithm for on-line bipartite matching. In *The 22nd ACM Symposium on Theory of Computing (STOC)*, 1990.

14 Harold W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 1955.

15 Aranyak Mehta, Amin Saberi, Umesh V. Vazirani, and Vijay V. Vazirani. AdWords and generalized online matching. *J. ACM*, 2007.

16 Milena Mihail and Thorben Tröbst. Online Matching with High Probability, December 2021. arXiv:2112.07228.

17 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.

18 Vijay V. Vazirani. Online Bipartite Matching and Adwords, February 2022. arXiv:2107.10777.

19 Vijay V. Vazirani. Online Bipartite Matching and Adwords (Invited Talk). In *The 47th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, 2022.

Fast, Verified Computation for Candle

Oskar Abrahamsson
Chalmers University of Technology, Gothenburg, Sweden

Magnus O. Myreen
Chalmers University of Technology, Gothenburg, Sweden

Abstract

This paper describes how we have added an efficient function for computation to the kernel of the Candle interactive theorem prover. Candle is a CakeML port of HOL Light which we have, in prior work, proved sound w.r.t. the inference rules of the higher-order logic. This paper extends the original implementation and soundness proof with a new kernel function for fast computation. Experiments show that the new computation function is able to speed up certain evaluation proofs by several orders of magnitude.

2012 ACM Subject Classification Software and its engineering → Software verification

Keywords and phrases Prover soundness, Higher-order logic, Interactive theorem proving

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.4

Supplementary Material *Software*: `https://cakeml.org/candle`
Software (state at time of writing): `github.com/CakeML/cakeml/tree/90e158ecb6`

Funding *Oskar Abrahamsson*: Swedish Foundation for Strategic Research.
Magnus O. Myreen: Swedish Foundation for Strategic Research.

Acknowledgements We want to thank Jeremy Avigad, John Harrison, Tobias Nipkow and Freek Wiedijk for feedback we received when the first author prepared this as a chapter for his PhD thesis [1]. We thank Thomas Sewell for showing us how to benchmark in-logic evaluation in Isabelle/HOL.

1 Introduction

Interactive theorem provers (ITPs) include facilities for computing within the hosted logic. To illustrate what we mean by such a feature, consider the following function, sum, which sums a list of natural numbers:

$$\mathsf{sum}\ xs \stackrel{\mathsf{def}}{=} \mathsf{if}\ xs = [] \ \mathsf{then}\ 0\ \mathsf{else}\ \mathsf{hd}\ xs + \mathsf{sum}\ (\mathsf{tl}\ xs)$$

A facility for computing within the logic can be used to automatically produce theorems such as the following, where sum [5; 9; 1] was given as input, and the following equation is the output, showing that the input reduces to 15:

$$\vdash \mathsf{sum}\ [5;\ 9;\ 1] = 15 \tag{1}$$

The ability to compute such equations in ITPs is essential for use of verified decision procedures, for proving ground cases in proofs, and for running a parser, pretty printer or even compiler inside the logic for a smaller trusted computing base (TCB).

Higher-order logic (HOL) does not have a primitive rule for (or notion of) computation. Instead, HOL ITPs such as HOL Light [11], HOL4 [13], and Isabelle/HOL [12] implement computation as a derived rule using rewriting, which in turn is a derived rule implemented outside their trusted kernels. As a result, computation is slow in these systems.

To understand why computation is so sluggish in HOL ITPs, it is worth noting that the primitive steps taken for the computation of Example (1) are numerous:


© Oskar Abrahamsson and Magnus O. Myreen;
licensed under Creative Commons License CC-BY 4.0

- At each step, rewriting has to match the subterm that is to be reduced next (according to a call-by-value order) against each pattern it knows (the left-hand side of the definitions of sum, hd, tl, if-then-else and more); when a match is found, it needs to instantiate the equation whose left-hand-side matched, and then reconstruct the surrounding term.
- Computation over natural numbers is far from constant-time, since 5, 9 and 1 are syntactic sugar for numerals built using the constructor-like functions and constants: Bit0, Bit1 and 0. For example, 5 = Bit1 (Bit0 (Bit1 0)). Deriving equations describing the evaluation of simple operations such as + requires rewriting with lemmas such as these:

  Bit1 m + Bit0 n = Bit1 $(m + n)$
  Bit1 m + Bit1 n = Bit0 (Suc $(m + n)$)
  Suc (Bit0 n) = Bit1 n
  Suc (Bit1 n) = Bit0 (Suc n)
  ...

HOL ITPs employ such laborious methods for computation in order to keep their soundness critical kernel as small as possible: the small size and simplicity of the kernel is key to the soundness argument.

This paper is about how we have added a fast function for computation to the Candle HOL ITP[1]. Candle has a different soundness argument that allows it to move away from being simple in order to be trustworthy: Candle has been proved (in HOL4) to be sound w.r.t. a formal semantics of higher-order logic [3].

With this new function for computation, proving equations via computation is cheap. For the sum example:

- The input term is traversed once, and is converted to a datatype better suited for fast computation. In this representation, each occurrence of sum, hd, tl, etc. can be expanded directly without pattern-matching.
- The representation makes use of host-language integers, so 5 + (9 + (1 + 0)) can be computed using three native addition operations.
- Once the computation is complete, the result is converted back to a HOL term and an equation similar to (1) is returned to the user.

Our function for computation works on a first-order, untyped, monomorphic subset of higher-order logic. Our implementation interprets terms of this subset using a call-by-value strategy and host-language (CakeML) features such as arbitrary precision integer arithmetic.

In our experiments, we observe speed gains of several orders of magnitude when comparing Candle's new compute function against established in-logic computation implementations used by other HOL ITPs (Sec. 8).

Contributions

We make the following contributions:

- We implement a fast interpreter for terms as a user-accessible primitive in the Candle kernel. The implementation allows users to supply code equations dictating how user-defined (recursive) functions are to be interpreted.
- The new primitive has been proven correct with respect to the inference rules of higher-order logic, and has been fully integrated into the existing end-to-end soundness proof of the Candle ITP.
- Our compute function is, in our experiments, significantly faster than the equivalent runs of in-logic compute facilities provided by other HOL ITPs.

[1] Kernel functions are analogous to inference rules in HOL implementations.

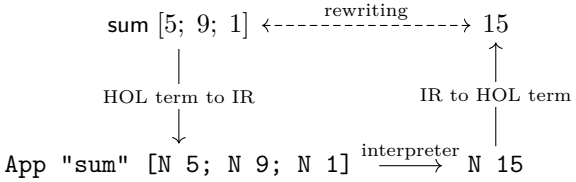


Figure 1 Diagram illustrating the approach we take to embedding logical terms into compute expressions and evaluating them using an interpreter.

Notation: $=$ and $=_c$, $\vdash$ and $\vdash_c$, etc.

This paper contains syntax at multiple, potentially confusing levels. The Candle logic is formalized inside the HOL4 logic. Symbols that exist in both logics are suffixed by a subscript $_c$ in its Candle version; as an example, $=$ denotes equality in the HOL4 logic, and $=_c$ denotes equality in the embedded Candle logic. Likewise, a theorem in HOL4 is prefixed by $\vdash$, while a Candle theorem is prefixed by $\vdash_c$.

Source code and proofs

Our sources are at github.com/CakeML/cakeml/tree/master/candle/prover/compute, and the Candle project is hosted at cakeml.org/candle.

2 Approach

This section explains, at a high level, the approach we have taken to add a new function for computation to Candle.

First, we introduce a new computation friendly internal representation (IR) for expressions that we want to do computation on. On entry to the new compute primitive, the given input term is translated into this new IR. This step corresponds to the downwards arrow in Figure 1. We use an IR that is separate from the syntax of HOL (theorems, terms and types), since the datatypes used by HOL ITPs are badly suited for efficient computation.

We perform computation on the terms of our IR via interpretation. This step is the solid right arrow in Figure 1. On termination, this interpretation arrives at a return value, which is translated to a HOL term r. This step is the up arrow in Figure 1. The new compute primitive returns, to the user, a theorem stating that the input term is equal to the result of computation r. The theorem states that an equality between the points connected with a dashed arrow in Figure 1.

The new compute primitive is a user-accessible function in the Candle kernel and must therefore be proved to be sound, i.e., every theorem it returns must follow by the primitive inference rules of higher-order logic (HOL).

We prove the soundness of our computation function by showing that there is some way of using the inference rules of HOL to mimic the operations of the interpreter. Our use of the inference rules amounts to showing that there is some proof by rewriting that establishes the desired equation. Since Candle performs no proof recording of any kind, it suffices, for the soundness proof, to prove (in HOL4) that there exists some derivation in the Candle logic.

The connection established by the existentially quantified proof is illustrated by the dashed arrow in Figure 1. All reasoning about the interpreter (the lower horizontal arrow) must be wrt. the view of the interpreter provided by the translations to and from the IR (the vertical arrows). Nearly all of our theorems are stated in terms of the arrow upwards, i.e. from IR to HOL.

2.1 Overview

The development of our new compute primitive for Candle was staged into increasingly complex versions.

1. Version 1 (Sec. 3) was a proof-of-concept Candle function for computing the result of additions of concrete natural numbers. This function was implemented as a conversion[2] in the Candle kernel that given a term $m +_c n$ computes the result of the addition r, and returns a theorem $\vdash_c m +_c n =_c r$ to the user. Internally, the implementation makes use of the arbitrary precision integer arithmetic of the host language, i.e. CakeML. The purpose of Version 1 was to establish the concepts needed for this work rather than producing something that is actually useful from a user's point of view.
2. Version 2 (Sec. 4) improved on Version 1 by replacing the type of natural numbers by a datatype for binary trees with natural numbers at the leaves, and by supporting structured control-flow (if-then-else), projections (fst, snd) and the usual arithmetic operations. This version supports nesting of expressions.
3. Version 3 (Sec. 5) extended Version 2 with support for user-supplied code equations for user-defined constants. The code equations are allowed to be recursive and thus the interpreter had to support recursion. This extension also brought with it variables: from Version 3 and on, all interpreters are able to interpret input terms containing variables.
4. Version 4 (Sec. 6) replaced the naive interpreter with one that is designed to evaluate with less overhead. This version uses $O(1)$ operations to look up to code equations and uses environments rather than substitutions for variable bindings. This is the version we perform benchmarks on (Sec. 8).
5. The final Version 5 (Sec. 7) is, at the time of writing, left as future work. In Version 5, our intention is to split the compute function into stages so that users can initialize and feed in code equations separately from calls to the main compute function. This should make repeated calls to the compute facility faster.

At the time of writing, Version 4 (Sec. 6) is integrated into the existing Candle implementation and end-to-end soundness proof.

3 Addition of Natural Numbers (Version 1)

In this section, we describe how we implemented and verified a function for computing addition on natural numbers in the Candle kernel. This is the first step towards a proven-correct function for computation. The approach can be reused to produce computation functions for other kinds of binary operations (multiplication, subtraction, division, etc.) on natural numbers, and it can be used to build evaluators for arithmetic inside more general expressions (Sec. 4).

3.1 Input and output

In Version 1, the user can input terms such as $3 +_c 5$ or $100 +_c 0$, i.e., terms consisting of one addition applied to two concrete numbers. The numbers are shown here as 3, 5, 100, 0, even though they are actually terms in a binary representation based on the constant 0_c, and the functions $\mathsf{Bit0}_c$ and $\mathsf{Bit1}_c$ in the Candle logic.

[2] A *conversion* is a proof procedure that takes a term t as input and proves a theorem $\vdash t = t'$ for some interesting t'.

The output is a theorem equating the input with a concrete natural number. For the examples above, the function returns the following equations. The subscript $_c$ is used below to highlight that these are theorems in the Candle logic.

$$\vdash_c 3 +_c 5 =_c 8 \quad \text{or} \quad \vdash_c 100 +_c 0 =_c 100$$

The results 8 and 100 are computed using addition outside the logic. The challenge is to show that the same computation can be derived from the equations defining $+_c$ (in Candle) using the primitive inference rules of the Candle logic.

3.2 Key soundness lemma

In order to prove the soundness of Version 1 (required for its inclusion in the Candle kernel), we need to prove the following theorem, which states: if the arithmetic operations are defined as expected (num_thy_ok) in the current Candle theory Γ, then the addition ($+_c$) of the binary representations (mk_num) of two natural numbers m and n is equal ($=_c$) to the binary representation of $(m + n)$, where $+$ is HOL4 addition.

$$\begin{array}{l} \vdash \mathsf{num\_thy\_ok}\ \Gamma\ \Rightarrow \\ \quad \Gamma \vdash_c \mathsf{mk\_num}\ m +_c \mathsf{mk\_num}\ n =_c \\ \qquad \mathsf{mk\_num}\ (m + n) \end{array} \tag{2}$$

To understand the theorem statement above, let us look at the definitions of mk_num and num_thy_ok. The function mk_num converts a HOL4 natural number into the corresponding Candle natural number in binary representation:

$$\begin{array}{l} \mathsf{mk\_num}\ n \stackrel{\mathsf{def}}{=} \\ \quad \mathsf{if}\ n = 0\ \mathsf{then}\ 0_c \\ \quad \mathsf{else\ if}\ \mathsf{even}\ n\ \mathsf{then}\ \mathsf{Bit0}_c\ (\mathsf{mk\_num}\ (n\ \mathsf{div}\ 2)) \\ \quad \mathsf{else}\ \mathsf{Bit1}_c\ (\mathsf{mk\_num}\ (n\ \mathsf{div}\ 2)) \end{array}$$

The definition of num_thy_ok asserts that various characterizing equations hold for the Candle constants $+_c$, $\mathsf{Bit0}_c$ and $\mathsf{Bit1}_c$ (the complete definition is not shown below). Here m and n are natural number typed variables in Candle's logic:

$$\begin{array}{l} \mathsf{num\_thy\_ok}\ \Gamma \stackrel{\mathsf{def}}{=} \\ \quad \Gamma \vdash_c 0_c +_c \mathsf{n} =_c \mathsf{n} \wedge \\ \quad \Gamma \vdash_c \mathsf{Suc}_c\ \mathsf{m} +_c \mathsf{n} =_c \mathsf{Suc}_c\ (\mathsf{m} +_c \mathsf{n}) \wedge \\ \quad \Gamma \vdash_c \mathsf{Bit0}_c\ \mathsf{n} =_c \mathsf{n} +_c \mathsf{n} \wedge \\ \quad \Gamma \vdash_c \mathsf{Bit1}_c\ \mathsf{n} =_c \mathsf{Suc}_c\ (\mathsf{n} +_c \mathsf{n}) \wedge \ldots \end{array}$$

We use num_thy_ok as an assumption in Theorem (2), since the computation function is part of the Candle kernel, which does not include these definitions when the prover starts from its initial state (and thus the user might define them differently).

A closer look at num_thy_ok reveals that $+_c$ is characterized by its simple Suc-based equations and $\mathsf{Bit1}_c$ is characterized in terms of Suc and $+_c$. As a result, a direct proof of Theorem (2) would be awkward at best.

To keep the proof of Theorem (2) as neat as possible, we defined the expansion of a HOL number into a tower of Suc_c applications to 0_c:

$$\begin{array}{l} \mathsf{mk\_suc}\ n \stackrel{\mathsf{def}}{=} \\ \quad \mathsf{if}\ n = 0\ \mathsf{then}\ 0_c \\ \quad \mathsf{else}\ \mathsf{Suc}_c\ (\mathsf{mk\_suc}\ (n - 1)) \end{array}$$

and split the proof of Theorem (2) into two lemmas. The first lemma is a mk_suc variant of Theorem (2):

$$\begin{array}{l}\vdash \mathsf{num\_thy\_ok}\ \Gamma\ \Rightarrow \\ \quad \Gamma \vdash_{\mathsf{c}} \mathsf{mk\_suc}\ m +_{\mathsf{c}} \mathsf{mk\_suc}\ n =_{\mathsf{c}} \\ \qquad \mathsf{mk\_suc}\ (m+n)\end{array} \tag{3}$$

and the second lemma $=_{\mathsf{c}}$-equates mk_num with mk_suc:

$$\begin{array}{l}\vdash \mathsf{num\_thy\_ok}\ \Gamma\ \Rightarrow \\ \quad \Gamma \vdash_{\mathsf{c}} \mathsf{mk\_num}\ n =_{\mathsf{c}} \mathsf{mk\_suc}\ n\end{array} \tag{4}$$

The proof of Theorem (3) was done by induction on m, and involved manually constructing the $\vdash_{\mathsf{c}}$-derivation that connects the two sides of $=_{\mathsf{c}}$ in Theorem (3). The proof of Theorem (4) is a complete induction on n and uses Theorem (3) when $+_{\mathsf{c}}$ is encountered. Finally, the proof of Theorem (2) is a manually constructed $\vdash_{\mathsf{c}}$-derivation that uses Theorems (4) and (3), and symmetry of $=_{\mathsf{c}}$.

3.3 From Candle terms to natural numbers

The development described above is in terms of functions (mk_num, mk_suc) that map HOL4 natural numbers into Candle terms, but the implementation also converts in the opposite direction: on initialization, the computation function converts the given input term into its internal representation (see the leftmost arrow in Figure 1).

We use the following function, dest_num, to extract a natural number from a Candle term. This function traverses terms, and recognizes the function symbols used in Candle's binary representation of natural numbers:

$$\begin{array}{l}\mathsf{dest\_num}\ tm \stackrel{\mathsf{def}}{=} \\ \ \mathsf{case}\ tm\ \mathsf{of} \\ \ |\ 0_{\mathsf{c}}\ \Rightarrow\ \mathsf{Some}\ 0 \\ \ |\ \mathsf{Bit0}_{\mathsf{c}}\ r\ \Rightarrow\ \mathsf{option\_map}\ (\lambda\, n.\ 2\times n)\ (\mathsf{dest\_num}\ r) \\ \ |\ \mathsf{Bit1}_{\mathsf{c}}\ r\ \Rightarrow\ \mathsf{option\_map}\ (\lambda\, n.\ 2\times n+1)\ (\mathsf{dest\_num}\ r) \\ \ |\ \_\ \Rightarrow\ \mathsf{None}\end{array}$$

One should read the application Bit$b$ bs as a natural number in binary with least significant bit b and other bits bs.

The correctness of dest_num is captured by the following theorem, which states that $=_{\mathsf{c}}$ is preserved when moving from Candle terms to natural numbers in HOL4, and back:

$$\begin{array}{l}\vdash \mathsf{num\_thy\_ok}\ \Gamma \wedge \\ \quad \mathsf{dest\_num}\ t = \mathsf{Some}\ t'\ \Rightarrow \\ \qquad \Gamma \vdash_{\mathsf{c}} \mathsf{mk\_num}\ t' =_{\mathsf{c}} t\end{array} \tag{5}$$

Version 1 of the computation function also has a function for taking apart a Candle term with a top-level addition $+_{\mathsf{c}}$:

$$\begin{array}{l}\mathsf{dest\_add}\ tm \stackrel{\mathsf{def}}{=} \\ \ \mathsf{case}\ tm\ \mathsf{of} \\ \ |\ (x +_{\mathsf{c}} y)\ \Rightarrow\ \mathsf{Some}\ (x,y) \\ \ |\ \_\ \Rightarrow\ \mathsf{None}\end{array}$$

Equipped with the functions dest_num and dest_add, and Theorems (2) and (5), it is easy to prove the following soundness result. This theorem states: if a term t can be taken apart using dest_add and dest_num, then the term constructed by mk_num and the HOL4 addition, $+$, can be used as the right-hand side of an equation that is $\vdash_c$-derivable.

$$\begin{array}{l} \vdash \mathsf{num\_thy\_ok}\ \Gamma\ \Rightarrow \\ \quad \mathsf{dest\_add}\ t = \mathsf{Some}\ (x,y)\ \wedge \\ \quad \mathsf{dest\_num}\ x = \mathsf{Some}\ m\ \wedge \\ \quad \mathsf{dest\_num}\ y = \mathsf{Some}\ n\ \Rightarrow \\ \quad \Gamma \vdash_c t =_c \mathsf{mk\_num}\ (m+n) \end{array} \tag{6}$$

This theorem can be used as the blueprint for an implementation that uses dest_add, dest_num and mk_num.

3.4 Checking num_thy_ok

Note that Theorem (6) assumes num_thy_ok, which requires certain equations to be true in the current theory Γ. To be sound, an implementation of our computation function must check that this assumption holds.

We deal with this issue in a pragmatic manner, by requiring that the user provides a list of theorems corresponding to the equations of num_thy_ok on each invocation of our computation function. This approach makes num_thy_ok easy to establish, but causes extra overhead on each call to the computation function. Subsequent versions will remove this overhead (Sec. 7).

3.5 Soundness of CakeML implementation

Throughout this section, we have treated functions in the logic of HOL4 as if they were the implementation of the Candle kernel. We do this because the actual CakeML implementation of the Candle kernel is automatically synthesized from these functions in the HOL4 logic, using the tool described in prior work [2].

Updating the entire Candle soundness proof for the addition of Version 1 of the compute function was straightforward, once Theorem (6) was proved and the code for checking num_thy_ok was verified.

4 Compute Expressions (Version 2)

This section describes Version 2, which generalizes the very limited Version 1. While Version 1 only computed addition of natural numbers, Version 2 can compute the value of any term that fits in a subset of Candle terms that we call *compute expressions*. Compute expressions operate over a Lisp-inspired datatype which we call *compute values*; in Candle, this type is called cval.

Even though this second version might at first seem significantly more complicated than the first, it is merely a further development of Version 1. The approach is the same: the soundness theorems we prove are very similar looking. Technically, the most significant change is the introduction of a datatype, cexp, that is the internal representation of all valid input terms, i.e., compute expressions.

4.1 Compute values

To the Candle user, the following cval datatype is important, since all terms supplied to the new compute function must be of this type. The cval datatype is a Lisp-inspired binary tree with natural numbers (num) at the leaves:

```
cval = Pairc cval cval
     | Numc num
```

4.2 Compute expressions

The other important datatype is cexp, which is the internal representation that user input is translated into:

```
cexp = Pair cexp cexp
     | Num num
     | If cexp cexp cexp
     | Uop uop cexp
     | Binop binop cexp cexp

uop = Fst | Snd | IsPair

binop = Add | Sub | Mul | Div | Mod | Less | Eq
```

The cexp datatype is extended with more constructors in Version 3, described in Section 5.

4.3 Input terms

On start up, the compute function maps the given term into the cexp type. For example, given this term as input:

```
cifc (Numc 1) (Numc 2)
   (fstc (Pairc (Numc 3) (Numc 4)))
```

the function will create this cexp expression:

```
If (Num 1) (Num 2) (Uop Fst (Pair (Num 3) (Num 4)))
```

This mapping assumes that certain functions in the Candle logic (e.g. $\mathsf{fst_c}$) correspond to certain constructs in the cexp datatype (e.g. Uop Fst). Note that there is nothing strange about this: in Version 1, we assumed that $+_c$ corresponds to addition. We formalize the assumptions about $\mathsf{fst_c}$, etc., next.

4.4 Context assumption: cexp_thy_ok

Just as in Version 1, Version 2 also has an assumption on the current theory context. In Version 1, the assumption num_thy_ok ensured that the Candle definition of $+_c$ satisfied the relevant characterizing equations. For Version 2, this assumption was extended to cover characterizing equations for all names that the conversion from user input to cexp recognizes: $\mathsf{cif_c}$, $\mathsf{fst_c}$, etc. These characterizing equations fix a semantics for the Candle functions that correspond to constructs of the cexp type. For simplicity, all of the Candle functions take inputs of type cval and produce outputs of type cval.

Our implementation makes no attempt at ensuring that functions are applied to sensible inputs. Consequently, it is perfectly possible to write strange terms in this syntax, such as $\mathsf{fst_c}$ (Numc 3), or $\mathsf{add_c}$ (Numc 3) (Pairc p q). We resolve such cases in a systematic way:

- Operations that expect numbers as input treat $\mathsf{Pair_c}$ values as $\mathsf{Num_c}\ 0$.
- Operations that expect a pair as input return $\mathsf{Num_c}\ 0$ when applied to $\mathsf{Num_c}$ values.

This treatment of the primitives can be seen in the assumption, called cexp_thy_ok, that we make about the context for Version 2. Below, x and y are variables in the Candle logic with type cval. The lines specifying $\mathsf{add_c}$ are:

$$\begin{aligned}&\mathsf{cexp\_thy\_ok}\ \Gamma \stackrel{\text{def}}{=}\\ &\ldots \wedge\\ &\Gamma \vdash_c \mathsf{add_c}\ (\mathsf{Num_c}\ m)\ (\mathsf{Num_c}\ n) =_c \mathsf{Num_c}\ (m +_c n) \wedge\\ &\Gamma \vdash_c \mathsf{add_c}\ (\mathsf{Pair_c}\ x\ y)\ (\mathsf{Num_c}\ n) =_c \mathsf{Num_c}\ n \wedge\\ &\Gamma \vdash_c \mathsf{add_c}\ (\mathsf{Num_c}\ m)\ (\mathsf{Pair_c}\ x\ y) =_c \mathsf{Num_c}\ m \wedge \ldots\end{aligned}$$

The lines specifying $\mathsf{fst_c}$ are:

$$\begin{aligned}&\Gamma \vdash_c \mathsf{fst_c}\ (\mathsf{Pair_c}\ x\ y) =_c x \wedge\\ &\Gamma \vdash_c \mathsf{fst_c}\ (\mathsf{Num_c}\ n) =_c \mathsf{Num_c}\ 0_c \wedge \ldots\end{aligned}$$

The following characteristic equations for $\mathsf{cif_c}$ illustrate that we treat $\mathsf{Num_c}\ 0_c$ as false and all other values as true:

$$\begin{aligned}&\Gamma \vdash_c \mathsf{cif_c}\ (\mathsf{Num_c}\ 0_c)\ x\ y =_c y \wedge\\ &\Gamma \vdash_c \mathsf{cif_c}\ (\mathsf{Num_c}\ (\mathsf{Suc}\ n))\ x\ y =_c x \wedge\\ &\Gamma \vdash_c \mathsf{cif_c}\ (\mathsf{Pair_c}\ x'\ y')\ x\ y =_c x \wedge \ldots\end{aligned}$$

Comparison primitives return $\mathsf{Num_c}\ 1$ for true.

4.5 Soundness

The following theorem summarizes the operations and soundness of Version 2. If a term t can be successfully converted (using dest_term) into a compute expression *cexp*, then t is equal to a Candle term created (using mk_term) from the result of evaluating *cexp* using a straightforward evaluation function (cexp_eval):

$$\begin{aligned}&\vdash \mathsf{cexp\_thy\_ok}\ \Gamma\ \Rightarrow\\ &\quad \mathsf{dest\_term}\ t = \mathsf{Some}\ \mathit{cexp}\ \Rightarrow\\ &\quad \Gamma \vdash_c t =_c \mathsf{mk\_term}\ (\mathsf{cexp\_eval}\ \mathit{cexp})\end{aligned} \tag{7}$$

Note the similarity between Theorems (6) and (7). Where Theorem (6) uses +, Theorem (7) calls cexp_eval. The evaluation function cexp_eval is defined to traverse the cexp bottom-up in the most obvious manner, respecting the evaluation rules set by the characterizing equations of cexp_thy_ok.

4.6 CakeML code and integration

The functions dest_term, cexp_eval and mk_term are the main workhorses of the implementation of Version 2. Corresponding CakeML implementations are synthesized from these functions. The definition of the evaluator function cexp_eval uses arithmetic operations ($+$, $-$, $\times$, div, mod, $<$, $=$) over the natural numbers. Such arithmetic operations translate into arbitrary precision arithmetic operations in CakeML.

Updating the Candle proofs for Version 2 was a straightforward exercise, given the prior integration of Version 1.

5 Recursion and user-supplied code equations (Version 3)

Version 3 of our compute function for Candle adds support for (mutually) recursive user-defined functions. The user supplies function definitions in the form of *code equations*.

5.1 Code equations

In our setting, a code equation for a user-defined constant c is a Candle theorem of the form:

$$\vdash_{\mathsf{c}} \mathsf{c}\ v_1\ \ldots\ v_n = e$$

where each variable v_i has type cval and the expression e has type cval. Furthermore, the free variables of e must be a subset of $\{v_1, \ldots, v_n\}$. Note that any user-defined constants, including c, are allowed to appear in e in fully applied form. Every user-defined constant appearing in some right-hand side e must have a code equation describing that constant.

5.2 Updated compute expressions

We updated the cexp datatype to allow variables (Var), applications of user-supplied constants (App), and, at the same time, we added let-expressions (Let):

```
cexp = Pair cexp cexp
     | Num num
     | Var string
     | App string (cexp list)
     | Let string cexp cexp
     | If cexp cexp cexp
     | Uop uop cexp
     | Binop binop cexp cexp
```

Variables are present to capture the values bound by the left-hand sides of code equations and by let-expressions.

The interpreter for Version 3 of our compute function uses a substitution-based semantics, and keeps track of code equations as a simple list. This style of semantics maps well to the Candle logic's substitution primitive, thus simplifying verification, but at a price:

- At each let-expression or function application, the entire body of the let-expression or the code equation corresponding to the function may be traversed an additional time, to substitute out variables.
- At each function application, the code equation corresponding to the function name is found using linear search, making the interpreter's performance degrade as more code equations are added.

We address these shortcomings in Version 4 of our compute function, in Section 6.

5.3 Soundness

The following theorem is the essential part of the soundness argument for Version 3. The user supplies the Version 3 compute function with: a list of theorems that allows it to establish cexp_thy_ok, a list *eqs* of code equations, and a term t to evaluate. Every theorem in *eqs* must be a Candle theorem ($\vdash_{\mathsf{c}}$). Definitions *defs* are extracted from the given code equations *eqs*. A compute expression *cexp* is extracted from the given input term w.r.t. the available definitions *defs*. An interpreter, interpret, is run on the *cexp*, and its execution

is parameterized by *defs* and a clock which is initialized to a large number init_ck. If the interpreter returns a result *res*, i.e. Some *res*, then an equation between the input term t and mk_term *res* can be returned to the user.

$$\begin{array}{l}\vdash \mathsf{cexp\_thy\_ok}\ \Gamma\ \Rightarrow \\ \quad (\forall eq.\ \ \mathsf{mem}\ eq\ eqs\ \Rightarrow\ \Gamma \vdash_{\mathsf{c}} eq) \wedge \\ \quad \mathsf{dest\_eqs}\ eqs = \mathsf{Some}\ defs \wedge \\ \quad \mathsf{dest\_tm}\ defs\ t = \mathsf{Some}\ cexp \wedge \\ \quad \mathsf{interpret\ init\_ck}\ defs\ cexp = \mathsf{Some}\ res\ \Rightarrow \\ \quad \Gamma \vdash_{\mathsf{c}} t =_{\mathsf{c}} \mathsf{mk\_term}\ res\end{array} \tag{8}$$

There are a few subtleties hidden in this theorem that we will comment on next.

First, the statement of Theorem 8 includes an assumption that the user-provided code equations *eqs* are theorems in the context Γ. The user is not in any way obliged to prove this: the fact that they can supply the compute primitive with a list of theorems means that they are valid in Candle's context at that point. Candle's soundness result allows us to discharge this assumption where Theorem 8 is used.

Second, the functions dest_eqs and dest_term perform sanity checks on their inputs. For example, dest_eqs checks that all right-hand sides in the equations *eqs* mention only constants for which there are code equations in *eqs*.

Third, the interpret function, which is used for the actual computation, takes a clock (sometimes called fuel parameter) in order to guarantee termination. This clock is not strictly necessary, but made it easier to use the existing CakeML code synthesis tools. The clock is decremented by interpret on each function application (i.e. App), and, due to the substitution semantics, also on each Let. If the clock is exhausted, interpret returns None.

5.4 CakeML code

As with previous versions, the CakeML implementation of the computation function is synthesized from the HOL4 functions. For efficiency purposes, the generated CakeML code for interpret avoids returning an option and instead signals running out of clock using an ML exception. We note that it is very unlikely that a user has the patience to wait for a timeout since the value of init_ck is very large (maximum smallnum).

5.5 Integration

Updating the Candle proofs for Version 3 required more work than Versions 1 and 2, since we had to verify the correctness of the sanity checks performed on the user-provided list of code equations.

6 Efficient interpreter (Version 4)

For Version 4, we replaced the interpreter function, interpret, with compilation to a different datatype for which we have a faster interpreter.

The new datatype for representing programs is called ce, shown below. It uses de Bruijn indexing for local variables, and represents function names as indices into a vector of function bodies, which means lookups happen in constant time during interpretation. Rather than representing primitive functions by names, the ce datatype represents primitive functions as (shallowly embedded) function values that can immediately be applied to the result of

evaluating the argument expressions.

```
ce = Const num
   | Var num
   | Let ce ce
   | If ce ce ce
   | Monop (cval → cval) ce
   | Binop (cval → cval → cval) ce ce
   | App num (ce list)
```

The new faster interpreter exec, shown in Figure 2, for the ce datatype addresses the two main shortcomings of Version 3. First, it drops the substitution semantics in favor of de Bruijn variables and an explicit environment, so that variable substitution can be deferred until (and if) the value bound to a variable is needed. Second, all function names are replaced by an index into a vector which stores all user-provided code equations.

When updating Version 3 to Version 4, we simply replaced the following line in the implementation:

```
interpret init_ck defs cexp
```

with the line below, which calls the compilers compile_all and compile (these translate cexp into ce, turning variables and function names into indices) and then runs exec, which interprets the program represented in terms of ce:

```
exec init_ck [] (compile_all defs) (compile defs [] cexp)
```

Updating the proofs for Version 4 was a routine exercise in proving the correctness of the compilers compile_all and compile. In this proof, compiler correctness is an equality: the new line computes exactly the same result as the line that it replaced (under some assumptions that are easily established in the surrounding proof). The adjustments required in the existing proofs were minimal.

7 Staged set up (Version 5)

At the time of writing, Version 5 is not yet implemented. However, the plan is to reduce the overhead of calling the compute function.

In Versions 1–4, the characteristic equations need to be checked (i.e., establishing cexp_thy_ok) and the user-supplied code equations must be compiled, on each call to the compute primitive. In these versions, even a simple evaluation, such as $1 + 2$, will make the compute function check all of the characteristic equations, every time.

For Version 5, the plan is to curry the arguments of the compute primitive and arrange the implementation to: perform the characteristic equations checks after the first argument is given and then return a function that performs the rest of the computation given the remaining arguments. Note that the returned function can only exist if all of the characteristic equation checks have passed. The verification of the Candle prover has not yet dealt with any such conditionally existing function values. We expect these values will need special treatment in the Candle prover's soundness theorem.

```
exec funs env ck (Const n) def=
  return (Num n)
exec funs env ck (Var n) def=
  return (env_lookup n env)
exec funs env ck (Monop m x) def=
  do
    v ← exec funs env ck x;
    return (m v)
  od
exec funs env ck (Binop b x y) def=
  do
    v ← exec funs env ck x;
    w ← exec funs env ck y;
    return (b v w)
  od
exec funs env ck (App f xs) def=
  do
    check_clock ck;
    vs ← execl funs env ck xs [];
    c ← get_code f funs;
    exec funs vs (ck − 1) c
  od
exec funs env ck (Let x y) def=
  do
    check_clock ck;
    v ← exec funs env ck x;
    exec funs (v::env) (ck − 1) y
  od
exec funs env ck (If x y z) def=
  do
    v ← exec funs env ck x;
    exec funs env ck
      (if v = Num 0 then z else y)
  od
execl funs env ck [] acc def=
  return acc
execl funs env ck (x::xs) acc def=
  do
    v ← exec funs env ck x;
    execl funs env ck xs (v::acc)
  od
```

Figure 2 Definition of the fast interpreter as functions in HOL.

```
fun exec funs env ck e =
  case e of
    Const n => Num n
  | Var n => List.nth n env
  | Monop m x =>
      let
        val v = exec funs env ck x
      in
        m v
      end
  | Binop b x y =>
      let
        val v = exec funs env ck x
        val w = exec funs env ck y
      in
        b v w
      end
  | App f xs =>
      let
        val _ = check_clock ck
        val vs = execl funs env ck xs []
        val c = Vector.nth f funs
      in
        exec funs vs (ck - 1) c
      end
  | Let x y =>
      let
        val _ = check_clock ck
        val v = exec funs env ck x
      in
        exec funs (v::env) (ck - 1) y
      end
  | If x y z =>
      let
        val v = exec funs env ck x
      in
        exec funs env ck
          (if v = Num 0 then z else y)
      end
and execl funs env ck l acc =
  case l of
    [] => acc
  | (x::xs) =>
      let
        val v = exec funs ck x
      in
        execl funs env ck xs (v::acc)
      end
```

Figure 3 CakeML code generated from definition of exec.

Table 1 Running times for Candle's compute primitive, HOL4's EVAL, HOL Light's EVAL, and Isabelle/HOL's in-logic `Code_Simp.dynamic_conv`. Below dash, —, indicates *not measured*.

fact n for different values of n

n	Candle	HOL4	H.Light	Isabelle
256	<1 ms	2.3 s	0.6 s	14 s
512	<1 ms	4.1 s	3.5 s	202 s
1024	<1 ms	127 s	17.6 s	2451 s
2048	11 ms	684 s	86.1 s	—
32768	0.9 s	—	—	—

primes_upto n for different values of n

n	Candle	HOL4	H.Light	Isabelle
256	<1 ms	0.5 s	1.3 s	2.6 s
512	<1 ms	1.6 s	5.2 s	9.8 s
1024	2 ms	6.3 s	20.7 s	35.6 s
2048	9 ms	24.2 s	83.4 s	132 s
32768	1.7 s	—	—	—

rev_enum n for different values of n

n	Candle	HOL4	H.Light	Isabelle
256	0.02 s	1.1 s	66.2 s	10.2 s
512	0.03 s	2.3 s	251 s	37.1 s
1024	0.07 s	4.7 s	1005 s	172 s
2048	0.1 s	9.5 s	4203 s	791 s
32768	2.5 s	—	—	—

n steps of Conway's Game of Life

n	Candle	HOL4	H.Light	Isabelle
1	0.03 s	0.6 s	14.9 s	1.5 s
10	0.08 s	5.3 s	147 s	15.0 s
100	0.8 s	54 s	1474 s	148 s
1000	8.0 s	568 s	14623 s	1466 s
10000	79 s	—	—	—

8 Evaluation

In this section, we report on experiments comparing our new compute function to the in-logic interpreters of HOL4, HOL Light, and Isabelle/HOL. We tested the performance of each on the following four example programs written as function in the logic of HOL.

- the factorial function,
- enumeration of primes,
- generating and reversing a list of numbers,
- simulation of a 100-by-100 grid of cells in Conway's Game of Life.

The tests were run on an Intel i7-7700K 4.2GHz with 64 GiB RAM running Ubuntu 20.04. The code used for these experiments is available at cakeml.org/candle_benchmarks.html.

The results, in Figure 1, show that Candle's new compute function runs orders of magnitude faster than the derived rules of HOL4, HOL Light, and Isabelle/HOL, on all four examples. In fact, it was difficult to choose input sizes large enough for us to gather meaningful measurements from our computation function, while keeping the runtimes of its derived counterparts within minutes. For this reason, we added one large data point to the end of each experiment. In Figure 1, a dash, —, indicates that we did not test this.

The first two examples, factorial and primes, demonstrate the speed of computing basic arithmetic, while the latter two examples, list reversal and Conway's Game of Life, highlight that Candle's compute primitive is also well suited for structural computations, such as tree traversals, that do not involve much arithmetic.

Factorial

The first example is a standard, non-tail-recursive factorial function, tested on inputs of various sizes. The results of the tests are shown in the upper left corner of Table 1. This is the only test where HOL Light out performs HOL4. We suspect HOL Light benefits from the effort that has gone into making basic arithmetic evaluate fast in HOL Light.

Prime enumeration

The second example, primes_upto, enumerates all primes up to n and returns them as a list. We chose to implement the checks for primality using trial division, since it is challenging to compute division and remainder efficiently inside the logic. The results of the tests are shown in the upper right corner of Table 1.

List reversal

The third example performs repeated list reversals. The function rev_enum creates a list of the natural numbers $[1, 2, \ldots, n]$ and then calls a tail-recursive list reverse function rev on this list 1000 times. The results of the tests are shown in the lower left corner of Table 1. On this and the next benchmark HOL Light performs much worse than HOL4 and Isabelle/HOL.

Conway's Game of Life

The fourth example simulates a 100-by-100 grid of cells in Conway's Game of Life. The surface of this 100-by-100 square is set up to have a set up that consists of five Gosper glider generators that interact. The set up is self contained, i.e., it never touches the boundaries of the 100-by-100 grid. The simulation runs for n steps of Conway's Game of Life. The results of the tests are shown in the lower right corner of Table 1.

9 Related Work

This section discusses related work in the area of computation in interactive theorem provers.

9.1 HOL4

Barras implemented a fast interpreter for terms in HOL4 [5], usually referred to as Eval. Eval implements an extended version of Crégut's abstract machine KN [6], and performs strong reduction of open terms, and supports user-defined datatypes and pattern-matching, and rewriting using user-supplied conversions. It is this Eval function that was used when benchmarking HOL4 in Section 8.

Unlike our work, Eval operates directly on HOL terms. The HOL4 kernel was modified by Barras to make this as efficient as possible: the HOL4 kernel uses de Bruijn terms and explicit substitutions to ensure that Eval runs fast. However, true to LCF tradition, all interpreter steps are implemented using basic kernel inferences.

9.2 HOL Light

A HOL Light port of Eval exists [14] and was used in Section 8. However, unlike HOL4, the HOL Light kernel has not been optimized for running Eval; HOL Light uses name-carrying terms without explicit substitutions, making this port comparably slow.

9.3 Isabelle/HOL

Isabelle/HOL supports two mechanisms for efficient evaluation, both due to Haftmann and Nipkow. A code generation feature [9, 10] can be used to synthesize ML, Haskell and Scala programs from closed terms, which can then be compiled and executed efficiently. We borrow the concept of code equations (Sec. 5) from their work, but note that Isabelle's code equations are more general than ours.

The second option is based on normalization-by-evaluation (NBE) mechanism [4] and synthesizes ad-hoc ML interpreters over an untyped lambda calculus datatype from (possibly open) HOL terms. The ML code is compiled and executed by an ML compiler, and the resulting values are reinterpreted as HOL terms.

Both methods support a rich, higher-order, computable fragment of HOL. However, both also escape the logic, make use of unverified functions for synthesizing functional programs, and rely on unverified compilers and language runtimes for execution.

9.4 Dependent type theories

Computation is an integral part of ITPs based on higher-order type theories, such as Coq [15], and Lean [7]. Their logics identify terms up to normal form and must reduce terms as part of their proof checking (i.e., type checking). Consequently, their trusted kernels must implement an interpreter or compiler of some sort.

Coq supports proof by computation using its interpreter (accessible via vm_compute), as well as native code generation to OCaml (accessible via native_compute). Internally, Coq's interpreter implements an extended version of the ZAM machine used in the interactive mode of the OCaml compiler [8], but with added support for open terms.

A formalization of the abstract machine used in the interpreter exists [8], but the actual Coq implementation is completely unverified.

9.5 First-order logic

ACL2 is an ITP for a quantifier-free first-order logic with recursive, untyped functions. It axiomatizes a purely functional fragment of Common Lisp, which doubles as term syntax and host language for the system. As a consequence, some terms can be compiled and executed at native speed. However, this execution speed comes at a cost: no verified Lisp compiler exists that can host ACL2, its soundness critical code encompasses essentially the entire theorem prover.

10 Conclusion

We have added a new verified function for computation to the Candle ITP. The new computation function was developed in stages through different versions. For each version, we proved that the new function only produces theorems that follow by the inference rules of HOL. In our experiments, Candle's new computation functionality produced performance numbers that are several orders of magnitude faster than in-logic evaluation mechanisms provided by mainstream HOL ITPs.

Our new compute function requires all functions that it uses to be first-order functions that perform all computations using a Lisp-inspired datatype for compute values (cval). We leave it to future work to relax this requirement.

At present, the performance numbers suggest that we do not need to go to the trouble of replacing our interpreter-based solution with a solution that compiles the given input to native machine code for extra performance. However, future case studies might lead us to explore such options too.

We envision that future case studies might explore how facilities for fast in-logic computation might open the door to for verified decision procedures (for linear arithmetic, linear algebra, or word problems) in HOL provers. Such proof procedures have typically been programmed in the meta language (SML and OCaml) of HOL provers.

References

1 Oskar Abrahamsson. *A Verified Theorem Prover for Higher-Order Logic*. PhD thesis, Chalmers University of Technology, 2022.

2 Oskar Abrahamsson, Son Ho, Hrutvik Kanabar, Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Yong Kiam Tan. Proof-producing synthesis of CakeML from monadic HOL functions. *Journal of Automated Reasoning (JAR)*, 2020. URL: https://rdcu.be/b4FrU.

3 Oskar Abrahamsson, Magnus O. Myreen, Ramana Kumar, and Thomas Sewell. Candle: A verified implementation of HOL Light. In June Andronick and Leonardo de Moura, editors, *Interactive Theorem Proving (ITP)*, volume 237 of *LIPIcs*, pages 3:1–3:17, 2022. doi:10.4230/LIPIcs.ITP.2022.3.

4 Klaus Aehlig, Florian Haftmann, and Tobias Nipkow. A compiled implementation of normalisation by evaluation. *J. Funct. Program.*, 22(1):9–30, 2012. doi:10.1017/S0956796812000019.

5 Bruno Barras. Programming and computing in HOL. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 1869 of *Lecture Notes in Computer Science*, pages 17–37. Springer, 2000. doi:10.1007/3-540-44659-1_2.

6 Pierre Crégut. An abstract machine for lambda-terms normalization. In Gilles Kahn, editor, *Conference on LISP and Functional Programming (LFP)*, pages 333–340. ACM, 1990. doi:10.1145/91556.91681.

7 Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *International Conference on Automated Deduction (CADE)*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. doi:10.1007/978-3-030-79876-5_37.

8 Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In Mitchell Wand and Simon L. Peyton Jones, editors, *International Conference on Functional Programming (ICFP)*, pages 235–246. ACM, 2002. doi:10.1145/581478.581501.

9 Florian Haftmann. *Code generation from specifications in higher-order logic*. PhD thesis, Technical University Munich, 2009.

10 Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Functional and Logic Programming (FLOPS)*, volume 6009 of *Lecture Notes in Computer Science*, pages 103–117. Springer, 2010. doi:10.1007/978-3-642-12251-4_9.

11 John Harrison. HOL Light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66. Springer, 2009. doi:10.1007/978-3-642-03359-9_4.

12 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.

13 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 (TPHOLs)*, volume 5170 of *LNCS*. Springer, 2008. doi:10.1007/978-3-540-71067-7_6.

14 Alexey Solovyev. HOL Light's computelib. Accessed 2022-06-11. https://github.com/jrh13/hol-light/blob/master/compute.ml.

15 The Coq Development Team. The Coq reference manual. Accessed 2022-06-11. https://coq.inria.fr/distrib/current/refman/.

Formalizing Functions as Processes

Beniamino Accattoli
Inria & LIX, École Poytechnique, Palaiseau, France

Horace Blanc
Independent researcher, Paris, France

Claudio Sacerdoti Coen
Alma Mater Studiorum - University of Bologna, Italy

Abstract

We present the first formalization of Milner's classic translation of the λ-calculus into the π-calculus. It is a challenging result with respect to variables, names, and binders, as it requires one to relate variables and binders of the λ-calculus with names and binders in the π-calculus. We formalize it in Abella, merging the set of variables and the set of names, thus circumventing the challenge and obtaining a neat formalization.

About the translation, we follow Accattoli's factoring of Milner's result via the linear substitution calculus, which is a λ-calculus with explicit substitutions and contextual rewriting rules, mediating between the λ-calculus and the π-calculus. Another aim of the formalization is to investigate to which extent the use of contexts in Accattoli's refinement can be formalized.

2012 ACM Subject Classification Theory of computation → Lambda calculus; Theory of computation → Process calculi; Theory of computation → Operational semantics; Theory of computation → Automated reasoning

Keywords and phrases Lambda calculus, pi calculus, proof assistants, binders, Abella

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.5

Supplementary Material *Software (Abella sources)*: `github.com/sacerdot/FunctionsAsProcesses` archived at `swh:1:dir:3237e6680f66745d679f6976470e9a7bb465c931`

Funding Sacerdoti Coen has been supported by the INdAM-GNCS project "Proprietà qualitative e quantitative dei Sistemi Reversibili" and by the Cost Action CA20111 EuroProofNet.

1 Introduction

Milner's translation of the λ-calculus in the π-calculus [32] is a classic result relating two paradigmatic formalisms. It gave rise to many studies, most notably by Sangiorgi, both alone and with co-authors [39, 40, 18, 42, 27, 20], but also *e.g.* by Boudol [15, 14], Niehren [35], Kobayashi [28], Cai and Fu [16], Toninho et al. [44], and Biernacka et al. [13].

Properties of the π-calculus have been formalized a number of times, for instance by Melham [29], Aït Mohamed [34], Hirschkoff [25], Despeyroux [19], Röckl et al. [38], Honsell et al. [26], Gay [24], Bengston and Parrow [12], Gabbay [21], Chauduri et al. [17], Orchard and Yoshida [36], Perera and Cheney [37], Veltri and Vezzosi [45], and Ambal et al. [10]. To our knowledge, however, the correctness of the translation of λ into π has never been formalized. Miller and Nadathur implement the translation in [30] (p. 274), but not the proof of correctness. In [25], Hirschkoff clearly states that his work is preliminary to the formalization of the correctness of the translation. He also says that one of the main obstacles is the correspondence between term variables and process names, and that *"some work should be done to reformulate some parts of the proof in a way that would be more tractable for the task of mechanisation"*. This was in 1997. In the meantime, the theory of proof assistants has developed a variety of tools for dealing with names and binders, and Milner's translation has indeed been refactored, as we discuss below.


© Beniamino Accattoli, Horace Blanc, and Claudio Sacerdoti Coen;
licensed under Creative Commons License CC-BY 4.0

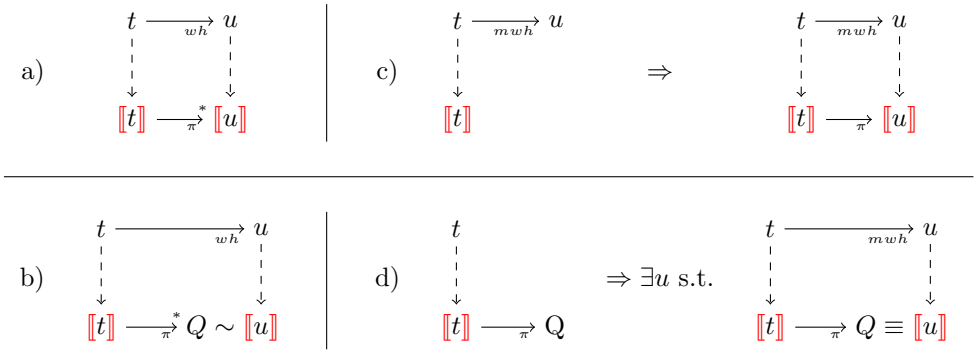


Figure 1 Diagrams describing the relationships between terms and processes.

Variable, Names, and Binders. The translation of λ into π poses additional difficulties with respect to studying a single language with bindings:

1. *Variables/names relation*: one has to relate two languages with binders and establish a relationship between variables and binders in a term t and the names and binders in its representation $[\![t]\!]$ as a process, in order to prove that rewriting steps on both sides preserve the translation;
2. *Not a bijection, and issues with de Bruijn*: the process $[\![t]\!]$ uses more binders than t. Typically, applications (which have no binders) are represented in π via the addition of various binders. Therefore, adopting de Bruijn representations of terms and processes, a variable occurrence of index $i \in \mathbb{N}$ in t is not represented with the same index in $[\![t]\!]$.
3. *Structural equivalence*: reduction on processes is defined only *up to* structural equivalence, which re-organizes the structure of binders by moving restrictions operators around.

Small-Step *vs* Micro-Step. Names are not the only difficulty. The ways in which the λ-calculus and the π-calculus compute are inherently different. A first aspect is that the π-calculus does not compute under prefixes, which corresponds to *weak* evaluation on the λ-calculus, that is, to not compute under abstractions. This is however not the key point.

The λ-calculus rests on a *small-step* operational semantics, based on meta-level substitution, which replaces all occurrences of a variable at the same time with a whole sub-term. The π-calculus instead has a *micro-step* approach: it only substitutes names, not sub-processes, and does the analogous of replacing only one occurrence at a time, keeping sort of *explicit substitutions* for the names that have not been fully substituted yet. In the context of λ-calculi, the relationship between small-step and micro-step evaluation is well studied, and dealt with via the notion of *unfolding* of explicit substitutions, which turns explicit substitutions into meta-level ones. The problem, in the case of the π-calculus, is that the unfolding operation has no natural analogous on processes.

The small-*vs*-micro-step issue implies that the simulation of λ into π, which relates *weak head reduction* $\to_{wh}$ on the λ-calculus with reduction in π, is not as strong as one might expect. The diagram in Fig. 1.a indeed does *not* hold. One only has the diagram in Fig. 1.b, for which $[\![t]\!]$ reduces to a process Q which is *strongly bisimilar* to $[\![u]\!]$, that is, that behaves equivalently *externally* (*i.e.* with respect to the environment), but which is in general very different from $[\![u]\!]$ both structurally and with respect to *internal* reductions.

Refactoring of the Translation. In his seminal work, Milner shows that Q can be seen as the representation of a term $[\![r]\!]$ plus a set of processes corresponding to explicit substitutions $[x_1 \leftarrow [\![w_1]\!]] \dots [x_k \leftarrow [\![w_k]\!]]$, and that $[\![u]\!] = [\![r]\!]\{x_1 \leftarrow [\![w_1]\!]\} \dots \{x_k \leftarrow [\![w_k]\!]\}$, that is, turning the explicit substitutions into meta-level ones and applying them to $[\![r]\!]$ (which is the unfolding of Q, if seen as a λ-term with explicit substitutions) gives $[\![u]\!]$.

In 2013, Accattoli refines Milner's argument factoring the translation of λ into π via a simple λ-calculus with explicit substitutions [2], namely Accattoli and Kesner's *linear substitution calculus* (LSC). The LSC is itself a refinement of a calculus by Milner [33], and can be considered as the canonical micro-step λ-calculus: Accattoli, Kesner, and co-authors have shown that it is strongly related to abstract machines [4, 8], linear logic proof nets [3], reasonable cost models [7], multi types [9], and rewriting theory [5]. Back to the translation, in [2] it is shown that the π-calculus evaluates λ-terms *exactly* as the LSC version $\to_{mwh}$ of the weak head strategy $\to_{wh}$ of the λ-calculus. Once one replaces $\to_{wh}$ with $\to_{mwh}$, indeed, the diagrams in Fig. 1.c and Fig. 1.d hold, that is, there is a *bijection* of steps between $\to_{mwh}$ and π, and the translation becomes a *strong bisimulation* between the *internal reductions* of the two (up to structural equivalence $\equiv$), considerably tightening Milner's result. Intuitively, the problem with the unfolding is avoided altogether, since both the LSC and π are micro-step.

The other half of the simulation, that is, the relationship between the LSC and the λ-calculus was studied independently by Accattoli and Dal Lago [6], as part of a study of time cost models for the λ-calculus. It amounts to relate small-step and micro-step notions of substitutions. The factorization via the LSC recasts such a study within a single formalism, since the LSC is an extension of the λ-calculus, and allows one to present it using the unfolding, which is natural in this setting, and makes the study conceptually cleaner.

This Paper. Here we present a formalization of the translation of λ into π, the first one in the literature. We develop it in the Abella proof assistant [22, 11], the first version of which appeared in 2008. It is based on Miller and Nadathur's λ-*tree syntax* [30], a variant of higher-order abstract syntax providing primitive support for binders and capture-avoiding substitution, and Miller and Tiu's ∇ *(nabla) quantifier* [31, 23], providing primitive support for free variables. The difficulties of the translation related to names are fully circumvented by defining terms and processes on the same type, as to share the set of variables/names. And we follow Accattoli and decompose Milner's result in two parts: first, the relationship between the LSC and π, and then the one between the LSC and λ. The resulting formalization is neat and compact.

Contexts and Distance. The LSC is a framework that relies on *contexts*, that is, terms with a hole, and contextual rewriting rules, also called *at a distance*. Beyond the formalization of the translation, this work also explores how to formalize the kind of context-based reasoning arising in the theory of the LSC.

In [2], Accattoli gives a novel contextual presentation of the π-calculus communication rules, the key property of which is that structural equivalence can be postponed. He uses this fact to simplify the proof of the LSC/π correspondence. We also formalize the equivalence of the new presentation with respect to the ordinary one.

2 λ-Terms, Contexts, Rules, and Processes, with Pen and Paper

Here we give the pen-and-paper definitions of the languages involved in our formalization.

λ-Terms and Contexts. We start by defining λ-terms and contexts.

$$\lambda\text{-TERMS}\quad t,u,r \;::=\; x \mid \lambda x.t \mid tu \qquad \text{CONTEXTS}\quad C,D \;::=\; \langle\cdot\rangle \mid \lambda x.C \mid Cu \mid tC$$

Contexts are terms with exactly one hole $\langle\cdot\rangle$, which is a placeholder for a removed sub-term. The basic operation about contexts is *plugging* $C\langle t\rangle$ of a term t for the hole of C, which amounts to simply replacing the hole with t. The tricky aspect of plugging is that it is *not* a capture-avoiding operation, and that it can capture many variables at once, that is, for instance $(\lambda x.\lambda y.\langle\cdot\rangle)\langle xyz\rangle = \lambda x.\lambda y.xyz$. Capture-avoiding plugging is instead denoted with $C\langle\!\langle t\rangle\!\rangle$. While Abella offers primitive first-class support for both binders and free variables, it does not provide support for general contexts. More precisely, it supports contexts with capture-*avoiding* plugging (*i.e.* $C\langle\!\langle t\rangle\!\rangle$), since it is an instance of capture-avoiding substitution (where the bound variable has only one occurrence). What is not available is the capture-allowing operation of plugging (*i.e.* $C\langle t\rangle$). In our development, luckily, in the only point where we really need contexts, we can limit ourselves to a capture-*avoiding* notion of plugging.

Weak Head Reduction. We shall only use the simplest possible evaluation strategy on λ-terms, *weak head reduction* $\to_{wh}$, also known as (weak) *call-by-name* reduction.

$$\text{WEAK HEAD REDUCTION}\quad \frac{}{(\lambda x.t)u \to_{wh} t\{x\leftarrow u\}} \qquad \frac{t \to_{wh} u}{tr \to_{wh} ur}$$

We call *monolithic* the given definition of $\to_{wh}$. It can equivalently be defined in a *split* way, by separating the root rule $\mapsto_{wh}$ and the general rule $\to_{wh}$ obtained by the inductive closure of $\mapsto_{wh}$; or *contextually*, by further compacting the inductive cases via a notion of context.

SPLIT DEFINITION	CONTEXTUAL DEFINITION
$\dfrac{}{(\lambda x.t)u \mapsto_{wh} t\{x\leftarrow u\}}$	APPLICATIVE CONTEXTS $A ::= \langle\cdot\rangle \mid At$
$\dfrac{t \mapsto_{wh} u}{t \to_{wh} u} \qquad \dfrac{t \to_{wh} u}{tr \to_{wh} ur}$	$\dfrac{}{(\lambda x.t)u \mapsto_{wh} t\{x\leftarrow u\}} \qquad \dfrac{t \mapsto_{wh} u}{A\langle t\rangle \to_{wh} A\langle u\rangle}$

These three definitions lead to different formalizations in Abella. For $\to_{wh}$, for which we do not need to prove many properties, we shall adopt the monolithic one. The only two properties of $\to_{wh}$ that we shall use are the following ones.

▶ **Lemma 1.** **1.** Determinism of $\to_{wh}$: *if* $t \to_{wh} u$ *and* $t \to_{wh} r$ *then* $u = r$.
2. Stability under substitution of $\to_{wh}$: *if* $t \to_{wh} u$ *then* $t\{x\leftarrow r\} \to_{wh} u\{x\leftarrow r\}$.

Micro Weak Head Reduction. We do not define the whole reduction of the linear substitution calculus, but only *micro weak head reduction* (also known as *linear* weak head reduction), the micro-step variant of weak head reduction. To define it, beyond explicit substitution (shortened to ES) constructor we need the following contexts with ESs.

$$\begin{array}{rrcl} \lambda\text{-TERMS WITH ESS} & t,u,r & ::= & x \mid \lambda x.t \mid tu \mid t[x\leftarrow u] \\ \text{SUBSTITUTION CONTEXTS} & S,S' & ::= & \langle\cdot\rangle \mid S[x\leftarrow u] \\ \text{(MICRO) WEAK HEAD CONTEXTS} & W,W' & ::= & \langle\cdot\rangle \mid Wu \mid W[x\leftarrow u] \end{array}$$

An explicit substitution $t[x\leftarrow u]$ is an annotation for a meta-level substitution $t\{x\leftarrow u\}$ which has been delayed. As such, it binds x in t but not in u. As it is standard for pen-and-paper reasoning, we work silently modulo α-equivalence, thus in a term such as $(xy)[x\leftarrow xy]$ the left and right occurrences of x are not occurrences of the same variable, as the left one is bound and thus its name is not really x, given that $(xy)[x\leftarrow xy] =_\alpha (zy)[z\leftarrow xy]$.

Substitution contexts are simply lists of ESs. Weak head contexts are the extension to ESs of the applicative contexts given above.

Micro weak head reduction $\to_{mwh}$ is the union of two rules, *β at a distance* $\to_{dB}$ and *micro substitution* $\to_{ms}$, which are usually defined *contextually*. A peculiar aspect of their definition is that contexts are used also in the root case.

$$\text{(Weak head) Beta at a distance} \quad \frac{}{S\langle\lambda x.t\rangle u \mapsto_{dB} S\langle t[x{\leftarrow}u]\rangle} \qquad \frac{t \mapsto_{dB} u}{W\langle t\rangle \to_{dB} W\langle u\rangle}$$

$$\text{(Weak head) Micro substitution} \quad \frac{}{W\langle\!\langle x\rangle\!\rangle[x{\leftarrow}u] \mapsto_{ms} W\langle\!\langle u\rangle\!\rangle[x{\leftarrow}u]} \qquad \frac{t \mapsto_{ms} u}{W\langle t\rangle \to_{ms} W\langle u\rangle}$$

Examples for β at a distance: $(\lambda x.t)[y{\leftarrow}u]r \mapsto_{dB} t[x{\leftarrow}r][y{\leftarrow}u]$ and $(\lambda x.t)[y{\leftarrow}u]r[z{\leftarrow}w] \to_{dB} t[x{\leftarrow}r][y{\leftarrow}u][z{\leftarrow}w]$. The word *distance* refers to the presence of ESs – in fact the substitution context S – between the abstraction and the argument, which then interact *at a distance*. Examples for micro substitution: $((xx)t)[y{\leftarrow}u][x{\leftarrow}r] \mapsto_{ms} ((rx)t)[y{\leftarrow}u][x{\leftarrow}r]$ and $((xx)t)[y{\leftarrow}u][x{\leftarrow}r]w \to_{ms} ((rx)t)[y{\leftarrow}u][x{\leftarrow}r]w$. The word *micro* refers to the fact that the rule replaces one occurrence of x at a time. Note the other occurrence of x in the example, which is left untouched. Note also that the root rule $\mapsto_{ms}$ uses the capture-avoiding form of plugging, while the contextual closure defining $\to_{ms}$ uses the capture-allowing variant.

We give alternative inductive definitions of $\mapsto_{dB}$ / $\mapsto_{ms}$, and of their contextual closures, that shall be used in the discussion on how to formalize $\to_{mwh}$ in Abella, in the next section.

$$\text{Inductive def of } \mapsto_{dB} \quad \frac{}{(\lambda x.t)u \mapsto_{dB} t[x{\leftarrow}u]} \qquad \frac{tu \mapsto_{dB} r}{t[x{\leftarrow}w]u \mapsto_{dB} r[x{\leftarrow}w]}$$

$$\text{Inductive def of } \mapsto_{ms} \quad \frac{}{x[x{\leftarrow}u] \mapsto_{ms} u[x{\leftarrow}u]} \qquad \frac{t[x{\leftarrow}u] \mapsto_{ms} u[x{\leftarrow}u]}{(tr)[x{\leftarrow}u] \mapsto_{ms} (ur)[x{\leftarrow}u]}$$

$$\frac{t[x{\leftarrow}u] \mapsto_{ms} u[x{\leftarrow}u]}{t[y{\leftarrow}r][x{\leftarrow}u] \mapsto_{ms} u[y{\leftarrow}r][x{\leftarrow}u]}$$

$$\text{Ctx closure of } a \in \mapsto_{dB}, \mapsto_{ms} \quad \frac{t \mapsto_a u}{t \to_a u} \qquad \frac{t \to_a u}{tr \to_a ur} \qquad \frac{t \to_a u}{t[y{\leftarrow}r] \to_a u[y{\leftarrow}r]}$$

Reachable Terms. Since, in the context of this paper, micro weak head reduction is just a way to refine weak head reduction, we shall consider only terms with ESs that are reachable from a term without ESs by $\to_{mwh}$. It is easily seen that the following characterization holds.

▶ **Definition 2** (Reachable terms). *Reachable terms (possibly with ESs) are defined as follows*

$$\frac{}{x \text{ is reachable}} \qquad \frac{t \text{ is reachable} \qquad u \text{ has no ESs}}{tu \text{ is reachable}}$$

$$\frac{\lambda x.t \text{ has no ESs}}{\lambda x.t \text{ is reachable}} \qquad \frac{t \text{ is reachable} \qquad u \text{ has no ESs}}{t[x{\leftarrow}u] \text{ is reachable}}$$

▶ **Lemma 3.** *If t has no ESs and $t \to^*_{mwh} u$ then u is reachable.*

Processes. The dialect of the π-calculus that we adopt contains only the constructs needed to represent the λ-calculus. Namely, we use the *asynchronous* π-calculus (thus outputs are not prefixes) with both unary and binary inputs and outputs, and pairing up unary inputs with replication. The grammar is:

$$\text{Processes} \quad P, Q, R \quad ::= \quad 0 \mid \overline{x}\langle y\rangle \mid \overline{x}\langle y, z\rangle \mid \nu x P \mid x(y,z).P \mid !x(y).P \mid P \mid Q$$

For channels, we use the same notation that we use for the variables of λ-terms. We postpone the definition of structural equivalence and of the rewriting rules for processes to Sect. 4.

3 Our Approach to Formalizing λ-Terms, Processes, and Contexts

Reasoning Level. Abella has two layers, the *specification level* and the *reasoning level*. They are based on different logics, the reasoning level being more powerful, having in particular the ∇ (nabla) quantifier, and provided with special tactics to reason about the specification level. In many formalizations in Abella, definitions are given at the specification level while statements and proofs are given at the reasoning level. We follow another approach, giving the definitions at the reasoning level. One of the reasons is that in this way we can exploit ∇ to formalize terms with free variables, obtaining definitions, statements, and reasoning that are closer to those with pen and paper. The same approach is used also (at least) by Tiu and Miller [43], Accattoli [1], Chaudhuri et al. [17], and section 7.3 of the Abella tutorial by Baelde et al. [11].

In this paper, we show Abella code to explain how crucial concepts are formalized, but, for lack of space, we do not systematically pair every definition/statement with its code (for the link to the code see the first page, after the abstract). We also assume basic familiarity with the representation of binders in higher-order abstract formalisms (the one adopted by Abella is Miller and Nadathur's λ-*tree syntax* [30]).

Induction on Types in Abella. In Abella, it is standard to define untyped λ-terms by introducing a type `tm` and two constructors for applications and abstractions, without specifying variables, as follows:

```
Kind    tm    type.
Type    app   tm -> tm -> tm.
Type    abs   (tm -> tm) -> tm.
```

Note that `abs` takes an argument of type `tm -> tm` which is how Abella encodes binders. More precisely, an object-level binding constructor such as $\lambda x.t$ is encoded via a pair: an ordinary constructor `abs` applied to a *meta-level abstraction* of type `tm -> tm`. For example, the term $\lambda x.xx$ that binds x in the scope xx is encoded as `abs x\app x x` (that is parsed as `abs (x\app x x)`) where `x\app x x` is a meta-level abstraction of type `tm -> tm` in the syntax of Abella. In the rest of the paper, with an abuse of terminology, we call *binders* such terms of type `tm -> tm`.

Reasoning by induction on the structure of `tm` terms is not possible in Abella because of the *open world assumption*, stating that new constructors can always be added later to any type. Thus, one rather defines a `is_tm` predicate, as follows, and reasons by induction over it:

```
Define is_tm : tm -> prop by
    nabla x, is_tm x;
    is_tm (abs T) := nabla x, is_tm (T x);
    is_tm (app T U) := is_tm T /\ is_tm U.
```

The first clause uses nabla to say that the free variable `x` is a term. Variables with capitalized names in the last two clauses are implicitly quantified by $\forall$ at the clause level. The second

clause states that an abstraction `abs T` is a term if its body is a term. The body is obtained applying the binder `T` (of type `tm -> tm`) to a fresh variable `x` to obtain a term of type `T`. Such application corresponds in a pen-and-paper proof to the (usually implicit) logical step that replaces the bound variable with a fresh one.

Predicates such as `is_tm` might seem an oddity. In our development, we exploit them crucially, as we now explain.

One Type to Formalize Them All. We deal with three main syntactic categories, ordinary λ-terms, λ-terms with ESs, and processes. In order to circumvent the issue of relating term variables and process names, we formalize the three categories over the same type `pt`, standing for *processes and terms*, and then distinguish them via dedicated predicates. The constructors are defined as follows:

```
Kind pt type.

% terms
Type abs (pt -> pt) -> pt.
Type app pt -> pt -> pt.
Type esub (pt -> pt) -> pt -> pt.
```

```
% pi-calculus terms
Type zero pt.
Type nu (pt -> pt) -> pt.
Type par pt -> pt -> pt.
Type out pt -> pt -> pt.
Type out2 pt -> pt -> pt -> pt.
Type in pt -> (pt -> pt) -> pt.
Type in2 pt -> (pt -> pt -> pt) -> pt.
```

About terms, `esub T U` represents $t[x \leftarrow u]$. For processes, `nu` is the restriction operator $\nu x P$, `par` is parallel composition $P \mid Q$, `out` and `out2` are unary and binary output, and similarly for inputs. Note the binders used by the input prefixes: `in2 x P` represents $x(y,z).P$, where y and z are implicit in the fact that `P` has type `pt -> pt -> pt`. They can be made explicit using the equivalent representation `in2 x (y\z\ P y z)`, where `y\z\P y z` is an η-expansion of `P`: Abella identifies all meta-level abstractions up to α-renaming and η-expansion.

We then use two predicates, one for isolating λ-terms with no ESs and one for reachable λ-terms with ESs – processes shall not need one, as there shall be no inductions on processes.

```
Define tm_with_no_ES : pt -> prop by
  nabla x, tm_with_no_ES x;
  tm_with_no_ES (abs T) := nabla x, tm_with_no_ES (T x);
  tm_with_no_ES (app T U) := tm_with_no_ES T /\ tm_with_no_ES U.

Define reachable_tm : pt -> prop by
  nabla x, reachable_tm x;
  reachable_tm (abs T) := tm_with_no_ES (abs T);
  reachable_tm (app T U) := reachable_tm T /\ tm_with_no_ES U;
  reachable_tm (esub T U) := (nabla x, reachable_tm (T x)) /\ tm_with_no_ES U.
```

Micro Weak Head Reduction. For the formal definitions of the two rules of $\to_{mwh}$ we explored various approaches. We started with the monolithic approach, but soon realized that it was very difficult, if not impossible, to reason about the properties of the rules. We then adopted a split approach. For both $\mapsto_{dB}$ and $\to_{dB}$ (`red_root_db` and `red_db` in the sources), we use the inductive definitions, because their use of context plugging is capture-allowing. For both $\mapsto_{ms}$ and $\to_{ms}$, we initially used the inductive definitions, which is enough to prove the relationship with the π-calculus. To prove the relationship between $\to_{wh}$ and $\to_{mwh}$, which requires a fine analysis of $\mapsto_{ms}$ / $\to_{ms}$ steps, we were however led

to switch (in the whole development) to a mixed style: contextual definition of $\mapsto_{ms}$, which enables finer reasoning, and inductive definition of $\to_{ms}$. This is possible because – crucially – the definition of $\mapsto_{ms}$ uses capture-avoiding plugging $W\langle\!\langle u \rangle\!\rangle$. The idea is that weak head contexts W can be formalized as follows (that is, as a function $u \mapsto W\langle\!\langle u \rangle\!\rangle$).

```
Define weak_head_ctx : (pt -> pt) -> prop by
    weak_head_ctx (h\h);
    weak_head_ctx (h\ app (W h) U) := weak_head_ctx  W;
    weak_head_ctx (h\ esub (x\ W x h) U) := nabla x, weak_head_ctx (W x).
```

We use `h` for *hole*. In the last clause, `h\ esub (x\ W x h) U` specifies that the explicit substitution bounds a variable x which is not the one representing the hole.

We now describe the definition of $\mapsto_{ms}$, based on two predicates, avoiding Abella code for lack of space, but reflecting the Abella formalization faithfully.

Firstly, we need the predicate t *has free head variable* x, shortened $\mathsf{fhv}(t) = x$, defined by:

$$\frac{}{\mathsf{fhv}(x) = x} \qquad \frac{\mathsf{fhv}(t) = x}{\mathsf{fhv}(tu) = x} \qquad \frac{\mathsf{fhv}(t) = x}{\mathsf{fhv}(t[y{\leftarrow}u]) = x}$$

Secondly, the predicate t *has free maximal weak head context* W, shortened $\mathsf{maxw}(t) = W$, defined by:

$$\frac{}{\mathsf{maxw}(x) = \langle\cdot\rangle} \qquad \frac{}{\mathsf{maxw}(\lambda x.t) = \langle\cdot\rangle} \qquad \frac{\mathsf{maxw}(t) = W}{\mathsf{maxw}(tu) = Wu} \qquad \frac{\mathsf{maxw}(t) = W}{\mathsf{maxw}(t[x{\leftarrow}u]) = W[x{\leftarrow}u]}$$

An immediate lemma guarantees that if $\mathsf{fhv}(t) = x$ then there exists a weak head context W such that $\mathsf{maxw}(t) = W$. Now, the micro substitution root rule $\mapsto_{ms}$ (`red_root_ms` in the sources) is defined as follows:

$$\frac{\mathsf{fhv}(t) = x \text{ and } \mathsf{maxw}(t) = W}{t[x{\leftarrow}u] \mapsto_{ms} W\langle\!\langle u \rangle\!\rangle[x{\leftarrow}u]}$$

Finally, the inductive contextual closure that defines $\to_{dB}$ (`red_db`) and $\to_{ms}$ (`red_ms`) is obtained via a higher-order predicate `ctx_cl_tm` taking a relation and returning its contextual closure defined as follows, where $\mathsf{a} \in \{dB, ms\}$.

$$\frac{t \mapsto_{\mathsf{a}} u}{t \to_{\mathsf{a}} u} \qquad \frac{t \to_{\mathsf{a}} u}{tr \to_{\mathsf{a}} ur} \qquad \frac{t \to_{\mathsf{a}} u}{t[x{\leftarrow}r] \to_{\mathsf{a}} u[x{\leftarrow}r]}$$

Note that it is not possible to use a contextual formalization of the contextual closure, because the closure rests on capture-allowing plugging, which is not supported by Abella.

Finally, micro weak head reduction $\to_{mwh}$ (`red_mwh`) is defined as $\to_{dB} \cup \to_{ms}$.

4 The π-Calculus, at a Distance

Here we describe the formalization of the rewriting rules of the π-calculus. We start with structural equivalence and give the standard presentation of the rules. Then, we redefine them according to the *at a distance* approach developed by Accattoli [2].

Structural Equivalence. Processes are considered modulo structural congruence $\equiv$, defined in two steps. First, we define *root structural equivalence* $P \doteq Q$ (`str_eq_root`) via the following clauses.

- *Neutrality of* 0 *with respect to parallel composition*: $P \mid 0 \doteq P$, $0 \mid P \doteq P$, $P \doteq P \mid 0$, and $P \doteq 0 \mid P$;

- *Irrelevance of vacuous name restrictions*: $\nu x P \doteq P$, and $P \doteq \nu x P$ if $x \notin \mathsf{fn}(P)$[1];
- *Commutativity of parallel composition*: $P \mid Q \doteq Q \mid P$
- *Associativity of parallel composition*: $P \mid (Q \mid R) \doteq (P \mid Q) \mid R$ and $(P \mid Q) \mid R \doteq P \mid (Q \mid R)$;
- *Permutation of name restriction and parallel composition*: $\nu x(P \mid Q) \doteq P \mid \nu x Q$ and $P \mid \nu x Q \doteq \nu x(P \mid Q)$ if $x \notin \mathsf{fv}(P)$, $\nu x(P \mid Q) \doteq \nu x P \mid Q$ and $\nu x P \mid Q \doteq \nu x(P \mid Q)$ if $x \notin \mathsf{fv}(Q)$;
- *Commutativity of name restrictions*: $\nu x \nu y P \doteq \nu y \nu x P$.

Note that all clauses but the commutativity ones have symmetric clauses. It is natural to wonder whether is it possible to have only half of the clauses plus a rule for the symmetry of $\doteq$. Such an alternative does not seem to work, as it shall be explained in the next section.

The second step is defining structural equivalence $\equiv$ (`str_eq`) as the equivalence and contextual closure of $\doteq$, what is sometimes called *the congruence closure*. The symmetric definition of $\doteq$ spares us a symmetry rule in the definition of $\equiv$, as proved after the definition.

Defining structural equivalence on top of root structural equivalence

$$\frac{P \doteq Q}{P \equiv Q} \qquad \frac{}{P \equiv P} \qquad \frac{P \equiv Q \qquad Q \equiv R}{P \equiv R}$$

$$\frac{P \equiv Q}{\nu x P \equiv \nu x Q} \qquad \frac{P \equiv Q}{P \mid R \equiv Q \mid R} \qquad \frac{P \equiv Q}{R \mid P \equiv R \mid Q}$$

▶ **Lemma 4.**

1. Symmetry of $\doteq$: *if* $P \doteq Q$ *then* $Q \doteq P$.
2. Symmetry of $\equiv$: *if* $P \equiv Q$ *then* $Q \equiv P$.

Proof. Point 1 is by case analysis: every clause of $\doteq$ is either symmetric or has a symmetric clause. Point 2 is by induction on $P \equiv Q$, using Point 1 in the case for $\doteq$. ◀

The Ordinary Rewriting Rules. The π-calculus at work here has two rewriting rules, a linear one for binary communication and a rule involving process replication for unary communication. The root binary and replication rules are defined as follows (`ord_pi-red_root_bin P Q` and `ord_pi-red_root_rep P Q` in Abella).

$$\overline{x}\langle y, z\rangle \mid x(y', z').Q \mapsto_{bin} Q\{y' \leftarrow y\}\{z' \leftarrow z\} \qquad \overline{x}\langle y\rangle \mid !x(z).Q \mapsto_{!} Q\{z \leftarrow y\} \mid !x(z).Q$$

The root rules are closed contextually under restrictions and parallel composition as follows.

Contextual closure of root rules on processes, $a \in \{bin, !\}$

$$\frac{P \mapsto_a Q}{\nu x P \to_a \nu x Q} \qquad \frac{P \to_a Q}{\nu x P \to_a \nu x Q} \qquad \frac{P \to_a Q}{P \mid R \to_a Q \mid R} \qquad \frac{P \to_a Q}{R \mid P \to_a R \mid Q} \tag{1}$$

In Abella, the closure is realised via a higher-order closure predicate `ctx_cl_pr`, similarly to what is done for micro weak head reduction.

The rules at work in the π-calculus are actually $\to_{bin}$ and $\to_{!}$ *modulo* $\equiv$, that is, one rather considers the rules $\to_{bin/\equiv}$ and $\to_{!/\equiv}$ where $P \to_{bin/\equiv} Q$ if there exist P' and Q' such that $P \equiv P' \to_{bin} Q' \equiv Q$ (one might also compactly write $\to_{bin/\equiv} := \equiv \to_{bin} \equiv$) and similarly for $\to_{!/\equiv}$.

[1] In [2], Accattoli uses $\nu x 0 \doteq 0$, and $0 \doteq \nu x 0$, which is correct, but then requires one to prove the general version as an easy lemma.

Towards Communication at a Distance. The use of structural equivalence in the definition of the rewriting relation of the π-calculus induces some annoying complications when one tries to reflect process reductions on terms. We are then going to reformulate the π-calculus reduction rules *at a distance*, that is, in a way that allows us to prove a postponement theorem with respect to structural equivalence, inspired by Accattoli [2] but in a slightly different way. Let us recall the idea from [2].

The first step is to define *non-blocking contexts*, which are the contexts used in the contextual closure in (1), as follows:

$$\text{Non-blocking ctxs} \quad N, M \quad ::= \quad \langle\cdot\rangle \;\mid\; N \,|\, Q \;\mid\; P \,|\, N \;\mid\; \nu x N$$

The second step is to generalize the root case of, say, the binary rule, as follows:

$$\frac{N \text{ and } M \text{ do not capture } x}{N\langle \overline{x}\langle y, z\rangle\rangle \,|\, M\langle x(y', z').Q\rangle \;\mapsto_{bin}\; N\langle M\langle Q\{y'{\leftarrow}y\}\{z'{\leftarrow}z\}\rangle\rangle} \tag{2}$$

and then one closes the root rule by contexts as in (1). To be precise, since the aim is to avoid structural equivalence in rewriting steps, one also needs the symmetric case of (2). The idea behind the postponement is that the role of structural equivalence $\equiv$ in the ordinary approach is to re-organize the term as to move the non-blocking contexts N and M out of the way, that is, as to obtain:

$$N\langle \overline{x}\langle y, z\rangle\rangle \,|\, M\langle x(y', z').Q\rangle \equiv N\langle M\langle \overline{x}\langle y, z\rangle \,|\, x(y', z').Q\rangle\rangle$$

to then allow one to apply the ordinary communication rule. The approach at a distance avoids the re-organization altogether, by defining communication *up to* non-blocking contexts.

Here, we slightly refine the presented idea, in two respects. Firstly, the plugging at work in (2) is capture-allowing (in contrast to the root case of $\mapsto_{ms}$ for terms), so that we replace it with an inductive contextual closure. Secondly, the scheme in (2) is *asymmetric*: in the reduct, the contexts are composed as $N\langle M\rangle$, while the opposite composition would work as well. By turning to an inductive contextual closure, we can restore the symmetry. In fact, we obtain a strictly more permissive rule, as we permit the reduct to have any shuffling of the constructors in N and M. The rule is then non-deterministic, but harmlessly so, as all the reductions of a same redex are structurally equivalent.

Communication at a Distance. The rule at a distance for binary prefixes $\Rrightarrow_{bin}$ is obtained by first defining its *root* variant $\Rightarrow_{bin}$ (`new_pi-red_root_bin P Q` in Abella), as follows.

$$\frac{}{\overline{x}\langle y, z\rangle \,|\, x(y', z').Q \;\Rightarrow_{bin}\; Q\{y'{\leftarrow}y\}\{z'{\leftarrow}z\}} \qquad \frac{}{x(y', z').Q \,|\, \overline{x}\langle y, z\rangle \;\Rightarrow_{bin}\; Q\{y'{\leftarrow}y\}\{z'{\leftarrow}z\}}$$

$$\frac{P \,|\, Q \Rightarrow_{bin} R}{(P \,|\, O) \,|\, Q \Rightarrow_{bin} R \,|\, O} \qquad \frac{P \,|\, Q \Rightarrow_{bin} R}{P \,|\, (Q \,|\, O) \Rightarrow_{bin} R \,|\, O} \qquad \frac{P \,|\, Q \Rightarrow_{bin} R}{\nu x P \,|\, Q \Rightarrow_{bin} \nu x R}$$

$$\frac{P \,|\, Q \Rightarrow_{bin} R}{(O \,|\, P) \,|\, Q \Rightarrow_{bin} O \,|\, R} \qquad \frac{P \,|\, Q \Rightarrow_{bin} R}{P \,|\, (O \,|\, Q) \Rightarrow_{bin} O \,|\, R} \qquad \frac{P \,|\, Q \Rightarrow_{bin} R}{P \,|\, \nu x Q \Rightarrow_{bin} \nu x R}$$

Then, $\Rrightarrow_{bin}$ (`new_pired_bin P Q` in Abella) is obtained by applying the same context closure as in (1) to $\Rightarrow_{bin}$.

The rule for unary prefixes $\Rrightarrow_!$ is obtained via the same construction by simply changing the base case. The root case $\Rightarrow_!$ (`new_pi-red_root_rep P Q` in Abella) follows.

$$\frac{}{\overline{x}\langle y\rangle \,|\, !x(z).Q \mapsto_! Q\{z{\leftarrow}y\} \,|\, !x(z).Q} \qquad \frac{}{!x(z).Q \,|\, \overline{x}\langle y\rangle \mapsto_! !x(z).Q \,|\, Q\{z{\leftarrow}y\}}$$

$$\frac{P \,|\, Q \Rrightarrow_! R}{(P \,|\, O) \,|\, Q \Rrightarrow_! R \,|\, O} \qquad \frac{P \,|\, Q \Rrightarrow_! R}{P \,|\, (Q \,|\, O) \Rrightarrow_! R \,|\, O} \qquad \frac{P \,|\, Q \Rrightarrow_! R}{\nu x P \,|\, Q \Rrightarrow_! \nu x R}$$

$$\frac{P \,|\, Q \Rrightarrow_! R}{(O \,|\, P) \,|\, Q \Rrightarrow_! O \,|\, R} \qquad \frac{P \,|\, Q \Rrightarrow_! R}{P \,|\, (O \,|\, Q) \Rrightarrow_! O \,|\, R} \qquad \frac{P \,|\, Q \Rrightarrow_! R}{P \,|\, \nu x Q \Rrightarrow_! \nu x R}$$

The general rule (`new_pi-red_rep P Q` in Abella) is then obtained by a contextual closure, as for $\Rrightarrow_{bin}$. Finally, we set $\Rrightarrow := \Rrightarrow_{bin} \cup \Rrightarrow_!$. Some basic properties of reduction follow.

▶ **Lemma 5.** *Let $a \in \{bin, !\}$.*
1. No creation of free names*: if $P \Rrightarrow_a Q$ then $\mathsf{fv}(Q) \subseteq \mathsf{fv}(P)$.*
2. Parallel symmetry*: if $P \,|\, Q \Rightarrow_a R$ then exists O such that $Q \,|\, P \Rightarrow_a O$ and $O \equiv R$.*

5 Postponement of $\equiv$ and Equivalence of the Presentations

The main property of the presentation at a distance is that $\equiv$ strongly postpones with respect to $\Rrightarrow_{bin}$ and $\Rrightarrow_!$, that is, it is not required for reduction. A noticeable point of the proof is that the two cases of $\Rrightarrow_{bin}$ and $\Rrightarrow_!$ are handled in the *exact* same way: all the statements and all the proofs are indeed parametric in the reduction rule (the Abella proofs are identically structured but not proved parametrically). We need two auxiliary lemmas.

▶ **Lemma 6** (Auxiliary properties for postponement of structural equivalence)**.** *Let $a \in \{bin, !\}$.*
1. *If $(P \,|\, Q) \,|\, R \Rightarrow_a O$ then (exists O' such that $P \,|\, R \Rightarrow_a O'$ and $O \equiv O' \,|\, Q$) or (exists O' such that $Q \,|\, R \Rightarrow_a O'$ and $O \equiv P \,|\, O'$).*
2. *If $\nu x P \,|\, Q \Rightarrow_a R$ then there exists O such that $P \,|\, Q \Rightarrow_a O$ and $\nu x O \equiv R$.*

Proof. Every point is by induction on the $\Rightarrow_a$ step in its hypothesis. In Point 1, for $a = bin$ one actually has $O = R$. ◀

The lemma allows us to prove the postponement in the root case of structural equivalence, that we prefer to isolate to stress that it does not need an induction and because it does not need the lemma that follows it.

▶ **Proposition 7** (Root strong postponement of $\equiv$ wrt $\Rrightarrow$)**.** *Let $a \in \{bin, !\}$. If $P \doteq P'$ and $P \Rrightarrow_a Q$ then there exists Q' such that $P' \Rrightarrow_a Q'$ and $Q' \equiv Q$.*

Proof. By case analysis of $P \doteq P'$, using Lemma 5 and Lemma 6. ◀

To deal with the general case of structural equivalence, we need a further auxiliary lemma.

▶ **Lemma 8.** *Let $a \in \{bin, !\}$. If $P \equiv P'$ and $P \,|\, Q \Rrightarrow_a R$ then there exists R' such that $P' \,|\, Q \Rrightarrow_a R'$ and $R \equiv R'$.*

Proof. By case analysis of $P \doteq P'$, using Lemma 5 and Lemma 6. ◀

▶ **Theorem 9** (Strong postponement of $\equiv$ wrt $\Rrightarrow$)**.** *Let $a \in \{bin, !\}$. If $P \equiv P'$ and $P \Rrightarrow_a Q$ then there exists Q' such that $P' \Rrightarrow_a Q'$ and $Q' \equiv Q$.*

Proof. By induction on $P \equiv P'$. The case for $\doteq$ is exactly Prop. 7. The case of reflexivity is immediate. The case of contextual closure with respect to parallel composition on the right uses Lemma 8, and the case on the left uses Lemma 8 and Lemma 5.2. The other cases (transitivity and name restriction closure) follow by the *i.h.* ◀

▶ Remark 10. We can now explain why adding a symmetry rule to the definition of $\equiv$ (or $\doteq$), and so dividing by 2 the number of cases defining $\doteq$, does not work. Consider the proof of the strong postponement property for the symmetry case: the hypotheses of the theorem are $P \equiv P'$ and $P \Rrightarrow_a Q$, and the inductive case we are facing is that $P \equiv P'$ because $P' \equiv P$. To be able to apply the *i.h.* one should have $P' \Rrightarrow_a R$ for some R, which is what we actually have to prove. Note also that $P' \Rrightarrow_a R$ is not enough because the *i.h.* would then give a process O which is not necessarily Q. Therefore, to prove the statement one should have $P' \Rrightarrow_a Q$, which is not even true.

Equivalence of Presentations. The strong postponement property is the key point in showing that $\Rrightarrow_a \equiv$ is equivalent to $\rightarrow_{a/\equiv}$ for $a \in \{bin, !\}$.

▶ **Lemma 11.** *Let $a \in \{bin, !\}$. If $P \rightarrow_a Q$ then $P \Rrightarrow_a Q$.*

▶ **Theorem 12** (Ordinary and new presentations are equivalent)**.** *Let $a \in \{bin, !\}$.*
1. *If $P \rightarrow_{a/\equiv} Q$ then there exists R such that $P \Rrightarrow_a R \equiv Q$.*
2. *If $P \Rightarrow_a Q$ then $P \rightarrow_{a/\equiv} Q$.*
3. *If $P \Rrightarrow_a Q$ then $P \rightarrow_{a/\equiv} Q$.*

Proof.
1. Explicitly, $P \equiv P' \rightarrow_a Q' \equiv Q$. By Lemma 11, $P' \Rrightarrow_a Q'$. By strong postponement (Theorem 9), there exists R such that $P \Rrightarrow_a R \equiv Q'$, and, by transitivity of structural equivalence, $P \Rrightarrow_a R \equiv Q$.
2. Immediate induction on $P \Rightarrow_a Q$.
3. Immediate induction on $P \Rrightarrow_a Q$, using the previous point in the base case. ◀

6 Translation and Simulations

Milner's Translation. Here we present Milner's call-by-name translation $[\![t]\!]$ of the λ-calculus to the π-calculus, extended to account also for ESs. Let $a, b, c, \ldots$ be *special channel names*. Milner presented the translation as $[\![t]\!]_a$, that is, as parametrized by a special channel name a, meant to be the channel on which t itself can be communicated. Sometimes, typically by Sangiorgi, the translation is rather presented moving the parametrization on the π-calculus side, stating that the representation of a λ-term (with ESs) is a function $\lambda\!\!\lambda a.P$ (where $\lambda\!\!\lambda a.$ is a meta-level notation not part of the syntax of processes) which when applied to b gives the process $(\lambda\!\!\lambda a.P)_b = P\{a \leftarrow b\}$. While both approaches are viable (and we explored both), we prefer Sangiorgi's. Firstly, it can easily be represented in Abella, by seeing the functions $\lambda\!\!\lambda a.P$ as process binders. Secondly, it reduces the amount of free names that have to be managed (too many free names make Abella produce unreadable intermediate goals during the formalization). The translation is then defined as follows.

Translation of terms with ESs to processes

$$
\begin{array}{rcll}
[\![x]\!] & := & \lambda\!\!\lambda a.\overline{x}\langle a\rangle & \\
[\![\lambda x.t]\!] & := & \lambda\!\!\lambda a.a(x,b).[\![t]\!]_b & a \notin \mathsf{fv}([\![t]\!]_b) \\
[\![tu]\!] & := & \lambda\!\!\lambda a.\nu b \nu x([\![t]\!]_b \mid \overline{b}\langle x, a\rangle \mid !x(c).[\![u]\!]_c) & a, b \notin \mathsf{fv}(!x(c).[\![u]\!]_c) \\
 & & & a \notin \mathsf{fv}([\![t]\!]_b), x \notin \mathsf{fv}(t) \cup \mathsf{fv}(u) \\
[\![t[x \leftarrow u]]\!] & := & \lambda\!\!\lambda a.\nu x([\![t]\!]_a \mid !x(b).[\![u]\!]_b) & a \notin \mathsf{fv}([\![u]\!]_b)
\end{array}
$$

Beyond the given side conditions, the translation rests on the assumption that the special names do not occur as variables of terms. Note that the subjects of binary input/outputs are always special names, while the subjects of unary input/outputs are variable names. We

distinguish special names from variables for readability and because the distinction helps us understand the translation. There is no formal need to distinguish them, however, and indeed our Abella formalization, which follows, does not use a separate category for them.

```
Define translation : pt -> (pt -> pt) -> prop by
  nabla x, translation x (out x);
  translation (abs T) (a\in2 a P) := nabla x, translation (T x) (P x);
  translation (app T U) (a\nu b\nu x\par (P b) (par (out2 b x a) (in x Q)))
      := translation T P /\ translation U Q;
  translation (esub T U) (a\nu x\par (P x a) (in x Q))
      := (nabla x, translation (T x) (P x)) /\ translation U Q.
```

Note the use of `out x` in the translation of a variable x. The type of `out x` is `pt -> pt -> pt`, so the unary output $\overline{x}\langle a\rangle$ is represented by `out x a`. But the translation uses `out x` instead: this is done to map x to the function (or binding) $\lambda\!\!\lambda a.\overline{x}\langle a\rangle$ rather than to $\overline{x}\langle a\rangle$. Similar remarks apply to the other cases. Note also that the Abella definition states the side conditions *dually*, by saying where variables *can* appear, rather than where they do not, *e.g.* `b` can appear in `P b` but not in `Q`, in the application case.

π Simulates $\to_{mwh}$. A single step of the micro weak head reduction rules $\to_{dB}$ and $\to_{ms}$ is simulated in the π-calculus by exactly one step of $\Rrightarrow_{bin}$ and $\Rrightarrow_!$, respectively, followed by structural equivalence. We detail the case of $\to_{ms}$, mimicking closely the proof in Abella. One needs an auxiliary lemma essentially capturing the simulation of the root case $\mapsto_{ms}$ by the root case $\Rightarrow_!$, and then the simulation extends smoothly to the contextual closures.

▶ **Lemma 13** (Auxiliary lemma for $\mapsto_{ms}$). *Let* $\mathsf{fhv}(t) = x$, $\mathsf{maxw}(t) = W$, $[\![t[x{\leftarrow}u]]\!] = \lambda\!\!\lambda a.\nu x P$, *and* $[\![W\langle u\rangle[x{\leftarrow}u]]\!] = \lambda\!\!\lambda a.\nu x Q$. *Then there exists* R *such that* $P \Rightarrow_! R$ *and* $R \equiv Q$.

Proof. By induction on $\mathsf{fhv}(t) = x$. Note that $[\![t[x{\leftarrow}u]]\!] = \lambda\!\!\lambda a.\nu x([\![t]\!]_a \mid !x(b).[\![u]\!]_b)$ and $[\![W\langle u\rangle[x{\leftarrow}u]]\!] = \lambda\!\!\lambda a.\nu x([\![W\langle u\rangle]\!]_a \mid !x(b).[\![u]\!]_b)$, thus we have to prove that there is R such that

$$P = [\![t]\!]_a \mid !x(b).[\![u]\!]_b \quad \Rightarrow_! \quad R \quad \equiv \quad [\![W\langle u\rangle]\!]_a \mid !x(b).[\![u]\!]_b = Q.$$

Cases of $\mathsf{fhv}(t) = x$:

1. *Head variable*, that is, $t = x$. Then $W = \langle\cdot\rangle$ and $W\langle u\rangle = u$. We have:
$$P \quad = \quad [\![x]\!]_a \mid !x(b).[\![u]\!]_b \quad = \quad \overline{x}\langle a\rangle \mid !x(b).[\![u]\!]_b \quad \Rightarrow_! \quad [\![u]\!]_a \mid !x(b).[\![u]\!]_b \quad = \quad Q$$
2. *Left of application*, that is, $t = rw$ with $\mathsf{fhv}(r) = x$. Then $W = W'w$ and $W\langle u\rangle = W'\langle u\rangle w$. By *i.h.*, there exists R' such that
$$[\![r]\!]_a \mid !x(b).[\![u]\!]_b \quad \Rightarrow_! \quad R' \quad \equiv \quad [\![W'\langle u\rangle]\!]_a \mid !x(b).[\![u]\!]_b.$$
Then:
$$\begin{array}{rcll} P & = & [\![rw]\!]_a \mid !x(b).[\![u]\!]_b & = \\ & & \nu c\nu y([\![r]\!]_c \mid \overline{c}\langle y, a\rangle \mid !y(d).[\![w]\!]_d) \mid !x(b).[\![u]\!]_b & \Rightarrow_! \quad (i.h.) \\ & & \nu c\nu y(R' \mid \overline{c}\langle y, a\rangle \mid !y(d).[\![w]\!]_d) & \equiv \quad (i.h.) \\ & & \nu c\nu y([\![W'\langle u\rangle]\!]_a \mid !x(b).[\![u]\!]_b \mid \overline{c}\langle y, a\rangle \mid !y(d).[\![w]\!]_d) & \equiv \\ & & \nu c\nu y([\![W'\langle u\rangle]\!]_a \mid \overline{c}\langle y, a\rangle \mid !y(d).[\![w]\!]_d) \mid !x(b).[\![u]\!]_b & = \\ & & [\![W\langle u\rangle w]\!]_a \mid !x(b).[\![u]\!]_b & = \quad Q \end{array}$$
3. *Left of substitution*, that is, $t = r[y{\leftarrow}w]$ with $\mathsf{fhv}(r) = x$. Then $W = W'[y{\leftarrow}w]$ and $W\langle u\rangle = W'\langle u\rangle[y{\leftarrow}w]$. By *i.h.*, there exists R' such that
$$[\![r]\!]_a \mid !x(b).[\![u]\!]_b \quad \Rightarrow_! \quad R' \quad \equiv \quad [\![W'\langle u\rangle]\!]_a \mid !x(b).[\![u]\!]_b.$$
Then:

$$
\begin{array}{rcll}
P & = & [\![r[y{\leftarrow}w]]\!]_a \mid !x(b).[\![u]\!]_b & = \\
 & & \nu y([\![t]\!]_a \mid !y(c).[\![w]\!]_c) \mid !x(b).[\![u]\!]_b & \Rrightarrow_! \quad (i.h.) \\
 & & \nu y(R' \mid !y(c).[\![w]\!]_c) & \equiv \quad (i.h.) \\
 & & \nu y([\![W'\langle u\rangle]\!]_a \mid !x(b).[\![u]\!]_b \mid !y(c).[\![w]\!]_c) & \equiv \\
 & & \nu c\nu y([\![W'\langle u\rangle]\!]_a \mid !y(c).[\![w]\!]_c) \mid !x(b).[\![u]\!]_b & = \\
 & & [\![W\langle u\rangle[y{\leftarrow}w]]\!]_a \mid !x(b).[\![u]\!]_b & = \quad Q
\end{array}
$$

◀

▶ **Proposition 14.** *Let t be reachable.*

1. *If $t \mapsto_{ms} u$ then $[\![t]\!] \Rrightarrow_! [\![u]\!]$.*
2. *If $t \to_{ms} u$ then $[\![t]\!] \Rrightarrow_! [\![u]\!]$.*

Proof. For Point 1, note that if $t \mapsto_{ms} u$ then $t = r[x{\leftarrow}w]$ with $\mathsf{fhv}(r) = x$ and $\mathsf{maxw}(r) = W$, so that $u = W\langle w\rangle[x{\leftarrow}w]$. Since the translation of ESs starts with $\lambda\!\!\lambda a.\nu x$, we apply Lemma 13 and obtain $[\![t]\!] \Rightarrow_! [\![u]\!]$. Point 2 is an induction on $t \to_{ms} u$ using Point 1 in the base case. ◀

$\to_{mwh}$ Simulates π. The converse simulation follows exactly the same schema, the root case needs an auxiliary lemma (proof omitted), and then the simulation smoothly lifts to the contextual closures. The only difference, in Abella, is that the case analyses of π-calculus reduction require a few simple lemmas (omitted here) to rule out some impossible cases.

▶ **Lemma 15** (Auxiliary lemma for $\Rrightarrow_!$)**.** *Let $[\![t]\!] = \lambda\!\!\lambda a.P$, $[\![u]\!] = \lambda\!\!\lambda b.Q$. If $P \mid !x(b).Q \Rrightarrow_! R$ then $\mathsf{fhv}(t) = x$ and $\exists$ W and O s.t. $\mathsf{maxw}(t) = W$, $[\![W\langle u\rangle[x{\leftarrow}u]]\!] = \lambda\!\!\lambda a.\nu x O$, and $O \equiv R$.*

▶ **Proposition 16.**

1. *If $[\![t]\!] = \lambda\!\!\lambda a.\nu x P$ and $P \Rrightarrow_! Q$ then $\exists$ u and R s.t. $t \mapsto_{ms} u$, $[\![u]\!] = \lambda\!\!\lambda a.\nu x R$ and $Q \equiv R$.*
2. *If $[\![t]\!] = \lambda\!\!\lambda a.P$ and $[\![t]\!]_a = P \Rrightarrow_! Q$ then $\exists$ u and R s.t. $t \to_{ms} u$, $[\![u]\!] = \lambda\!\!\lambda a.R$ and $Q \equiv R$.*

Proof.

1. For $P \Rrightarrow_! Q$ to hold, t has to be an ES $t = r[x{\leftarrow}w]$ and r and w verify the hypotheses of Lemma 15. Then the conclusions of the lemma are exactly that $r = W\langle x\rangle$, and so $t \mapsto_{\mathsf{ls}} u$ with $u := W\langle w\rangle[x{\leftarrow}w]$, and that there exist processes as in the statement.
2. By induction on $[\![t]\!] = \lambda\!\!\lambda a.P$. Cases:
   - **a.** *Variable*: $t = x$. Impossible because then $P = \overline{x}\langle a\rangle$ which is $\Rrightarrow_!$-normal.
   - **b.** *Abstraction*: $t = \lambda x.r$. Impossible, because $P = a(x,b).[\![t]\!]_b$ is $\Rrightarrow_!$-normal.
   - **c.** *Application*: $t = rw$. Then $P = \nu b\nu x([\![r]\!]_b \mid \overline{b}\langle x,a\rangle \mid !x(c).[\![w]\!]_c)$. Cases of $P \Rrightarrow_! Q$:
     - **i.** *Root step of $P' = [\![r]\!]_b \mid \overline{b}\langle x,a\rangle \mid !x(c).[\![w]\!]_c \Rrightarrow_! Q'$.* Since by definition of the translation $x \notin \mathsf{fv}([\![r]\!]_b)$, there cannot be any root step in P'. In Abella proving this fact requires a couple of straightforward auxiliary lemmas.
     - **ii.** *Inductive*, that is, $P \Rrightarrow_! Q$ because $[\![r]\!]_b \Rrightarrow_! Q'$ for some Q'. By *i.h.*, there exist u' and R' such that $r \to_{\mathsf{ls}} u'$, $[\![u']\!] = \lambda\!\!\lambda b.R'$, and $Q' \equiv R'$. Then $t = rw \to_{\mathsf{ls}} u'w = u$. By applying these equalities to the step $P \Rrightarrow_! Q$ we obtain:

$$
\begin{array}{rcll}
P & = & \nu b\nu x([\![r]\!]_b \mid \overline{b}\langle x,a\rangle \mid !x(c).[\![w]\!]_c) & \\
 & \Rrightarrow_! & \nu b\nu x(Q' \mid \overline{b}\langle x,a\rangle \mid !x(c).[\![w]\!]_c) & \\
(i.h.) & \equiv & \nu b\nu x(R' \mid \overline{b}\langle x,a\rangle \mid !x(c).[\![w]\!]_c) & \\
(i.h.) & \equiv & \nu b\nu x([\![u']\!]_b \mid \overline{b}\langle x,a\rangle \mid !x(c).[\![w]\!]_c) & =: \quad R
\end{array}
$$

     Note that $[\![u]\!] = [\![u'w]\!] = \lambda\!\!\lambda a.R$.
   - **d.** *Substitution*: $t = r[y{\leftarrow}w]$. Then $P = \nu x([\![r]\!]_a \mid !x(b).[\![w]\!]_b)$. Cases of $P \Rrightarrow_! Q$:
     - **i.** *Root step of $[\![r]\!]_a \mid !x(b).[\![w]\!]_b \Rrightarrow_! Q'$.* Then it follows from Point 1.

ii. *Inductive*, that is, $P \Rrightarrow_! Q$ because $[\![r]\!]_a \Rrightarrow_! Q'$ for some Q'. By *i.h.*, there exist u' and R' such that $r \to_{\mathsf{ls}} u'$, $[\![u']\!] = \lambda\!\!\lambda a.R'$, and $Q' \equiv R'$. Then $t = r[y{\leftarrow}w] \to_{\mathsf{ls}} u'[y{\leftarrow}w] = u$. By applying these equalities to the step $P \Rrightarrow_! Q$ we obtain:

$$\begin{array}{rcll} P & = & \nu x([\![r]\!]_a \,|\, !x(b).[\![w]\!]_b) & \\ & \Rrightarrow_! & \nu x(Q' \,|\, !x(b).[\![w]\!]_b) & \\ (i.h.) & \equiv & \nu x(R' \,|\, !x(b).[\![w]\!]_b) & \\ (i.h.) & \equiv & \nu x([\![u']\!]_a \,|\, !x(b).[\![w]\!]_b) & =: \quad R \end{array}$$

Note that $[\![u]\!] = [\![u'w]\!] = \lambda\!\!\lambda a.R$. ◀

Summing Up. By iterating the simulation of single steps, and postponing structural equivalence, we obtain our first main result (point 1 is `redn_pi_simulates_redn_mwh` and point 2 is `redn_lwh_simulates_redn_pi` in the sources).

▶ **Theorem 17.**

1. π simulates LSC*: if t is reachable and $t \to^n_{mwh} u$ then for every a there exists Q such that $[\![t]\!]_a \Rrightarrow^n Q$ and $Q \equiv [\![u]\!]_a$.*
2. *LSC simulates π: if $[\![t]\!]_a \Rrightarrow^n Q$ then there exists u such that $t \to^n_{mwh} u$ and $[\![u]\!]_a \equiv Q$.*

By composing Theorem 17 with the equivalence of presentations for the reduction of π (Theorem 12), one can also relate the LSC to the ordinary reduction of π.

7 Relating Weak Head Reduction and Micro Weak Head Reduction

Here we study the relationship between $\to_{wh}$ and $\to_{mwh}$ via the *unfolding* of ESs. We present the pen-and-paper analogous of the Abella formalization, omitting trivial lemmas.

Unfolding. Terms with ES can be *unfolded* into terms without ESs by turning ESs into meta-level substitutions. As explained in Sect. 2, we restrict to *reachable* terms with ESs, characterized by having no ESs in arguments, inside ESs, and under abstractions. Accordingly, the following definition of unfolding $t{\downarrow}$ assumes that it is applied to a reachable term t.

UNFOLDING

$$\begin{array}{rcl@{\qquad}rcl} x{\downarrow} & ::= & x & (tu){\downarrow} & ::= & t{\downarrow}\, u \\ (\lambda x.t){\downarrow} & ::= & \lambda x.t & t[x{\leftarrow}u]{\downarrow} & ::= & t{\downarrow}\,\{x{\leftarrow}u\} \end{array}$$

Projection Via Unfolding. The unfolding turns every weak head β at a distance step $\to_{dB}$ into exactly one weak head step $\to_{wh}$ on the unfolded terms, and every micro substitution step $\to_{ms}$ into an equality. Here we detail only the proof for $\to_{ms}$ steps, which is more interesting and requires an auxiliary lemma.

▶ **Proposition 18.** *Let t be a reachable term. If $t \to_{dB} u$ then $t{\downarrow} \to_{wh} u{\downarrow}$.*

▶ **Lemma 19** (Auxiliary lemma for $\mapsto_{ms}$)**.** *Let $\mathsf{maxw}(t) = W$, t be reachable, u be a term with no ES, and $x \notin \mathsf{fv}(u)$. Then $W\langle x\rangle{\downarrow}\{x{\leftarrow}u\} = W\langle u\rangle{\downarrow}\{x{\leftarrow}u\}$.*

The proof is an easy induction but Abella has trouble with it because the conclusion of the statement is a non-pattern equality. Therefore, the inductive cases need help from the user.

Proof. By induction on $\mathsf{maxw}(t) = C$. Cases:

- $\mathsf{maxw}(y) = \langle\cdot\rangle$, $\mathsf{maxw}(x) = \langle\cdot\rangle$, and $\mathsf{maxw}(\lambda x.r) = \langle\cdot\rangle$ are identical: $W\langle x\rangle{\downarrow}\{x{\leftarrow}u\} = x\{x{\leftarrow}u\} = u = u\{x{\leftarrow}u\} =^* u{\downarrow}\{x{\leftarrow}u\} = W\langle u\rangle{\downarrow}\{x{\leftarrow}u\}$, where the $=^*$ steps holds because unfolding terms with no ES does nothing (it is an easy omitted lemma).

- $\mathsf{maxw}(rw) = W'w$ because $\mathsf{maxw}(r) = W'$. Note that t reachable implies r reachable. By *i.h.*, $W'\langle x\rangle{\downarrow}\{x{\leftarrow}u\} = W'\langle u\rangle{\downarrow}\{x{\leftarrow}u\}$. Then $W\langle x\rangle{\downarrow}\{x{\leftarrow}u\} = W\langle x\rangle{\downarrow}\{x{\leftarrow}u\}w\{x{\leftarrow}u\} = W'\langle u\rangle{\downarrow}\{x{\leftarrow}u\}w\{x{\leftarrow}u\} = W\langle u\rangle{\downarrow}\{x{\leftarrow}u\}$.
- $\mathsf{maxw}(r[y{\leftarrow}w]) = W'[y{\leftarrow}w]$ because $\mathsf{maxw}(r) = W'$. Similar to the previous point. ◀

▶ **Proposition 20.** *Let t be a reachable term.*
1. *If $t \mapsto_{ms} u$ then $t{\downarrow} = u{\downarrow}$.*
2. *If $t \to_{ms} u$ then $t{\downarrow} = u{\downarrow}$.*

Proof.
1. Unfolding the hypothesis, we obtain $t = W\langle x\rangle[x{\leftarrow}r] \mapsto_{ms} W\langle r\rangle[x{\leftarrow}r] = u$. Note that $t{\downarrow} = W\langle x\rangle{\downarrow}\{x{\leftarrow}r\}$ and $u{\downarrow} = W\langle r\rangle{\downarrow}\{x{\leftarrow}r\}$. By Lemma 19, they coincide.
2. By induction on $t \to_{ms} u$ using point 1 in the base case. ◀

Converse Simulation. The converse simulation, that is, that every $\to_{wh}$ step is simulated by a sequence of $\to_{mwh}$ steps, is less easy, and it is where the difficulty of relating small-step and micro-step formalisms lies. While it is true that if t is a term with no ESs and $t \to_{wh} u$ then $t \to_{dB} r$ and $r{\downarrow} = u$, such a property cannot be used for the simulation of rewriting *sequences*, as it does not compose for consecutive steps: if then $u \to_{wh} u'$ we cannot apply the property again because $r \neq u$. One needs the following refined *one-step reflection property*:

$$\text{If } t{\downarrow} \to_{wh} u \text{ then there exists } r \text{ and } w \text{ such that } t \to_{ms}^{*} r \to_{dB} w \text{ and } w{\downarrow} = u$$

To prove such a reflection, we need various properties, in particular that $\to_{ms}$ terminates.

Micro Substitution Normal Terms. For proving the termination of $\to_{ms}$ we use a predicate characterizing $\to_{ms}$-normal terms. We need the concept of answer.

▶ **Definition 21** (Answer). *An* answer *is a term of the form $a ::= \lambda x.t \mid a[x{\leftarrow}t]$.*

The predicate characterizing $\to_{ms}$-normal terms is the disjunction of three predicates.

▶ **Definition 22** (*ms*-normal). *The predicate t is ms-normal is defined as follows.*

$$\frac{a \text{ answer}}{a \text{ is ms-normal}} \qquad \frac{\mathsf{fhv}(t) = x}{t \text{ is ms-normal}} \qquad \frac{t \to_{dB} u}{t \text{ is ms-normal}}$$

The characterization takes the following form in Abella. Curiously, our proof of termination of $\to_{ms}$ relies on the *ms*-normal predicate but never uses its characterization (which we have nonetheless formalized).

▶ **Proposition 23** (Characterization of being $\to_{ms}$-normal). *The following two facts cannot co-exist (that is, together they imply $false$):*
1. *$t \to_{ms} u$;*
2. *t is ms-normal.*

Termination of Micro Substitutions. The proof of termination of $\to_{ms}$ is neat, as we do not need a termination measure, we simply prove it by induction on the structure of terms, via two auxiliary lemmas.

▶ **Lemma 24.** *If W is a weak head context, $W\langle t\rangle$ is ms-normal, and $x \notin \mathsf{fv}(t)$ then $W\langle t\rangle[x{\leftarrow}u]$ is ms-normal.*

Proof. By case analysis of $W\langle t\rangle$ is ms-normal. The answer and $\to_{dB}$-step case are immediate. If $\mathsf{fhv}(W\langle t\rangle) = y \neq x$ then $\mathsf{fhv}(W\langle t\rangle[x\leftarrow u]) = y$, and so $W\langle t\rangle[x\leftarrow u]$ is ms-normal. Finally, $\mathsf{fhv}(W\langle t\rangle) = x$ is impossible, because by hypothesis x does not occur in t. ◀

▶ **Lemma 25.** *If t has no ESs and W is a weak head context not capturing variables of t then $W\langle t\rangle$ is ms-normal.*

Proof. By induction on W. Cases:

- *Empty*, that is, $W = \langle\cdot\rangle$. Then $W\langle t\rangle = t$. The statement is given by the fact that terms with no ESs are ms-normal (easy omitted lemma).
- *Application*, that is $W = W'u$. The statement is given by the *i.h.* and the stability of ms-normal by application (easy omitted lemma).
- *Substitution*, that is $W = W'[x\leftarrow u]$. By *i.h.*, $W'\langle t\rangle$ is ms-normal. By hypothesis, $x \notin \mathsf{fv}(t)$. Then by Lemma 24 $W'\langle t\rangle[x\leftarrow u] = W\langle t\rangle$ is ms-normal. ◀

▶ **Proposition 26** ($\to_{ms}$ terminates)**.** *If t is reachable then $t \to_{ms}^* u$ with u ms-normal.*

Proof. By induction on t is reachable. Cases:

- *Variable* and *abstraction*: immediate since t cannot $\to_{ms}$-reduce and it is ms-normal.
- *Application*, that is $t = rw$: by *i.h.*, $r \to_{ms}^* u'$ with u' ms-normal. Then $rw \to_{ms}^* u'w$ and $u'w$ is ms-normal because being ms-normal is stable by application (omitted lemma).
- *Substitution*, that is $t = r[x\leftarrow w]$: by *i.h.*, $r \to_{ms}^* u'$ with u' ms-normal. Then $r[x\leftarrow w] \to_{ms}^* u'[x\leftarrow w]$. Case analysis of u' ms-normal:
  - If u' is an answer, a $\to_{dB}$-step, or $\mathsf{fhv}(u') = y \neq x$ then so is for $u'[x\leftarrow w]$.
  - $\mathsf{fhv}(u') = x$. Then there is a weak head context W such that $\mathsf{maxw}(u') = W$. Then $u'[x\leftarrow u] \to_{ms} W\langle u\rangle[x\leftarrow u]$. Since W does not capture variables of u, by Lemma 25 we have that $W\langle u\rangle$ is ms-normal. Since $x \notin \mathsf{fv}(u)$, by Lemma 24 $W\langle u\rangle[x\leftarrow u]$ is ms-normal. The statement holds because $r[x\leftarrow w] \to_{ms}^* u'[x\leftarrow w] \to_{ms} W\langle u\rangle[x\leftarrow u]$. ◀

Reflection of $\to_{wh}$ Steps. We can finally prove the one-step reflection property. It rests on the auxiliary case of reflection on ms-normal terms, which is given here without proof.

▶ **Proposition 27** (ms-normal terms reflect $\to_{wh}$ steps as $\to_{dB}$ steps)**.** *If t is reachable, $t{\downarrow} \to_{wh} u$ and t is ms-normal then exists r such that $t \to_{dB} r$.*

▶ **Proposition 28** (Reflection of $\to_{wh}$ steps)**.** *If t is reachable, $t{\downarrow} \to_{wh} u$ then exists r and w such that $t \to_{ms}^* r \to_{dB} w$ and $w{\downarrow} = u$.*

Proof. By termination of $\to_{ms}$ (Prop. 26), there exists r such that $t \to_{ms}^* r$ and r is ms-normal. By $\to_{ms}$-projection (Prop. 20.2), $t{\downarrow} = r{\downarrow}$. We can then apply Prop. 27, obtaining that there exists w such that $r \to_{dB} w$. Since r is reachable, by $\to_{dB}$-projection (Prop. 18) $r{\downarrow} \to_{wh} w{\downarrow}$. Since $\to_{wh}$ is deterministic (Lemma 1), $u = w{\downarrow}$. ◀

Summing Up. We then conclude with our second main result. In the Abella sources, point 1 is `micro_to_small_simulation` and point 2 is `small_to_micro_simulation_no_ES`.

▶ **Theorem 29.** *Let t be reachable.*

1. Micro to small steps*: if $t \to_{mwh}^* u$ then $t{\downarrow} \to_{wh}^* u{\downarrow}$.*
2. Small to micro steps*: if t has no ES and $t \to_{wh}^* u$ then there exists r such that $t \to_{mwh}^* r$ and $r{\downarrow} = u$.*

Proof. Point 1 is an easy induction on the length of $t \to^*_{mwh} u$, using the projection properties (Prop. 18 and Prop. 20). For point 2, one proves that if t is reachable and $t{\downarrow} \to^*_{wh} u$ then there exists r such that $t \to^*_{wh} r$ and $r{\downarrow} = u$, by induction on the length of $t{\downarrow} \to^*_{wh} u$ using the reflection property (Prop. 28). The statement follows from the fact that terms without ES are reachable and satisfy $t{\downarrow} = t$. ◀

Putting it All Together? At this point, it is natural to expect that our two main results – namely the relationships LSC/π (Theorem 17) and λ/LSC (Theorem 29) – can be composed, obtaining a final theorem relating λ and π. This is however not possible, because the key concept for connecting λ and the LSC is the unfolding $t{\downarrow}$ of ESs, which has no analogous on processes. It is exactly such a difficulty that, in presentations without ESs, forces the use of strong bisimulation on processes to close the simulation diagram between λ and π.

At first sight, turning an ES $t[x{\leftarrow}u]$ into a meta-level substitution $t\{x{\leftarrow}u\}$, which is the operation iterated by the unfolding, corresponds on processes to a *broadcast*, that is, to send $[\![u]\!]$ to all sub-processes of $[\![t[x{\leftarrow}u]]\!]_a$ that can perform an input on channel x, but this is misleading because occurrences of x actually correspond to *outputs*, not inputs, and the sub-process encoding $[x{\leftarrow}u]$ is an *input*, not an output.

8 Conclusions

We provide the first formalization of Milner's translation of λ to π, by actually formalizing Accattoli's factorization of the translation via the linear substitution calculus. The difficulties with names and binding are circumvented thanks to the features of the Abella proof assistant, and by defining terms and processes over the same variables.

About future work, it would be interesting to extend our formalization to Sangiorgi's result relating barbed congruence in the π-calculus with his normal form bisimulation in the λ-calculus [41]. It would also be interesting to see how to exploit the new presentation of the rewriting rules of the π-calculus for formalizing other results of its theory.

References

1 Beniamino Accattoli. Proof pearl: Abella formalization of λ-calculus cube property. In Chris Hawblitzel and Dale Miller, editors, *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, volume 7679 of *Lecture Notes in Computer Science*, pages 173–187. Springer, 2012. doi:10.1007/978-3-642-35308-6_15.

2 Beniamino Accattoli. Evaluating functions as processes. In Rachid Echahed and Detlef Plump, editors, *Proceedings 7th International Workshop on Computing with Terms and Graphs, TERMGRAPH 2013, Rome, Italy, 23th March 2013*, volume 110 of *EPTCS*, pages 41–55, 2013. doi:10.4204/EPTCS.110.6.

3 Beniamino Accattoli. Proof nets and the linear substitution calculus. In Bernd Fischer and Tarmo Uustalu, editors, *Theoretical Aspects of Computing - ICTAC 2018 - 15th International Colloquium, Stellenbosch, South Africa, October 16-19, 2018, Proceedings*, volume 11187 of *Lecture Notes in Computer Science*, pages 37–61. Springer, 2018. doi:10.1007/978-3-030-02508-3_3.

4 Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. Distilling abstract machines. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 363–376. ACM, 2014. doi:10.1145/2628136.2628154.

5 Beniamino Accattoli, Eduardo Bonelli, Delia Kesner, and Carlos Lombardi. A nonstandard standardization theorem. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 659–670. ACM, 2014. doi:10.1145/2535838.2535886.

6 Beniamino Accattoli and Ugo Dal Lago. On the invariance of the unitary cost model for head reduction. In Ashish Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA'12) , RTA 2012, May 28 - June 2, 2012, Nagoya, Japan*, volume 15 of *LIPIcs*, pages 22–37. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPIcs.RTA.2012.22.

7 Beniamino Accattoli and Ugo Dal Lago. (leftmost-outermost) beta reduction is invariant, indeed. *Log. Methods Comput. Sci.*, 12(1), 2016. doi:10.2168/LMCS-12(1:4)2016.

8 Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. The machinery of interaction. In *PPDP '20: 22nd International Symposium on Principles and Practice of Declarative Programming, Bologna, Italy, 9-10 September, 2020*, pages 4:1–4:15. ACM, 2020. doi:10.1145/3414080.3414108.

9 Beniamino Accattoli, Stéphane Graham-Lengrand, and Delia Kesner. Tight typings and split bounds, fully developed. *J. Funct. Program.*, 30:e14, 2020. doi:10.1017/S095679682000012X.

10 Guillaume Ambal, Sergueï Lenglet, and Alan Schmitt. Hoπ in coq. *J. Autom. Reason.*, 65(1):75–124, 2021. doi:10.1007/s10817-020-09553-0.

11 David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2):1–89, 2014. doi:10.6092/issn.1972-5787/4650.

12 Jesper Bengtson and Joachim Parrow. Formalising the pi-calculus using nominal logic. *Log. Methods Comput. Sci.*, 5(2), 2009. URL: http://arxiv.org/abs/0809.3960.

13 Malgorzata Biernacka, Dariusz Biernacki, Sergueï Lenglet, Piotr Polesiuk, Damien Pous, and Alan Schmitt. Fully abstract encodings of λ-calculus in hocore through abstract machines. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12. IEEE Computer Society, 2017. doi:10.1109/LICS.2017.8005118.

14 Gérard Boudol. The p-calculus in direct style. *High. Order Symb. Comput.*, 11(2):177–208, 1998. doi:10.1023/A:1010064516533.

15 Gérard Boudol and Cosimo Laneve. The discriminating power of multiplicities in the lambda-calculus. *Inf. Comput.*, 126(1):83–102, 1996. doi:10.1006/inco.1996.0037.

16 Xiaojuan Cai and Yuxi Fu. The λ-calculus in the π-calculus. *Math. Struct. Comput. Sci.*, 21(5):943–996, 2011. doi:10.1017/S0960129511000260.

17 Kaustuv Chaudhuri, Matteo Cimini, and Dale Miller. A lightweight formalization of the metatheory of bisimulation-up-to. In Xavier Leroy and Alwen Tiu, editors, *Proceedings of the 4th ACM-SIGPLAN Conference on Certified Programs and Proofs*, pages 157–166, Mumbai, India, January 2015. ACM. doi:10.1145/2676724.2693170.

18 Matteo Cimini, Claudio Sacerdoti Coen, and Davide Sangiorgi. Functions as processes: Termination and the $\lambda\mu\tilde{\mu}$-calculus. In Martin Wirsing, Martin Hofmann, and Axel Rauschmayer, editors, *Trustworthly Global Computing - 5th International Symposium, TGC 2010, Munich, Germany, February 24-26, 2010, Revised Selected Papers*, volume 6084 of *Lecture Notes in Computer Science*, pages 73–86. Springer, 2010. doi:10.1007/978-3-642-15640-3_5.

19 Joëlle Despeyroux. A higher-order specification of the pi-calculus. In *Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics, International Conference IFIP TCS 2000, Sendai, Japan, August 17-19, 2000, Proceedings*, volume 1872 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 2000. doi:10.1007/3-540-44929-9_30.

20 Adrien Durier, Daniel Hirschkoff, and Davide Sangiorgi. Eager functions as processes. *Theor. Comput. Sci.*, 913:8–42, 2022. doi:10.1016/j.tcs.2022.01.043.

21 Murdoch J. Gabbay. *The π-Calculus in FM*, pages 247–269. Springer Netherlands, Dordrecht, 2003. doi:10.1007/978-94-017-0253-9_10.

22 Andrew Gacek. The abella interactive theorem prover (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 154–161. Springer, 2008. doi:10.1007/978-3-540-71070-7_13.

23 Andrew Gacek, Dale Miller, and Gopalan Nadathur. Nominal abstraction. *Inf. Comput.*, 209(1):48–73, 2011. doi:10.1016/j.ic.2010.09.004.

24 Simon J. Gay. A framework for the formalisation of pi calculus type systems in isabelle/hol. In *Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs 2001, Edinburgh, Scotland, UK, September 3-6, 2001, Proceedings*, volume 2152 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 2001. doi:10.1007/3-540-44755-5_16.

25 Daniel Hirschkoff. A full formalisation of pi-calculus theory in the calculus of constructions. In Elsa L. Gunter and Amy P. Felty, editors, *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97, Murray Hill, NJ, USA, August 19-22, 1997, Proceedings*, volume 1275 of *Lecture Notes in Computer Science*, pages 153–169. Springer, 1997. doi:10.1007/BFb0028392.

26 Furio Honsell, Marino Miculan, and Ivan Scagnetto. π-calculus in (co)inductive type theories. *Theoretical Computer Science*, 2(253):239–285, 2001. doi:10.1016/S0304-3975(00)00095-5.

27 Guilhem Jaber and Davide Sangiorgi. Games, mobile processes, and functions. In Florin Manea and Alex Simpson, editors, *30th EACSL Annual Conference on Computer Science Logic, CSL 2022, February 14-19, 2022, Göttingen, Germany (Virtual Conference)*, volume 216 of *LIPIcs*, pages 25:1–25:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.CSL.2022.25.

28 Naoki Kobayashi. A partially deadlock-free typed process calculus. *ACM Trans. Program. Lang. Syst.*, 20(2):436–482, 1998. doi:10.1145/276393.278524.

29 Thomas F. Melham. A mechanized theory of the pi-calculus in HOL. *Nord. J. Comput.*, 1(1):50–76, 1994.

30 Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012. URL: http://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/programming-higher-order-logic?format=HB.

31 Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Trans. Comput. Log.*, 6(4):749–783, 2005. doi:10.1145/1094622.1094628.

32 Robin Milner. Functions as processes. *Math. Struct. Comput. Sci.*, 2(2):119–141, 1992. doi:10.1017/S0960129500001407.

33 Robin Milner. Local bigraphs and confluence: Two conjectures: (extended abstract). In Roberto M. Amadio and Iain Phillips, editors, *Proceedings of the 13th International Workshop on Expressiveness in Concurrency, EXPRESS 2006, Bonn, Germany, August 26, 2006*, volume 175 of *Electronic Notes in Theoretical Computer Science*, pages 65–73. Elsevier, 2006. doi:10.1016/j.entcs.2006.07.035.

34 Otmane Aït Mohamed. Mechanizing a pi-calculus equivalence in HOL. In *Higher Order Logic Theorem Proving and Its Applications, 8th International Workshop, Aspen Grove, UT, USA, September 11-14, 1995, Proceedings*, volume 971 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1995. doi:10.1007/3-540-60275-5_53.

35 Joachim Niehren. Functional computation as concurrent computation. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 333–343. ACM Press, 1996. doi:10.1145/237721.237801.

36 Dominic A. Orchard and Nobuko Yoshida. Using session types as an effect system. In *Proceedings Eighth International Workshop on Programming Language Approaches to Concurrency-*

and Communication-cEntric Software, PLACES 2015, London, UK, 18th April 2015, volume 203 of *EPTCS*, pages 1–13, 2015. doi:10.4204/EPTCS.203.1.

37 Roly Perera and James Cheney. Proof-relevant π-calculus: a constructive account of concurrency and causality. *Math. Struct. Comput. Sci.*, 28(9):1541–1577, 2018. doi:10.1017/S096012951700010X.

38 Christine Röckl, Daniel Hirschkoff, and Stefan Berghofer. Higher-order abstract syntax with induction in isabelle/hol: Formalizing the pi-calculus and mechanizing the theory of contexts. In *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2030 of *Lecture Notes in Computer Science*, pages 364–378. Springer, 2001. doi:10.1007/3-540-45315-6_24.

39 Davide Sangiorgi. The lazy lambda calculus in a concurrency scenario. *Inf. Comput.*, 111(1):120–153, 1994. doi:10.1006/inco.1994.1042.

40 Davide Sangiorgi. From lambda to pi; or, rediscovering continuations. *Math. Struct. Comput. Sci.*, 9(4):367–401, 1999. URL: http://journals.cambridge.org/action/displayAbstract?aid=44843.

41 Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001.

42 Davide Sangiorgi and Xian Xu. Trees from functions as processes. *Log. Methods Comput. Sci.*, 14(3), 2018. doi:10.23638/LMCS-14(3:11)2018.

43 Alwen Tiu and Dale Miller. Proof search specifications of bisimulation and modal logics for the pi-calculus. *ACM Trans. Comput. Log.*, 11(2):13:1–13:35, 2010. doi:10.1145/1656242.1656248.

44 Bernardo Toninho, Luís Caires, and Frank Pfenning. Functions as session-typed processes. In Lars Birkedal, editor, *Foundations of Software Science and Computational Structures - 15th International Conference, FOSSACS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7213 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 2012. doi:10.1007/978-3-642-28729-9_23.

45 Niccolò Veltri and Andrea Vezzosi. Formalizing π-calculus in guarded cubical agda. 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 270–283. ACM, 2020. doi:10.1145/3372885.3373814.

An Elementary Formal Proof of the Group Law on Weierstrass Elliptic Curves in Any Characteristic

David Kurniadi Angdinata
London School of Geometry and Number Theory, UK

Junyan Xu
Cancer Data Science Laboratory, National Cancer Institute, Bethesda, MD, USA

Abstract

Elliptic curves are fundamental objects in number theory and algebraic geometry, whose points over a field form an abelian group under a geometric addition law. Any elliptic curve over a field admits a Weierstrass model, but prior formal proofs that the addition law is associative in this model involve either advanced algebraic geometry or tedious computation, especially in characteristic two. We formalise in the Lean theorem prover, the type of nonsingular points of a Weierstrass curve over a field of any characteristic and a purely algebraic proof that it forms an abelian group.

2012 ACM Subject Classification Theory of computation → Interactive proof systems; Security and privacy → Logic and verification; Mathematics of computing → Mathematical software

Keywords and phrases formal math, algebraic geometry, elliptic curve, group law, Lean, mathlib

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.6

Supplementary Material *Software (Source Code)*: `https://github.com/alreadydone/mathlib/tree/associativity`, archived at `swh:1:dir:1bd75e80371560806d5f287177a4922b6282f7e2`

Funding This work was supported by the Engineering and Physical Sciences Research Council [EP/S021590/1], EPSRC Centre for Doctoral Training in Geometry and Number Theory (London School of Geometry and Number Theory), University College London. This research was supported in part by the Intramural Research Program of the Center for Cancer Research, National Cancer Institute, NIH.

Acknowledgements We thank the Lean community for their continual support. We thank the `mathlib` contributors, especially Anne Baanen, for developing libraries this work depends on. We thank Marc Masdeu and Michael Stoll for proposing alternative proofs. DKA would like to thank Kevin Buzzard for his guidance and Mel Levin for suggesting the formalisation in the first place.

1 Introduction

1.1 Elliptic curves

In its earliest form, algebraic geometry is the branch of mathematics studying the solutions to systems of polynomial equations over a base field F, namely sets of the form

$$\{(x_1, \dots, x_n) \in F^n : f_1(x_1, \dots, x_n) = 0, \ \dots, \ f_k(x_1, \dots, x_n) = 0\},$$

for some polynomials $f_i \in F[X_1, \dots, X_n]$. These are called **affine varieties**, and they can be endowed with topologies and a notion of morphisms which makes them simultaneously geometric objects. General **varieties** are locally modelled on affine ones, with morphisms between them locally given by polynomials, and are often classified by their geometric properties such as *smoothness*, or invariants such as the *dimension* or the *genus*.

Having dimension one and genus one, *elliptic curves* are amongst the simplest varieties with respect to these geometric notions, and their set of points can be endowed with the structure of an abelian group. When the base field is the rationals $\mathbb{Q}$, a common definition

© David Kurniadi Angdinata and Junyan Xu;
licensed under Creative Commons License CC-BY 4.0

uses the *short Weierstrass model*, given by the equation $y^2 = x^3 + ax + b$ for some fixed $a, b \in \mathbb{Q}$, and its group law can be defined explicitly by quotients of polynomial functions.

Elliptic curves are blessed with an extremely rich theory, spanning the fields of algebraic geometry, complex analysis, number theory, representation theory, dynamical systems, and even information security. The *Birch and Swinnerton-Dyer conjecture* [29] in number theory, one of the seven *Millennium Prize Problems*, is an equality between an analytic quantity of an elliptic curve over $\mathbb{Q}$ and an algebraic quantity defined in terms of its group structure. Their close relation with modular forms is precisely the content of the *Taniyama–Shimura conjecture* proven by Andrew Wiles [30], which implies *Fermat's last theorem* and laid the foundations of the *Langlands programme*, a web of influential conjectures linking number theory and geometry described to be "kind of a grand unified theory of mathematics" [19]. Outside of mathematics, elliptic curves over finite fields see applications in primality proving [2] and integer factorisation [22], as well as in public key cryptography [13].

1.2 Formalisation attempts

The group law on an elliptic curve in the short Weierstrass model over a field F has been formalised in several theorem provers,[1] but this model fails to be an elliptic curve when $\mathrm{char}(F) = 2$, and there has been no known successful attempts to formalise the group law in a *universal* model that captures all elliptic curves in all $\mathrm{char}(F)$. The issue was that a computational proof of associativity in any universal model is, as Russinoff described, "an elementary but computationally intensive arithmetic exercise" involving massive polynomials [27],[2] while a typical conceptual proof is "a deep theorem of algebraic or projective geometry" requiring prior formalisation of advanced geometric constructions, with "evidence that an elementary hand proof of this result is a practical impossibility" [25]. On the other hand, having the group law in $\mathrm{char}(F) = 2$ is necessary for certain applications, such as the proof of Fermat's last theorem. It is less crucial from an information security viewpoint, as curves over binary fields are prone to attacks and no longer recommended by NIST [9].

We give a conceptual yet computation-light proof of the group law in the full *Weierstrass model*, which is universal in all $\mathrm{char}(F)$. The argument is purely algebraic and easily surveyable, in the sense that all logical deductions and necessary computations can be performed by hand in a matter of minutes. The proof is formalised in *Lean 3* [12], an interactive theorem prover based on the calculus of constructions with inductive types, and is integrated as part of its monolithic mathematical library `mathlib` [11]. The implementation extensively uses existing constructions in the linear algebra and ring theory libraries of `mathlib`, particularly constructions and results surrounding `algebra.norm` and `class_group` [10]. The relevant code is primarily split into two files `weierstrass.lean` and `point.lean` under the folder `algebraic_geometry/elliptic_curve` in the `associativity` branch of `mathlib`, both of which are currently undergoing reviews for their merge. With this paper, we hope that our simple proof will be replicated and will open the way for the formalisation of elliptic curve cryptography in many other theorem provers, which has been a major motivation of recent formalisation attempts [25, 15, 18, 4].

The remainder of this paper will describe the relevant constructions (Section 2), detail the mathematical argument of the proof (Section 3), and discuss implementation considerations (Section 4). Throughout, definitions will be described in code snippets where relevant, but proofs of lemmas will be outlined mathematically for the sake of clarity. The reader may refer to the `mathlib` documentation [10] for definitions and lemmas involved.

[1] see Section 4.1 for related work
[2] see Section 4.2 for experimental results

2 Weierstrass equations

Let F be a field. In the sense of modern algebraic geometry, an **elliptic curve** E over F is a smooth projective curve[3] of genus one equipped with a base point $\mathcal{O}_E \in E$ with coordinates in F. More concretely, any elliptic curve over F admits a model in the projective plane $\mathbb{P}^2_F$ defined by an explicit polynomial equation in homogeneous coordinates $[X : Y : Z]$.

▶ **Proposition 1.** *If E is an elliptic curve over F, then there are rational functions $x, y : E \to F$ such that the map $\phi : E \to \mathbb{P}^2_F$ given by $\phi(P) = [x(P) : y(P) : 1]$ maps $\mathcal{O}_E$ to $[0 : 1 : 0]$, and defines an isomorphism of varieties between E and the curve*

$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3,$$

for some coefficients $a_i \in F$. Conversely, any curve W in $\mathbb{P}^2_F$ given by such an equation, with coefficients $a_i \in F$, is an elliptic curve over F with base point $[0 : 1 : 0]$ if the quantity

$$\begin{aligned}\Delta_W := &-(a_1^2 + 4a_2)^2(a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2) - 8(2a_4 + a_1a_3)^3\\ &- 27(a_3^2 + 4a_6)^2 + 9(a_1^2 + 4a_2)(2a_4 + a_1a_3)(a_3^2 + 4a_6) \in F\end{aligned}$$

is nonzero.

Proof. This is a consequence of the Riemann–Roch theorem [26, Proposition III.3.1]. ◀

This is the **Weierstrass model** of E, and such a curve is called a **Weierstrass curve**, whose corresponding **Weierstrass equation** in the affine chart $Z \neq 0$ is

$$Y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a_4X + a_6.$$

In this model, the smoothness condition on E becomes equivalent to the **discriminant** $\Delta_E \in F$ being nonzero, so an elliptic curve over F may instead be defined as a Weierstrass curve with the discriminant condition, which is more amenable for computational purposes.

2.1 Weierstrass curves

Let F be a commutative ring, and let W be a Weierstrass curve over F. Explicitly, this is merely the data of five coefficients $a_1, a_2, a_3, a_4, a_6 \in F$, noting that the Weierstrass equation is not visible at this stage. For the sake of generality, the structure `weierstrass_curve` is defined more generally over an arbitrary type F, but all subsequent constructions will assume that F is at least a commutative ring. The structure `elliptic_curve` then `extends` `weierstrass_curve` by adding the data of an element Δ' in the group of units $F^\times$ of F and a proof that $\Delta' = \Delta_W$, so most definitions for `weierstrass_curve` carry over automatically.

```
structure weierstrass_curve (F : Type) := (a₁ a₂ a₃ a₄ a₆ : F)

structure elliptic_curve (F : Type) [comm_ring F] extends weierstrass_curve F :=
  (Δ' : Fˣ) (coe_Δ' : ↑Δ' = to_weierstrass_curve.Δ)
```

Here, `to_weierstrass_curve` is a function generated automatically by the `extends` keyword, which projects an `elliptic_curve` down to its underlying `weierstrass_curve` by forgetting Δ' and the proof that $\Delta' = \Delta_W$. Note that `elliptic_curve` was originally defined in one stretch by Buzzard, but is now refactored through the more general `weierstrass_curve`.

[3] variety of dimension one

▶ Remark 2. This definition of an elliptic curve is universal over a large class of commutative rings, namely those with trivial Picard group [21, Section 2.2], which includes fields, and also local rings and unique factorisation domains. In general, an elliptic curve E may be defined relative to an arbitrary scheme S as a smooth proper morphism $E \to S$ in the category of schemes whose geometric fibres are all integral curves of genus one, equipped with a section $S \to E$ that plays the role of the base point $\mathcal{O}_E$ for all fibres simultaneously. When S is the spectrum of such a commutative ring, the Riemann–Roch theorem can be generalised so that E remains isomorphic to a Weierstrass curve, but over a general commutative ring, E only has Weierstrass equations locally that may not patch together to form a global equation.

The discriminant $\Delta_W \in F$ is expressed entirely in terms of the five coefficients, but it is clearer to extract intermediate quantities [26, Section III.1] to simplify the large expression.

```
namespace weierstrass_curve

variables {F : Type} [comm_ring F] (W : weierstrass_curve F)

def b₂ : F := W.a₁^2 + 4*W.a₂
def b₄ : F := 2*W.a₄ + W.a₁*W.a₃
def b₆ : F := W.a₃^2 + 4*W.a₆
def b₈ : F := W.a₁^2*W.a₆ + 4*W.a₂*W.a₆ - W.a₁*W.a₃*W.a₄ + W.a₂*W.a₃^2 - W.a₄^2

def Δ : F := -W.b₂^2*W.b₈ - 8*W.b₄^3 - 27*W.b₆^2 + 9*W.b₂*W.b₄*W.b₆
```

Here, dot notation allows for the fields corresponding to the five coefficients $a_i \in F$ to be accessed as `W.a`$_i$, and living inside the `namespace weierstrass_curve` means that the quantities $b_i \in F$ and $\Delta \in F$ may also be accessed as `W.b`$_i$ and `W.`Δ respectively.

These quantities are indexed as such as a result of their transformation upon applying the linear change of variables $(X, Y) \mapsto (u^2X + r, u^3Y + u^2sX + t)$, for some $u \in F^\times$ and some $r, s, t \in F$. In fact, these are all the possible isomorphisms of varieties between elliptic curves in the Weierstrass model [26, Proposition III.3.1].

```
variables (u : Fˣ) (r s t : F)

@[simps] def variable_change : weierstrass_curve F :=
  { a₁ := ↑u⁻¹*(W.a₁ + 2*s),
    a₂ := ↑u⁻¹^2*(W.a₂ - s*W.a₁ + 3*r - s^2),
    a₃ := ↑u⁻¹^3*(W.a₃ + r*W.a₁ + 2*t),
    a₄ := ↑u⁻¹^4*(W.a₄ - s*W.a₃ + 2*r*W.a₂ - (t + r*s)*W.a₁ + 3*r^2 - 2*s*t),
    a₆ := ↑u⁻¹^6*(W.a₆ + r*W.a₄ + r^2*W.a₂ + r^3 - t*W.a₃ - t^2 - r*t*W.a₁) }

@[simp] lemma variable_change_b₂ : (W.variable_change u r s t).b₂ = ↑u⁻¹^2*(...)
@[simp] lemma variable_change_b₄ : (W.variable_change u r s t).b₄ = ↑u⁻¹^4*(...)
@[simp] lemma variable_change_b₆ : (W.variable_change u r s t).b₆ = ↑u⁻¹^6*(...)
@[simp] lemma variable_change_b₈ : (W.variable_change u r s t).b₈ = ↑u⁻¹^8*(...)

@[simp] lemma variable_change_Δ : (W.variable_change u r s t).Δ = ↑u⁻¹^12*W.Δ
```

Here, `variable_change` defines a new `weierstrass_curve` under the change of variables by explicitly stating how each of the five coefficients are transformed, and is tagged with `simps` to automatically generate `simp` lemmas corresponding to each of the five projections. The exact expressions for the transformed quantities are not too important, but note that their indices precisely correspond to the exponent of $u^{-1} \in F^\times$ in the transformation.

2.2 Equations and nonsingularity

Now consider the polynomial in $F[X, Y]$ associated to W denoted by

$$W(X, Y) := Y^2 + (a_1X + a_3)Y - (X^3 + a_2X^2 + a_4X + a_6),$$

so that the Weierstrass equation literally reads $W(X, Y) = 0$, and its partial derivatives

$$W_X(X, Y) := a_1Y - (3X^2 + 2a_2X + a_4), \qquad W_Y(X, Y) := 2Y + a_1X + a_3.$$

When they do not simultaneously vanish when evaluated at an affine point $(x, y) \in W$, the affine point is said to be **nonsingular**. This is implemented slightly confusingly as follows.[4]

```
def polynomial : F[X][Y] :=
  Y^2 + C (C W.a₁*X + C W.a₃)*Y - C (X^3 + C W.a₂*X^2 + C W.a₄*X + C W.a₆)
def equation (x y : F) : Prop := (W.polynomial.eval (C y)).eval x = 0

def polynomial_X : F[X][Y] := C (C W.a₁)*Y - C (C 3*X^2 + C (2*W.a₂)*X + C W.a₄)
def polynomial_Y : F[X][Y] := C (C 2)*Y + C (C W.a₁*X + C W.a₃)
def nonsingular (x y : F) : Prop :=
  W.equation x y ∧ ((W.polynomial_X.eval (C y)).eval x ≠ 0
                  ∨ (W.polynomial_Y.eval (C y)).eval x ≠ 0)
```

Here, `F[X][Y]` denotes the polynomial ring over the polynomial ring over `F`, to simulate the bivariate polynomial ring $F[X, Y]$ over F. The outer variable Y is denoted by the symbol `Y` and the inner variable X is denoted by the constant function `C` applied to the symbol `X`, while actual constants are enclosed in two layers of the constant function `C`.

▶ Remark 3. This definition of nonsingularity in terms of partial derivatives is valid and convenient when the base ring is a field, but over a general commutative ring the same notion should be characterised locally with a notion of smoothness relative to the base spectrum.

Many properties of Weierstrass curves remain invariant under the aforementioned changes of variables, and it is often easier to prove results for Weierstrass equations with fewer terms. For instance, if $\mathrm{char}(F) \neq 2$, then $(X, Y) \mapsto (X, Y - \frac{1}{2}a_1X - \frac{1}{2}a_3)$ completes the square for $W(X, Y)$ and eliminates the XY and Y terms, and if further $\mathrm{char}(F) \neq 3$, then $(X, Y) \mapsto (X - \frac{1}{12}b_2, Y)$ completes the cube for $W(X, Y)$ and eliminates the X^2 term as well.

Perhaps a more prominent application is the proof that, for any affine point $(x, y) \in W$, the nonvanishing of Δ_W implies that (x, y) is nonsingular. This statement is easy for $(x, y) = (0, 0)$, since $(0, 0) \in W$ implies that $a_6 = 0$, and $(0, 0)$ being singular implies that $a_3 = a_4 = 0$, so that $\Delta_W = 0$ by a simple substitution. On the other hand, for any $(x, y) \in F^2$, the change of variables $(X, Y) \mapsto (X - x, Y - y)$ merely translates W so that $(0, 0)$ gets mapped to (x, y), so the same statement clearly also holds for (x, y).

```
lemma nonsingular_zero : W.nonsingular 0 0 ↔ W.a₆ = 0 ∧ (W.a₃ ≠ 0 ∨ W.a₄ ≠ 0)
lemma nonsingular_zero_of_Δ_ne_zero (h : W.equation 0 0) (hΔ : W.Δ ≠ 0) :
  W.nonsingular 0 0
lemma nonsingular_iff_variable_change (x y : F) :
  W.nonsingular x y ↔ (W.variable_change 1 x 0 y).nonsingular 0 0
lemma nonsingular_of_Δ_ne_zero {x y : F} (h : W.equation x y) (hΔ : W.Δ ≠ 0) :
  W.nonsingular x y
```

In fact, it is also true that $\Delta_W \neq 0$ if every point over the algebraic closure is nonsingular [26, Proposition III.1.4], but the proof is more difficult and has yet to be formalised.

[4] this representation of bivariate polynomials will be explained in Section 4.3

2.3 Point addition

The set of points on an elliptic curve can be endowed with an **addition law** defined by a geometric secant-and-tangent process enabled by Vieta's formulae.[5] This can be easily described in the Weierstrass model, where a point on W is either of the form (x, y) in the affine chart $Z \neq 0$ and satisfies the Weierstrass equation, or is the unique point at infinity $\mathcal{O}_W := [0 : 1 : 0]$ when $Z = 0$. If $S \in W$ is a singular point, the same geometric process will yield $P + S = S = S + P$ for any other point $P \in W$, so it necessitates considering only nonsingular points on W to obtain a group [26, Section III.2]. Note that if W is an elliptic curve, then all points are nonsingular by `nonsingular_of_Δ_ne_zero`.

```
inductive point
  | zero
  | some {x y : F} (h : W.nonsingular x y)

namespace point
```

Here, `zero` refers to $\mathcal{O}_W$ and `some` refers to an affine point on W. Note that a proof that an affine point $(x, y) \in W$ is nonsingular already subsumes the data of the coordinates $(x, y) \in F^2$ in its type, so such a point is constructed by giving only the proof.

▶ Remark 4. The set of points defined here will later be shown to form an abelian group under this addition law, but the presence of division means that the base ring needs to be a field. Over a general commutative ring R these should be replaced with scheme-theoretic points $\mathrm{Spec}(R) \to E$, and the elliptic curve acquires the structure of a group scheme.

In this model, the identity element in the group of points is defined to be $\mathcal{O}_W \in W$.

```
instance : has_zero W.point := ⟨zero⟩
```

Here, the `instance` of `has_zero` allows the notation `0` instead of `zero`.

Given a nonsingular point $P \in W$, its negation $-P$ is defined to be the unique third point of intersection between W and the line through $\mathcal{O}_W$ and P, which is vertical when drawn on the affine plane. Explicitly, if $P := (x, y)$, then $-P := (x, \sigma_x(y))$, where

$$\sigma_X(Y) := -Y - (a_1X + a_3)$$

is the **negation polynomial**, which gives a very useful involution of the affine plane.

```
def neg_polynomial : F[X][Y] := -Y - C (C W.a₁*X + C W.a₃)
def neg_Y (x y : F) : F := (W.neg_polynomial.eval (C y)).eval x
```

Here, `neg_Y` is defined in terms of `neg_polynomial` for clarity, but its actual definition is written out as `-y - C (C W.a₁*x + C W.a₃)`. This is merely to avoid requiring the `noncomputable` tag, since polynomial operations are currently `noncomputable` in `mathlib`.

To define negation, it remains to prove that the negation of a nonsingular point on W is again a nonsingular point on W, but this is straightforward.

▶ **Lemma 5.** *If $(x, y) \in W$ is nonsingular, then $(x, \sigma_x(y)) \in W$ is nonsingular.*

Proof. Since $(x, y) \in W$, verifying that $W(x, y) = W(x, \sigma_x(y))$ gives $(x, \sigma_x(y)) \in W$ as well. Now assume that $W_Y(x, \sigma_x(y)) = 0$. It can be checked that $y = \sigma_x(y)$, so $W_Y(x, y) = 0$ as well. Since (x, y) is nonsingular, $W_X(x, y) \neq 0$, so $W_X(x, \sigma_x(y)) \neq 0$ as well. ◀

[5] if a cubic polynomial has two roots in a field, then its third root is also in the field

Lemma 5 is `nonsingular_neg`, which maps a proof that $(x, y) \in W$ is nonsingular to a proof that $-(x, y) \in W$ is nonsingular. This leads to the definition of negation.

```
def neg : W.point → W.point
  | 0         := 0
  | (some h) := some (nonsingular_neg h)

instance : has_neg W.point := ⟨neg⟩
```

Here, the `instance` of `has_neg` allows the notation `-P` instead of `neg P`.

Given two nonsingular points $P_1, P_2 \in W$, their sum $P_1 + P_2$ is defined to be the negation of the unique third point of intersection between W and the line through P_1 and P_2, which again exists by Bézout's theorem. Explicitly, let $P_1 := (x_1, y_1)$ and $P_2 := (x_2, y_2)$.

- If $x_1 = x_2$ and $y_1 = \sigma_{x_2}(y_2)$, then this line is vertical and $P_1 + P_2 := \mathcal{O}_W$.
- If $x_1 = x_2$ and $y_1 \neq \sigma_{x_2}(y_2)$, then this line is the tangent of W at $P_1 = P_2$, and has slope

$$\ell := \frac{-W_X(x_1, y_1)}{W_Y(x_1, y_1)}.$$

- Otherwise $x_1 \neq x_2$, then this line is the secant of W through P_1 and P_2, and has slope

$$\ell := \frac{y_1 - y_2}{x_1 - x_2}.$$

In the latter two cases, the **line polynomial**

$$\lambda(X) = \lambda_{x_1, y_1, \ell}(X) := \ell(X - x_1) + y_1.$$

can be shown to satisfy $\lambda(x_1) = y_1$ and $\lambda(x_2) = y_2$, so that x_1 and x_2 are two roots of the **addition polynomial** $W(X, \lambda(X))$, obtained by evaluating $W(X, Y)$ at $\lambda(X)$, where $W(X, Y)$ is viewed as a polynomial over $F[X]$. In an attempt to reduce code duplication for the different cases, these accept an additional parameter `L` for the slope ℓ.

```
def line_polynomial (x y L : F) : F[X] := C L * (X - C x) + C y
def add_polynomial (x y L : F) : F[X] := W.polynomial.eval (line_polynomial x y L)
```

The X-coordinate of $P_1 + P_2$ is then the third root of $W(X, \lambda(X))$, so that

$$W(X, \lambda(X)) = -(X - x_1)(X - x_2)(X - x_3). \tag{1}$$

By inspecting the X^2 terms of $W(X, \lambda(X))$, this third root can be shown to be

$$x_3 := \ell^2 + a_1\ell - a_2 - x_1 - x_2,$$

so the Y-coordinate of $-(P_1 + P_2)$ is

$$y_3' := \lambda(x_3),$$

and that of $P_1 + P_2$ is

$$y_3 := \sigma_{x_3}(y_3').$$

These correspond to the coordinate functions `add_X`, `add_Y'`, and `add_Y` respectively.

```
def add_X (x₁ x₂ L : F) : F := L^2 + W.a₁*L - W.a₂ - x₁ - x₂
def add_Y' (x₁ x₂ y₁ L : F) : F := (line_polynomial x₁ y₁ L).eval (W.add_X x₁ x₂ L)
def add_Y (x₁ x₂ y₁ L : F) : F := W.neg_Y (W.add_X x₁ x₂ L) (W.add_Y' x₁ x₂ y₁ L)
```

Here, `add_Y'` is defined in terms of `line_polynomial` and `add_X`, but in actuality it is again written out in the evaluated form `C L*(X - C x₁) + C y₁` to avoid the `noncomputable` tag.

The slope itself is defined as a conditional over the three cases, and since two of them involve division, this is the first occasion where W needs to be defined over a field F.

```
variables {F : Type} [field F] (W : weierstrass_curve F)

def slope (x₁ x₂ y₁ y₂ : F) : F :=
  if hx : x₁ = x₂ then
    if hy : y₁ = W.neg_Y x₂ y₂ then 0
    else (3*x₁^2 + 2*W.a₂*x₁ + W.a₄ - W.a₁*y₁)/(y₁ - W.neg_Y x₁ y₁)
  else (y₁ - y₂)/(x₁ - x₂)
```

Note that `slope` returns the *junk value* of $0 \in F$ when the slope is vertical. This practice of assigning a junk value is common in `mathlib` to avoid excessive layers of `option`, and any useful result proven using such a definition would include a condition so that this junk value will never be reached. In the case of `slope`, this is the implication $x_1 = x_2 \to y_1 \neq \sigma_{x_2}(y_2)$, which holds precisely either when $x_1 \neq x_2$, or when $x_1 = x_2$ but $(x_1, y_1) \neq -(x_2, y_2)$.

```
variables {x₁ x₂ y₁ y₂ : F} (hxy : x₁ = x₂ → y₁ ≠ W.neg_Y x₂ y₂)

example (hx : x₁ ≠ x₂) : x₁ = x₂ → y₁ ≠ W.neg_Y x₂ y₂ := λ h, (hx h).elim
example (hy : y₁ ≠ W.neg_Y x₂ y₂) : x₁ = x₂ → y₁ ≠ W.neg_Y x₂ y₂ := λ _, hy
```

Here, the examples return proofs of `hxy` assuming $x_1 \neq x_2$ and $y_1 \neq \sigma_{x_2}(y_2)$ respectively. They are illustrated here for clarity, but they do not exist in the actual Lean code since their term mode proofs are short enough to be inserted directly whenever necessary.

To define addition, it remains to prove that the addition of two nonsingular points on W is again a nonsingular point on W. This is slightly lengthy but purely conceptual.

▶ **Lemma 6.** *If $(x_1, y_1), (x_2, y_2) \in W$ are nonsingular, then $(x_3, y_3) \in W$ is nonsingular.*

Proof. By `nonsingular_neg`, it suffices to prove that $(x_3, \lambda(x_3)) = (x_3, y_3')$ is nonsingular, since $(x_3, \lambda(x_3)) \in W$ is clear. Taking derivatives of both sides in (1) yields

$$W_X(X, \lambda(X)) + \ell \cdot W_Y(X, \lambda(X)) = -((X-x_1)(X-x_2) + (X-x_1)(X-x_3) + (X-x_2)(X-x_3)),$$

which does not vanish at $X = x_3$, so that $W(X, \lambda(X))$ has at least one nonvanishing partial derivative, unless possibly when $x_3 \in \{x_1, x_2\}$. The latter implies that $(x_3, \lambda(x_3)) \in \{\pm(x_1, y_1), \pm(x_2, y_2)\}$, but these are nonsingular by assumption or by `nonsingular_neg`. ◀

Lemma 6 is `nonsingular_add`, which accepts a proof that $(x_1, y_1) \in W$ is nonsingular, a proof that $(x_2, y_2) \in W$ is nonsingular, and a proof of `hxy`, and returns a proof that $(x_1, y_1) + (x_2, y_2) \in W$ is nonsingular. This finally leads to the definition of addition.

```
def add : W.point → W.point → W.point
  | 0                    P                    := P
  | P                    0                    := P
  | (@some _ _ _ x₁ y₁ h₁) (@some _ _ _ x₂ y₂ h₂) :=
    if hx : x₁ = x₂ then
      if hy : y₁ = W.neg_Y x₂ y₂ then 0
      else some (nonsingular_add h₁ h₂ (λ _, hy))
    else some (nonsingular_add h₁ h₂ (λ h, (hx h).elim))

instance : has_add W.point := ⟨add⟩
```

Here, the `instance` of `has_add` allows the notation `P₁ + P₂` instead of `add P₁ P₂`. The annotation `@` for `some` simply gives access to all implicit variables in its definition, particularly $x_1, x_2, y_1, y_2 \in F$ that is necessary to even state the conditions `hx` and `hy`.

3 Group law

Let W be a Weierstrass curve over a field F, and denote its set of nonsingular points by $W(F)$. The addition law defined in the previous section is in fact a **group law**.

▶ **Proposition 7.** *$W(F)$ forms an abelian group under this addition law.*

The axioms of this group law are mostly straightforward, typically just by examining the definition for each of the five cases. For instance, the `lemma add_left_neg` that says $-P + P = \mathcal{O}_W$ is immediate, since $-\mathcal{O}_W + \mathcal{O}_W = \mathcal{O}_W$ by definition, and $-(x, y) + (x, y) = (x, \sigma_x(y)) + (x, y) = \mathcal{O}_W$ for any $(x, y) \in W(F)$ by the first case of affine addition.

On the other hand, associativity is far from being straightforward. [6] A notable algebro-geometric proof involves canonically identifying $W(F)$ with its *Picard group*, a natural geometric construction associated to W with a known group structure, and proving that this identification respects the addition law on W [26, Proposition III.3.4]. This is generally regarded as the most conceptual proof, as it explains the seemingly arbitrary secant-and-tangent process, and more crucially because it works for any $\mathrm{char}(F)$.

The proof in this paper is an analogue of this proof, but the arguments involved are purely algebraic without the need for any geometric machinery, in contrast to the typical algebro-geometric proof. The main idea, originally inspired by Buzzard's post on Zulip [6] and Chapman's answer on MathOverflow [8], is to construct an explicit function `to_class` from $W(F)$ to the *ideal class group* $\mathrm{Cl}(R)$ of an integral domain R associated to W, then to prove that this function is injective and respects the addition law on W. This is a construction in commutative algebra whose definition will now be briefly outlined, but for a more comprehensive introduction to ideal class groups motivated by arithmetic examples, and especially specific details of their formalisation in `mathlib`, the reader is strongly urged to consult the original paper by Baanen, Dahmen, Narayanan, and Nuccio [3, Section 2].

3.1 Ideal class group of the coordinate ring

Given an integral domain R with a fraction field K, a **fractional ideal** of R is simply a R-submodule I of K such that $x \cdot I \subseteq R$ for some nonzero $x \in R$. This generalises the notion of an ideal of R, since any ideal is clearly a fractional ideal with $x = 1$, so ideals are sometimes referred to as **integral ideals** to distinguish them from fractional ideals.

In `mathlib`, this is expressed as a transitive coercion from `ideal` to `fractional_ideal`.

```
instance : has_coe_t (ideal R) (fractional_ideal R⁰ (fraction_ring R))
```

Here, R^0 is the submonoid of non-zero-divisors of R, and `fraction_ring` returns the canonical choice of a fraction field of R obtained by adjoining inverses of elements of R^0. Since R is an integral domain in this case, all nonzero elements of R become invertible in its fraction field.

Analogously to integral ideals, the set of fractional ideals of R forms a commutative semiring under the usual operations of addition and multiplication for submodules. A fractional ideal I of R is **invertible** if $I \cdot J = R$ for some fractional ideal J of R, and the subset of invertible fractional ideals of R forms an abelian group under multiplication. An important class of invertible fractional ideals are those generated by a nonzero element of K, called **principal fractional ideals**. The **ideal class group** $\mathrm{Cl}(R)$ of R is then defined to be the quotient group of invertible fractional ideals by principal fractional ideals.

[6] see Section 4.2 for alternative proofs

In `mathlib`, a `class_group` element is constructed from an invertible `fractional_ideal` via `class_group.mk`, and this association is a `monoid_hom` that respects multiplication.

```
def class_group.mk : (fractional_ideal R⁰ K)ˣ →* class_group R := ...
```

Here, it is worth noting that $\mathrm{Cl}(R)$ is typically defined only when R is **Dedekind**, namely when every nonzero fractional ideal is invertible, and in such domains, `class_group.mk0` constructs a `class_group` element directly from a nonzero `fractional_ideal`.

The integral domain in consideration is the **coordinate ring** of W, that is

$$F[W] := F[X, Y]/\langle W(X, Y)\rangle,$$

whose fraction field is the **function field** $F(W) := \mathrm{Frac}(F[W])$ of W.

```
@[derive comm_ring] def coordinate_ring : Type := adjoin_root W.polynomial
abbreviation function_field : Type := fraction_ring W.coordinate_ring

namespace coordinate_ring
```

Here, $W(X, Y)$ is viewed as a quadratic monic polynomial with coefficients in $F[X]$, so `adjoin_root` constructs the quotient ring $F[W]$ by adjoining its root Y. The tag `derive comm_ring` automatically generates a `instance` of `comm_ring` present in `adjoin_root`, [7] while `abbreviation` is just a `def` that inherits every `instance` from `fraction_ring`.

A priori, $F[W]$ is only a commutative ring, but for $\mathrm{Cl}(F[W])$ to make sense it needs to be at least an integral domain, which is straightforward from the shape of $W(X, Y)$.

▶ **Lemma 8.** *$F[W]$ is an integral domain.*

Proof. It suffices to prove that $W(X, Y)$ is prime, but $F[X, Y]$ is a unique factorisation domain since F is a field, so it suffices to prove that $W(X, Y)$ is irreducible. Suppose for a contradiction that it were reducible as a product of two factors. Since it is a monic polynomial in Y, the leading coefficients of the two factors multiply to 1, so without loss of generality

$$W(X, Y) = (Y - p(X))(Y - q(X)),$$

for some polynomials $p(X), q(X) \in F[X]$. Comparing coefficients yields

$$a_1X + a_3 = -(p(X) + q(X)), \qquad -(X^3 + a_2X^2 + a_4X + a_6) = p(X)q(X),$$

which cannot simultaneously hold by considering $\deg_X(p(X))$ and $\deg_X(q(X))$. ◀

Lemma 8 is formalised as an `instance` of `is_domain` for `W.coordinate_ring`. In fact, $F[W]$ is also Dedekind when $\Delta_W \neq 0$, but this will not be necessary in the proof.

▶ Remark 9. This argument with ideal class groups is essentially an algebraic translation of the algebro-geometric argument with Picard groups. An invertible fractional ideal on an integral domain R is equivalent to an invertible sheaf on its spectrum $\mathrm{Spec}(R)$, so the Picard group $\mathrm{Pic}(\mathrm{Spec}(F[W]))$ of invertible sheaves is precisely the ideal class group $\mathrm{Cl}(F[W])$ of invertible fractional ideals [20, Example II.6.3.2]. Note that an invertible R-submodule of $\mathrm{Frac}(R)$ is automatically a fractional ideal of R [14, Theorem 11.6], so the ideal class group may also be defined purely in the language of invertible submodules.

[7] this has a type unification performance issue that will be detailed in Section 4.3

3.2 Construction of `to_class`

The function `to_class` will map a nonsingular affine point $(x, y) \in W(F)$ to the class of the invertible fractional ideal arising from the integral ideal $\langle X - x, Y - y\rangle$. Defining the integral ideal explicitly is straightforward, and its associated fractional ideal is obtained by coercion.

```
def XY_ideal (x : F) (y : F[X]) : ideal W.coordinate_ring :=
  ideal.span {adjoin_root.mk W.polynomial (C (X - C x)),
              adjoin_root.mk W.polynomial (Y - C y)}
```

Here, `ideal.span` constructs an integral ideal generated by the elements of a specified set, and `adjoin_root.mk W.polynomial` is the canonical quotient map $F[X, Y] \to F[W]$. Note also that `XY_ideal` is defined slightly more generally than described, by allowing the second argument to be a polynomial in $F[X]$ rather than just a constant.

On the other hand, checking that `XY_ideal` is indeed invertible is slightly fiddly.

▶ **Lemma 10.** *For any* $(x, y) \in W(F)$,

$$\langle X - x, Y - \sigma_x(y)\rangle \cdot \langle X - x, Y - y\rangle = \langle X - x\rangle.$$

Proof. Since $W(x, y) = 0$, there is an identity in $F[W]$ given by

$$(Y - y)(Y - \sigma_x(y)) \equiv (X - x)(X^2 + (x + a_2)X + (x^2 + a_2x + a_4) - a_1Y),$$

so the required equality may be reduced to $\langle X - x\rangle \cdot I = \langle X - x\rangle$ in $F[X, Y]$, where

$$I := \langle X - x, Y - y, Y - \sigma_x(y), X^2 + (x + a_2)X + (x^2 + a_2x + a_4) - a_1Y\rangle.$$

Since (x, y) is nonsingular, either $W_X(x, y) \neq 0$ or $W_Y(x, y) \neq 0$, but

$$W_X(x, y) = -(X + 2x + a_2)(X - x) + a_1(Y - y) + (X^2 + (x + a_2)X + (x^2 + a_2x + a_4) - a_1Y),$$

and $W_Y(x, y) = -(Y - y) + (Y - \sigma_x(y))$, so $I = F[X, Y]$ in both cases. ◀

▶ Remark 11. Geometrically, Lemma 10 says that the line $X = x$ intersects W at $\mathcal{O}_W$ and at precisely two affine points (x, y) and $(x, \sigma_x(y))$, counted with multiplicity if they are equal.

Lemma 10 is `XY_ideal_neg_mul`, and it follows that the fractional ideal $\langle X - x, Y - y\rangle$ has inverse $\langle X - x, Y - \sigma_x(y)\rangle \cdot \langle X - x\rangle^{-1}$. This is formalised as `XY_ideal'_mul_inv`, which maps a proof that $(x, y) \in W$ is nonsingular to a proof that the fractional ideal $\langle X - x, Y - y\rangle$ has the specified right inverse. Passing this proof to `units.mk_of_mul_eq_one` then constructs the invertible fractional ideal of $F[W]$ associated to $\langle X - x, Y - y\rangle$.

```
def XY_ideal' (h : W.nonsingular x y) :
  (fractional_ideal W.coordinate_ring⁰ W.function_field)ˣ :=
  units.mk_of_mul_eq_one _ _ (XY_ideal'_mul_inv h)
```

Now `to_class` will be a `add_monoid_hom`, namely a function bundled with proofs that it maps zero to zero and respects addition. Its underlying unbundled function $W(F) \to \mathrm{Cl}(F[W])$, appropriately named `to_class_fun`, is defined separately to allow the equation compiler to generate lemmas automatically used in the proof that `to_class` respects addition.

```
def to_class_fun : W.point → additive (class_group W.coordinate_ring)
  | 0        := 0
  | (some h) := additive.of_mul (class_group.mk (XY_ideal' h))
```

Here, `additive G` creates a type synonym of a multiplicative group `G`, and the multiplicative `group` instance on `G` is turned into an additive `add_group` instance on `additive G`. This is necessary to bundle `to_class` as an `add_monoid_hom`, since `mathlib` does not have homomorphisms between an additive group and a multiplicative group by design.

Now `to_class_fun` maps zero to zero by construction, but proving that it respects addition requires checking the five cases for `add` separately. The first two cases are trivial and the third case follows from `XY_ideal_neg_mul`, while the last two cases are handled simultaneously by assuming `hxy` and checking an identity of integral ideals of $F[W]$.

▶ **Lemma 12.** *For any* $(x_1, y_1), (x_2, y_2) \in W(F)$*, if* $x_1 = x_2$ *implies* $y_1 \neq \sigma_{x_2}(y_2)$*, then*

$$\langle X - x_1, Y - y_1\rangle \cdot \langle X - x_2, Y - y_2\rangle \cdot \langle X - x_3\rangle = \langle X - x_3, Y - y_3\rangle \cdot \langle Y - \lambda(X)\rangle,$$

where $(x_3, y_3) := (x_1, y_1) + (x_2, y_2)$.

Proof. In both valid cases of `hxy`, the line $Y = \lambda(X)$ contains (x_1, y_1) and (x_2, y_2), so

$$\langle X - x_1, Y - y_1\rangle = \langle X - x_1, Y - \lambda(X)\rangle, \qquad \langle X - x_2, Y - y_2\rangle = \langle X - x_2, Y - \lambda(X)\rangle.$$

Furthermore, by (1) and the identity $W(X, \lambda(X)) \equiv (Y - \lambda(X))(\sigma_X(Y) - \lambda(X))$ in $F[W]$, the required equality is reduced to checking that $I := \langle X - x_1, X - x_2, Y - \sigma_X(Y)\rangle$ satisfies

$$I \cdot \langle X - x_3\rangle + \langle \sigma_X(Y) - \lambda(X)\rangle = \langle X - x_3, Y - y_3\rangle,$$

where $Y - \lambda(X)$ has been replaced by $Y - \sigma_X(Y)$ in I since $\sigma_X(Y) - \lambda(X)$ is present as a summand in the left hand side. By construction, the line $Y = \lambda(X)$ contains $(x_3, \lambda(x_3))$, so the negated line $\sigma_X(Y) = \lambda(X)$ contains its negation $(x_3, \sigma_{x_3}(\lambda(x_3))) = (x_3, y_3)$. Then

$$\langle X - x_3, Y - y_3\rangle = \langle X - x_3, \sigma_X(Y) - \lambda(X)\rangle,$$

so it suffices to check that $I = F[W]$. Now $x_1 - x_2 = -(X - x_1) + (X - x_2)$, so $I = F[W]$ if $x_1 \neq x_2$. Otherwise $y_1 \neq \sigma_{x_1}(y_1)$, then there are no common solutions to $Y = \sigma_{x_1}(Y)$ and $W(x_1, Y) = 0$, so $I = F[W]$ by the Nullstellensatz. Explicitly, this follows from the identity

$$(y_1 - \sigma_{x_1}(y_1))^2 \equiv -(4X^2 + (4x_1 + b_2)X + (4x_1^2 + b_2x_1 + 2b_4))(X - x_1) + (Y - \sigma_X(Y))^2$$

in $F[W]$, since $W(x_1, y_1) = 0$. ◀

▶ Remark 13. Geometrically, the line $Y = \lambda(X)$ intersects W at precisely three affine points (x_1, y_1), (x_2, y_2), and $(x_3, \sigma_{x_3}(y_3))$, which translates to the identity of integral ideals

$$\langle X - x_1, Y - y_1\rangle \cdot \langle X - x_2, Y - y_2\rangle \cdot \langle X - x_3, Y - \sigma_{x_3}(y_3)\rangle = \langle Y - \lambda(X)\rangle. \tag{2}$$

The identity in Lemma 12 is then deduced by multiplying (2) with the identity in Lemma 10 and cancelling $\langle X - x_3, Y - \sigma_{x_3}(y_3)\rangle$ from both sides. Note that Lemma 12 does not need the affine points to be nonsingular, while directly proving (2) does.

Lemma 12 is `XY_ideal_mul_XY_ideal`, and under these hypotheses, it follows immediately that the invertible fractional ideals $\langle X - x_1, Y - y_1\rangle$ and $\langle X - x_2, Y - y_2\rangle$ multiply to $\langle X - x_3, Y - y_3\rangle$ as classes in $\mathrm{Cl}(F[W])$, which along with `XY_ideal_neg_mul` say that `to_class` respects addition. The actual Lean proof is slightly technical, using the new library lemmas `class_group.mk_eq_one_of_coe_ideal` and `class_group.mk_eq_mk_of_coe_ideal` to reduce the equality between ideal classes arising from integral ideals to an equality between their underlying integral ideals up to multiplication by principal integral ideals, so the tactic mode proof below will only be sketched as a comment for the sake of brevity.

```
@[simps] def to_class : W.point →+ additive (class_group W.coordinate_ring) :=
  { to_fun    := to_class_fun,
    map_zero' := rfl,
    map_add'  := /- Split the cases for P₁ + P₂. If P₁ = 0 or P₂ = 0, simplify.
                    Otherwise P₁ = (x₁, y₁) and P₂ = (x₂, y₂).
                      If x₁ = x₂ and y₁ = W.neg_Y x₂ y₂, use XY_ideal_neg_mul.
                      Otherwise use XY_ideal_mul_XY_ideal. -/ }
```

3.3 Injectivity of `to_class`

Injectivity is the statement that $P_1 = P_2$ if `to_class` of P_1 equals `to_class` of P_2 for any $P_1, P_2 \in W(F)$, but a simple variant of `add_left_neg` shows that $-P_1 + P_2 = 0$ precisely when $P_1 = P_2$. Since `to_class` is a `add_monoid_hom`, injectivity is equivalent to showing that `to_class` of P is trivial implies $P = 0$ for any $P \in W(F)$. In other words, it suffices to show that the integral ideal $\langle X - x, Y - y \rangle$ is never principal for any affine point $(x, y) \in W(F)$.

The approach taken circles around the fact that $F[W]$ is a free $F[X]$-algebra of finite rank, so it carries the notion of a **norm** $\mathrm{Nm} : F[W] \to F[X]$. If $f \in F[W]$, then $\mathrm{Nm}(f) \in F[X]$ may be given by the determinant of left multiplication by f as an $F[X]$-linear map, which is most easily computed by exhibiting an explicit basis $\{1, Y\}$ of $F[W]$ over $F[X]$.

```
lemma monic_polynomial : W.polynomial.monic
lemma nat_degree_polynomial : W.polynomial.nat_degree = 2

def basis : basis (fin 2) F[X] W.coordinate_ring :=
  (adjoin_root.power_basis' W.monic_polynomial).basis.reindex
    (fin_congr W.nat_degree_polynomial)
```

Here, `adjoin_root.power_basis'` returns the canonical basis of powers $\{Y^i : 0 \leq i < \deg_Y(W(X, Y))\}$, given the proof `monic_polynomial` that $W(X, Y)$ is monic. This is a type indexed by the finite type with $\deg_Y(W(X, Y))$ elements, which can be reindexed by the canonical finite type with two elements, since $\deg_Y(W(X, Y)) = 2$ by `nat_degree_polynomial`.

With this basis, any element $f \in F[W]$ may be expressed uniquely as $f = p(X) + q(X)Y$ with $p(X), q(X) \in F[X]$, and the degree [8] of its norm can be computed directly.

▶ **Lemma 14.** *For any $p(X), q(X) \in F[X]$,*

$$\deg_X(\mathrm{Nm}(p(X) + q(X)Y)) = \max(2 \deg_X(p(X)), 2 \deg_X(q(X)) + 3).$$

Proof. Let $f := p(X) + q(X)Y$. In $F[W]$ with the basis $\{1, Y\}$ over $F[X]$,

$$\begin{aligned}\mathrm{Nm}(f) &\equiv \det \begin{pmatrix} p(X) & q(X) \\ q(X)(X^3 + a_2X^2 + a_4X + a_6) & p(X) - q(X)(a_1X + a_3) \end{pmatrix} \\ &= p(X)^2 - p(X)q(X)(a_1X + a_3) - q(X)^2(X^3 + a_2X^2 + a_4X + a_6).\end{aligned}$$

Let $p := \deg_X(p(X))$ and $q := \deg_X(q(X))$. Then

$$\deg_X(p(X)^2) = 2p, \qquad \deg_X(q(X)^2(X^3 + a_2X^2 + a_4X + a_6)) = 2q + 3,$$

$$\deg_X(p(X)q(X)(a_1X + a_3)) \leq p + q + 1.$$

If $p \leq q + 1$, then both $p + q + 1 < 2q + 3$ and $2p < 2q + 3$, so $\deg_X(\mathrm{Nm}(f)) = 2q + 3$. Otherwise $q + 1 < p$, then both $p + q + 1 < 2p$ and $2q + 3 < 2p$, so $\deg_X(\mathrm{Nm}(f)) = 2p$. ◀

Lemma 14 is `norm_smul_basis`, and it follows by considering cases that $\deg_X \mathrm{Nm}(f) \neq 1$ for any $f \in F[W]$, which is formalised as `nat_degree_norm_ne_one`.

▶ Remark 15. Geometrically, $\mathrm{Nm}(f)$ is the order of the pole of the rational function $f \in F(W)$ at $\mathcal{O}_W$. Using the norm allows for a purely algebraic argument for injectivity, which was inspired from an exercise in Hartshorne that assumes a short Weierstrass model where $\mathrm{char}(F) \neq 2$ [20, Exercise I.6.2]. This was also the last missing step in the whole argument, as JX only started computing degrees after he saw Borcherds's solutions to the exercise [5].

On the other hand, this degree is also the dimension of an F-vector space.

[8] `polynomial.degree` where $\deg_X(0) := -\infty$ rather than `polynomial.nat_degree` where $\deg_X(0) := 0$

▶ **Lemma 16.** *For any nonzero $f \in F[W]$,*

$$\deg_X(\mathrm{Nm}(f)) = \dim_F(F[W]/\langle f\rangle).$$

Proof. In $F[W]$ with the basis $\{1, Y\}$ over $F[X]$, multiplication by f as an $F[X]$-linear map can be represented by a square matrix $[f]$ over $F[X]$, which has a Smith normal form $M[f]N$, a diagonal matrix with diagonal entries some nonzero $p(X), q(X) \in F[X]$, for some invertible matrices M and N over $F[X]$. Now the quotient by f decomposes as a direct sum

$$F[W]/\langle f\rangle \cong F[X]/\langle p(X)\rangle \oplus F[X]/\langle q(X)\rangle,$$

whose dimension as F-vector spaces are precisely $\deg_X(p(X))$ and $\deg_X(q(X))$ respectively. On the other hand, the determinant of $M[f]N$ is $\det(M)\mathrm{Nm}(f)\det(N) = p(X)q(X)$, so

$$\deg_X(\mathrm{Nm}(f)) = \deg_X(p(X)) + \deg_X(q(X)),$$

since the units $\det(M), \det(N) \in F[X]$ are nonzero constant polynomials. ◀

Lemma 16 is `finrank_quotient_span_eq_nat_degree_norm`, and crucially uses the library lemma `ideal.quotient_equiv_pi_span` to decompose the quotient by $\langle f\rangle$ into a direct sum of quotients by its Smith coefficients. It is worth noting that the same argument clearly works more generally by replacing $F[W]$ by any $F[X]$-algebra with a finite basis. The proof of the injectivity of `to_class` then proceeds by contradiction.

▶ **Lemma 17.** *The function $W(F) \to \mathrm{Cl}(F[W])$ is injective.*

Proof. Let $(x, y) \in W(F)$. It suffices to show that $\langle X - x, Y - y\rangle$ is not principal, so suppose for a contradiction that it were generated by some $f \in F[W]$. By Lemma 16,

$$\deg_X(\mathrm{Nm}(f)) = \dim_F(F[W]/\langle f\rangle) = \dim_F(F[W]/\langle X - x, Y - y\rangle).$$

On the other hand, evaluating at $(X, Y) = (x, y)$ is a surjective homomorphism $F[X, Y] \to F$ with kernel $\langle X - x, Y - y\rangle$, and this contains the element $W(X, Y)$ since $W(x, y) = 0$. Explicitly, this follows from the identity in $F[X, Y]$ given by

$$W(X, Y) = (a_1 y - (X^2 + (x + a_2)X + (x^2 + a_2 x + a_4)))(X - x) + (y - \sigma_X(Y))(Y - y).$$

Thus by the first and third isomorphism theorems, there are F-algebra isomorphisms

$$F[W]/\langle X - x, Y - y\rangle \xrightarrow{\sim} F[X, Y]/\langle W(X, Y), X - x, Y - y\rangle = F[X, Y]/\langle X - x, Y - y\rangle \xrightarrow{\sim} F,$$

so $\deg_X(\mathrm{Nm}(f)) = \dim_F(F) = 1$, which contradicts `nat_degree_norm_ne_one`. ◀

▶ Remark 18. Lemma 17 can also be proven without the Smith normal form, by considering the ideal generated by the norms of elements in $\langle X - x, Y - y\rangle$ for $(x, y) \in W(F)$, namely

$$I := \langle (X - x)^2, (X - x)((Y - y) + (\sigma_X(Y) - y)), (Y - y)(\sigma_X(Y) - y)\rangle.$$

On one hand, as an integral ideal in $F[W]$, it can be shown that I is generated by the linear polynomial $X - x$. On the other hand, if $\langle X - x, Y - y\rangle$ were generated by some $f \in F[W]$, then its ideal norm I is generated by $\mathrm{Nm}(f)$, which cannot be linear by Lemma 14.

Lemma 17 is `to_class_injective`, and allows the proofs of commutativity and associativity in $\mathrm{Cl}(F[W])$ to be pulled back to $W(F)$, thus proving Proposition 7.

```
lemma add_comm (P₁ P₂ : W.point) : P₁ + P₂ = P₂ + P₁
lemma add_assoc (P₁ P₂ P₃ : W.point) : (P₁ + P₂) + P₃ = P₁ + (P₂ + P₃)

instance : add_comm_group W.point :=
  ⟨zero, neg, add, zero_add, add_zero, add_left_neg, add_comm, add_assoc⟩
```

4 Discussion

4.1 Related work

As aforementioned, formalising the group law of an elliptic curve E over a field F is not novel, and has been done in several theorem provers to varying extents. Friedl (1998) [16] gave a computational proof in the short Weierstrass model, leaving some of the heavy computations for associativity to CoCoA as a trusted oracle, and his argument was subsequently formalised by Théry (2007) [28] in Coq. Fox, Gordon, and Hurd (2006) [15] formalised the addition law in the full Weierstrass model in HOL, but did not prove associativity. Hales and Raya (2020) [18] formalised a computational proof in Isabelle, but worked in the alternative Edwards model, which also fails to be an elliptic curve when $\mathrm{char}(F) = 2$.

The first known formalisation of an algebro-geometric proof was done by Bartzia and Strub (2014) [4], who also worked in the short Weierstrass model. In 3,500 lines of Coq, they formalised the geometric notion of a Weil divisor [9] of a rational function $f \in F(E)$ to define the degree-zero Weil divisor class group $\mathrm{Pic}^0(E)$, which is isomorphic to the Picard group $\mathrm{Pic}(\mathrm{Spec}(F[E]))$ since E is nonsingular [20, Corollary II.6.16]. In another 6,500 lines of Coq, they constructed an analogous bijection between $\mathrm{Pic}^0(E)$ and the points of E over the algebraic closure, but their argument is a simplification of the typical conceptual proof via the Riemann–Roch theorem and does not generalise easily to $\mathrm{char}(F) = 2$. In contrast, the algebraic proof with the ideal class group $\mathrm{Cl}(F[E])$ only spans 1,500 lines of Lean 3, avoiding the geometric theory and reusing much of the well-maintained algebraic libraries.

4.2 Experimental attempts

The entire development process went through several iterations of trial and error, and various definitions of elliptic curves were proposed in Buzzard's topic on Zulip. The abstract definition as in Remark 2 would be ideal, but algebraic geometry in `mathlib` is at its primitive stages, where describing properties of scheme morphisms like smoothness or properness, or defining the genus of a curve, would be a challenge. Since the Weierstrass model is universal over fields, the general consensus was that proving its equivalence with the abstract definition should proceed independently from proving theorems under the Weierstrass model.

Unfortunately, proving associativity became a huge issue in this model. The obvious first course of action is to check the equalities in all possible combinations of cases of addition, using the `field_simp` and `ring` tactics to normalise rational expressions. In doing this, the number of cases quickly explode, and in the nontrivial cases of affine addition, the polynomial expressions involved become gargantuan. There are optimisations that could be made to reduce the number of cases, as coded by Masdeu [23] adapting Friedl's original argument into Lean, but a good way to manipulate the expressions remains elusive. In the generic case where three nonsingular affine points $P_1, P_2, P_3 \in W(F)$ are in general position, [10] experiments by DKA with the aid of SageMath suggested that proving $(P_1 + P_2) + P_3 = P_1 + (P_2 + P_3)$ by bashing out the algebra would involve polynomials each with tens of thousands of monomials, which is highly time-consuming in a formal system and definitely infeasible to work out by hand, despite taking only half a second in SageMath. The `ring` tactic, which uses proof generation by normalising to Horner form [17], seems to be an order of magnitude too slow to work with such expressions, resulting in deterministic timeouts.

[9] a formal $\mathbb{Z}$-linear combination of points $P \in E$ weighted by the order of vanishing of f at P
[10] the affine points P_1, P_2, P_3, $P_1 + P_2$, $P_1 + P_3$, and $P_2 + P_3$ have pairwise distinct X-coordinates

The main culprits for the huge polynomials are the XY and Y terms in the Weierstrass equation, which do not allow even exponents of Y in the expressions to be substituted with polynomials solely in X. When $\mathrm{char}(F) \neq 2$, these terms disappear with a change of variables, reducing the expressions to the computationally feasible range of hundreds of terms, hence enabling the work by Théry (2007), or a transformation to the Edwards model whose group law was already formalised by Hales and Raya (2020). In principle, since $2 = 0$ when $\mathrm{char}(F) = 2$, enough multiples of 2 may be cancelled from the expressions until a brute-force attack becomes feasible, but `mathlib` currently has no good tactic to do these cancellations except to manually extract these multiples of 2, such as by rewriting the expressions into the form $p + 2q$ using `ring`, which is too slow in the first place, and deleting $2q$.

The mathematical literature typically deals with associativity by providing alternative proofs, in addition to the aforementioned algebro-geometric proof via the Picard group. One notable method goes via the uniformisation theorem in complex analysis [26, Corollary VI.5.1.1], but `mathlib` also lacks much of the complex-analytic machinery to prove it, and this approach is only valid for $\mathrm{char}(F) = 0$ via the Lefschetz principle. Another approach uses the Cayley–Bacharach theorem in projective geometry [7, Lemma 7.1], which proves associativity generically by a dimension counting argument. By continuing on Masdeu's branch, this approach seemed viable, but required redefining Weierstrass curves in projective coordinates and a convenient way to switch back to affine coordinates via dehomogenisation. Furthermore, the argument fails in a less generic case with a repeated point, which could be fixed by introducing an ad-hoc notion of intersection multiplicity between a line and a cubic, as suggested by Stoll. DKA started refactoring the definitions in an attempt at this approach, but ultimately switched to the current approach when proposed by JX. Note that an explicit exposition of a version of this argument can also be found in Nuida (2021) [24].

4.3 Implementation issues

Bivariate polynomials. A bivariate polynomial in X and Y over a commutative ring R is typically represented in `mathlib` by a finitely supported function $(\{0,1\} \to \mathbb{N}) \to R$, associating a function $f : \{0,1\} \to \mathbb{N}$ to the coefficient of $X^{f(0)}Y^{f(1)}$. This representation is very cumbersome when performing concrete manipulations, such as those in Lemma 10 and Lemma 12, since explicit functions $\{0,1\} \to \mathbb{N}$ are needed to obtain coefficients.

In contrast, a polynomial in X over R is represented in `mathlib` by a finitely supported function $\mathbb{N} \to R$, associating a natural number $n \in \mathbb{N}$ to the coefficient of X^n. A polynomial in Y with coefficients polynomials in X performs the same function as a bivariate polynomial in X and Y, but the coefficient of X^nY^m is obtained by sequentially supplying two natural numbers $m, n \in \mathbb{N}$. This has the additional advantage of aligning with the API for `adjoin_root`, which gives a power basis needed in the proof of injectivity.

This representation does have the slightly awkward problem that X is denoted by `C X` while Y is denoted by `X`, but this is easily fixed by introducing `notation Y := X` and `notation R[X][Y] := polynomial (polynomial R)`. A more serious drawback is that existing results about multivariate polynomials, such as the Nullstellensatz, do not carry over to this representation, so explicit proofs with polynomial identities are sometimes necessary, namely in the proofs of Lemma 10, Lemma 12, and Lemma 17. Another issue is that the partial derivative with respect to X is obtained by applying the `polynomial.derivative` linear map to each coefficient of the polynomial in Y, but the current `polynomial.map` only accepts a ring homomorphism, which explains why the partial derivatives `polynomial_X` and `polynomial_Y` were defined manually instead. In light of this, it has been suggested that `polynomial.map` should be refactored to accept set-theoretic functions instead.

Performance issues. In the original definition of `to_class`, it was observed that the function `class_group.mk`, when applied to an invertible fractional ideal of `coordinate_ring`, took a while to compile. Baanen diagnosed this problem and proposed the following solution [1].

```
local attribute [irreducible] coordinate_ring.comm_ring
```

Although `coordinate_ring` is marked as `irreducible`, its `derive comm_ring` tag generates a `reducible` instance of `comm_ring`. In certain circumstances this is extremely slow, because the number of times an instance gets unified grows exponentially with its depth due to a lack of caching, and Baanen's solution was to force its `comm_ring` instance to be irreducible locally whenever necessary. Note that this should have been fixed in Lean 4, and the port of `mathlib` to Lean 4 is expected to finish in a few months' time.

There are other performance issues that led to timeouts during development, but they were fixed by generalising the statements so they involve less complicated types.

Proof automation. The proofs of many basic lemmas often reduce to checking an equality of two polynomial expressions, such as in Lemma 5 and Lemma 6, but equality often holds only under some local hypotheses. Rather than rewriting these into the goal and applying the `ring` tactic, it is convenient to use `linear_combination`, a newly-developed tactic that subtracts a linear combination of known equalities from the goal, before applying `ring`.

When several rewrite lemmas are often used together, it is also convenient to write a custom tactic to chain them. For instance, the evaluation map `eval` on a polynomial expression is often propagated inwards, so grouping the lemmas allows for a single tactic call.

```
meta def eval_simp : tactic unit :=
`[simp only [eval_C, eval_X, eval_neg, eval_add, eval_sub, eval_mul, eval_pow]]
```

4.4 Future projects

Formalising the group law opens the doors to an expansive array of possible further work. An immediate project would be to enrich the API for nonsingular points by adding basic functorial properties with respect to a base change to a field extension K/F. For instance, this could be defining the induced map $E(F) \to E(K)$, or if K/F is Galois, computing the subgroup of $E(K)$ invariant under the action of $\mathrm{Gal}(K/F)$ to be precisely $E(F)$.

It is worth noting the two ongoing projects by each of the two authors. DKA is formalising an inductive definition of division polynomials to understand the structure of the n-torsion subgroup $E[n]$ to compute the structure of the ℓ-adic Tate module $\varprojlim_n E[\ell^n]$, while JX is formalising a proof that the reduction map $E(K) \to E(R/\mathfrak{m})$ is a group homomorphism for a discrete valuation ring R with fraction field K and maximal ideal $\mathfrak{m}$.

In the longer run, one could explore the rich arithmetic theory over specific fields. Once the theory of local fields is sufficiently developed in `mathlib`, one could define the formal group of an elliptic curve, classify its reduction types, or state Tate's algorithm. These will be useful for the global theory, where one could define the Selmer and Tate–Shafarevich groups, give a Galois cohomological proof of the Mordell–Weil theorem, or state the full Birch and Swinnerton-Dyer conjecture. Over a finite field, one could verify the correctness of primality and factorisation algorithms as well as cryptographic protocols, or prove the Hasse–Weil bound or the Weil conjectures for elliptic curves.

Ultimately, a long term goal would be to redefine elliptic curves in `mathlib` as in Remark 2 and prove Proposition 1, but this will require a version of the Riemann–Roch theorem, whose proof will require a robust theory of sheaves of modules and their cohomology.

References

1 David Angdinata. class_group. URL: https://leanprover-community.github.io/archive/stream/116395-maths/topic/Why.20is.20class_group.2Emk.20so.20slow.3F.html.

2 A. O. L. Atkin and François Morain. Elliptic curves and primality proving. *Mathematics of Computation*, 61(203):29–68, 1993. doi:10.2307/2152935.

3 Anne Baanen, Sander Dahmen, Ashvni Narayanan, and Filippo Nuccio Mortarino Majno di Capriglio. A formalization of Dedekind domains and class groups of global fields. *Journal of Automated Reasoning*, 66:611–637, 2022. doi:10.1007/s10817-022-09644-0.

4 Evmorfia-Iro Bartzia and Pierre-Yves Strub. A formal library for elliptic curves in the Coq proof assistant. *ITP*, pages 77–92, 2014. doi:10.1007/978-3-319-08970-6_6.

5 Richard Borcherds. Hartshorne, Chapter 1.6, Answers to Exercises. URL: https://math.berkeley.edu/~reb/courses/256A/1.6.pdf.

6 Kevin Buzzard. Thoughts on elliptic curves. URL: https://leanprover-community.github.io/archive/stream/116395-maths/topic/thoughts.20on.20elliptic.20curves.html.

7 J. W. S. Cassels. *Lectures on Elliptic Curves*. Cambridge University Press, 1991.

8 Robin Chapman. Why is an elliptic curve a group? URL: https://mathoverflow.net/q/20623.

9 Lily Chen, Dustin Moody, Karen Randall, Andrew Regenscheid, and Angela Robinson. Recommendations for discrete logarithm-based cryptography: elliptic curve domain parameters, 2023. doi:10.6028/NIST.SP.800-186.

10 The Mathlib Community. mathlib documentation. URL: https://leanprover-community.github.io/mathlib_docs/.

11 The Mathlib Community. The Lean mathematical library. *CPP*, 2020. doi:10.1145/3372885.3373824.

12 Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). *CADE*, 2015. doi:10.1007/978-3-319-21401-6_26.

13 Pierre Philip du Preez. Understanding EC Diffie-Hellman. URL: https://medium.com/swlh/understanding-ec-diffie-hellman-9c07be338d4a.

14 David Eisenbud. *Commutative algebra with a view toward algebraic geometry*. Springer New York, 1995.

15 Anthony Fox, Mike Gordon, and Joe Hurd. Formalized elliptic curve cryptography. *High Confidence Software and Systems*, 2006.

16 Stefan Friedl. An elementary proof of the group law for elliptic curves. *Groups Complexity Cryptology*, 9(2):117–123, 2017. doi:10.1515/gcc-2017-0010.

17 Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. *Lecture Notes in Computer Science*, 3603:98–113, 2005. doi:10.1007/11541868_7.

18 Thomas Hales and Rodrigo Raya. Formal proof of the group law for Edwards elliptic curves. *Automated Reasoning*, 12167:254–269, 2020. doi:10.1007/978-3-030-51054-1_15.

19 Kevin Hartnett. Math quartet joins forces on unified theory. URL: https://www.quantamagazine.org/math-quartet-joins-forces-on-unified-theory-20151208/.

20 Robin Hartshorne. *Algebraic Geometry*. Springer New York, 1977.

21 Nicholas Katz and Barry Mazur. *Arithmetic Moduli of Elliptic Curves*. Princeton University Press, 1985.

22 Hendrik Lenstra. Factoring integers with elliptic curves. *Annals of Mathematics*, 126(3):649–673, 1987. doi:10.2307/1971363.

23 Marc Masdeu. ell_add_assoc. URL: https://github.com/leanprover-community/mathlib/blob/ell_add_assoc/src/algebraic_geometry/EllipticCurveGroupLaw.lean.

24 Koji Nuida. An elementary linear-algebraic proof without computer-aided arguments for the group law on elliptic curves. *International Journal of Mathematics for Industry*, 13(1), 2021. doi:10.1142/S2661335221500015.

25 David Russinoff. A computationally surveyable proof of the group properties of an elliptic curve. *In Proceedings ACL2Workshop*, 2017. doi:10.4204/EPTCS.249.3.

26 Joseph Silverman. *The Arithmetic of Elliptic Curves*. Springer New York, 2009.

27 Andrew Sutherland. Algebraic proof of the associativity of the elliptic curve group law on curves defined by a short Weierstrass equation, as presented in Lecture 2 of 18.783. URL: https://cocalc.com/share/public_paths/a6a1c2b188bd61d94c3dd3bfd5aa73722e8bd38b.

28 Laurent Théry. Proving the group law for elliptic curves formally. *INRIA*, 2007.

29 Andrew Wiles. The Birch and Swinnerton-Dyer conjecture. URL: https://www.claymath.org/sites/default/files/birchswin.pdf.

30 Andrew Wiles. Modular elliptic curves and Fermat's last theorem. *Annals of Mathematics*, 141(3):443–551, 1995. doi:10.2307/2118559.

A Proof-Producing Compiler for Blockchain Applications

Jeremy Avigad
Carnegie Mellon University, Pittsburgh, PA, USA

Lior Goldberg
StarkWare Industries Ltd., Netanya, Israel

David Levit
StarkWare Industries Ltd., Netanya, Israel

Yoav Seginer
Amsterdam, Netherlands

Alon Titelman
StarkWare Industries Ltd., Netanya, Israel

Abstract

Cairo is a programming language for running decentralized applications (dapps) at scale. Programs written in the Cairo language are compiled to machine code for the Cairo CPU architecture, and cryptographic protocols are used to verify the results of the execution traces efficiently on blockchain. We explain how we have extended the Cairo compiler with tooling that enables users to prove, in the Lean 3 proof assistant, that compiled code satisfies high-level functional specifications. We demonstrate the success of our approach by verifying primitives for computations with an elliptic curve over a large finite field, as well as their use in the validation of cryptographic signatures.

2012 ACM Subject Classification General and reference → Verification; Theory of computation → Logic and verification; Software and its engineering → Semantics

Keywords and phrases formal verification, smart contracts, interactive proof systems

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.7

Supplementary Material *Software*: `https://github.com/starkware-libs/formal-proofs` archived at `swh:1:dir:6e3f3ad6721711a9e6dc1643d890edd6835e196e`

Acknowledgements We are grateful to the Lean developers and the Lean community for providing infrastructure for this project, and to three anonymous reviewers for numerous corrections and improvements.

1 Introduction

Cairo [16] is a programming language for running decentralized applications (dapps) at scale. Programs written in the Cairo language are compiled to machine code for the Cairo CPU architecture [14], which is run off chain by an untrusted prover. Using the STARK cryptographic proof system [6], the prover then publishes a succinct certificate for the result of the off-chain computation, which is verified efficiently on blockchain.

Here we describe an augmentation of the Cairo compiler that enables users to produce formal proofs that compiled machine code meets its high-level specifications. We retain enough information during the compilation phase for our verification tool to extract a description of the machine code as well as naive functional specifications of the source code. We automatically construct formal proofs, in the Lean 3 proof assistant [9], that the machine code meets these specifications. Users can then write their own specifications of the source code in Lean and prove that they are implied by the automatically generated ones. In

© Jeremy Avigad, Lior Goldberg, David Levit, Yoav Seginer, and Alon Titelman;
licensed under Creative Commons License CC-BY 4.0

doing so, they can make use of their specifications of functions earlier in the dependency chain. Using Lean to check both the user-written and autogenerated proofs yields end-to-end verification of the user's specifications, down to CPU semantics. In other work [4], we have moreover verified the correctness of the algebraic encoding of the CPU semantics that is used to generate the certificates used in the STARK protocol.

In Sections 2 to 4 we describe the Cairo CPU architecture, assembly code, and programming language, and we explain how we generate high-level specifications and construct the formal correctness proofs in Lean. Section 5 focuses on features of our work that are specific to the domain of application and the STARK encoding, namely, memory management in Cairo and mechanisms for computing with elements of a finite field. In Section 6, we demonstrate the success of our approach by describing our verification of elliptic curve computations over a large finite field and our verification of a procedure for validating digital signatures. In Section 7, we explain why our approach has been effective in practice, enabling us to verify production code without hindering the development of the compiler or the library. Our main contributions are therefore as follows:

- We provide means of obtaining end-to-end verification, in a foundational proof assistant, of Cairo machine code with respect to high-level specifications.
- We handle novel features of the execution model that are specific to its use in blockchain applications.
- We explain how we managed to carry out our work in an industrial setting, while the language and compiler were under continuous development.
- We explain why our approach, which involves automatically generating source-level proofs that are elaborated and checked by Lean, has been surprisingly effective.
- We demonstrate that our approach scales well by presenting a substantial case study, an implementation of digital signature validation that is already being used in production.

Our Lean libraries, our verification tool, and the case study described here can be found online at `https://github.com/starkware-libs/formal-proofs`. At the time of writing, version 0.10 of the Cairo language has been released, and the verifier and libraries accompanying this paper correspond to that release. StarkWare is currently developing the next generation of the programming language, Cairo 1, which will be substantially different. Cairo 1, however, will call libraries and primitives implemented in Cairo 0, and this project is currently being used to verify those libraries and primitives. All references to the Cairo language in this paper therefore refer to Cairo 0.

2 The Cairo Machine Model

The Cairo machine model is based on a simple CPU architecture with three registers: a *program counter* (pc), which points to the current instruction in memory; a *frame pointer* (fp), which generally points to the location of the local variables in a function call, and an *allocation pointer* (ap), which generally points to the next free value in global working memory. A machine instruction consists of three 16-bit words, which generally serve as memory offsets for operations performed on memory, and 15 1-bit flags, which determine the nature of the instruction. The architecture is described in detail in the Cairo whitepaper [14], and a Lean formalization thereof is described in [4].

A notable feature of the machine model is that elements of memory, as well as the contents of the registers, are elements of the field of integers modulo a certain prime number (by default, $2^{251} + 17 \cdot 2^{192} + 1$). The CPU can add and multiply values, but it cannot ask

whether one value is greater than another. Cryptographic primitives, described in Section 5, can be used to assert that the contents of a memory location represent the cast of an integer in a certain range. The core library uses values that are checked to lie in the interval $[0, 2^{128})$.

Another notable feature of the machine model is that the memory is read-only. To establish a computational claim on blockchain, a prover makes public a partial assignment to memory that typically includes the program that is executed and the agreed-upon input. The prover also makes public the initial and final state of the registers and the number of steps in the computation. A certificate published on blockchain establishes, modulo common cryptographic assumptions, that the prover is in possession of a full assignment to memory extending the partial one such that the program runs to completion in the given number of steps. The code is then carefully designed to ensure that this implies the claim that is of interest to the verifier. For example, to establish that a calculation yields a claimed result, the prover and verifier agree on a Cairo program that carries out the calculation, asserts that it is equal to the claimed value, and fails otherwise. A certificate that the program terminates successfully establishes the computational claim.

Reading Cairo programs takes some getting used to. An instruction like `x = y + 5` is often thought of as an assignment of the value $y + 5$ to the memory location allocated to `x`, but it is really an assertion that the prover has assigned values to the memory so that the equation holds. It is an interesting feature of the model that a Cairo program can depend on values in memory that are assigned by the prover but not made public to the verifier. For example, a program can establish that a value `x` is a perfect square by asserting that it is equal to `y * y` for some `y`, without sharing the value of `y` with the verifier.

The Cairo CPU instruction set includes a call instruction, a return instruction, conditional and unconditional jumps, an instruction to advance the allocation pointer, and instructions that make arithmetic assertions about values stored in memory. The file `cpu.lean` consists of less than 200 lines of Lean definitions that provide a formal specification of the CPU and the next state relation. The next state depends on the contents of memory, `mem`, and the values of the CPU registers, `s`, but since the memory never changes, the next state relation `next_state mem s t` need only specify the successor state `t` of the registers. If the program counter points to an assert instruction that fails, there is no successor state. If the program counter points to an ill-formed instruction, the value of `t` is nondeterministic, so verifying that a Cairo program has the intended semantics generally requires establishing that a successful run of the program does not encounter such an instruction. By convention, programs halt with a jump instruction to the to itself, that is, an implicit infinite loop. The cryptographic proof published on blockchain establishes, with high probability, that the program has reached such a state.

Our project is designed to enable users to prove that the successful execution of a program guarantees that a property of interest holds. To that end, we define the following predicate:

```
def ensures (mem : F → F) (σ : register_state F)
      (P : ℕ → register_state F → Prop) : Prop :=
  ∀ n : ℕ, ∀ exec : fin (n+1) → register_state F, is_halting_trace mem exec →
    exec 0 = σ → ∃ i : fin (n + 1), ∃ κ ≤ i, P κ (exec i)
```

This says that any sequence `exec` of states that starts with σ, proceeds according to the machine semantics with respect to the memory assignment `mem`, and ends with a halting instruction eventually reaches a step `i` along the way such that the register state `exec i` satisfies `P`. Note that the predicate `P` can also make reference to the contents of memory, so we can express that at step `i` the memory location referenced by a certain register has a certain value, or that the value of a fixed memory location has a certain property. More precisely,

the predicate `P` κ τ takes a numeric value κ as well as a register state τ, and the `ensures` predicate says that there is a value of κ less than or equal to `i` such that `P` κ (`exec i`) holds. We will explain the use of κ in Section 5.

3 From Assembly Code to Machine Code

The Cairo compiler translates code written in the Cairo language to instructions in the Cairo assembly language [14, Section 5], which are then translated to machine instructions. Assembly instructions can also be inserted directly into Cairo programs. The first step toward bridging the gap between the Cairo programming language and Cairo machine code is therefore to model the Cairo assembly language in Lean. The file `soundness/assembly.lean` in our project provides a description of Cairo machine instructions in terms of the offsets and flags, and it defines a translation from that representation to 63-bit machine code instructions. It also defines Lean notation that approximates Cairo assembly-language syntax. For example, here are three elementary mathematical Cairo functions from the Cairo common library:

```
func assert_nn{range_check_ptr}(a) {
  a = [range_check_ptr];
  let range_check_ptr = range_check_ptr + 1;
  return (); }

func assert_le{range_check_ptr}(a, b) {
    assert_nn(b - a);
    return (); }

func assert_nn_le{range_check_ptr}(a, b) {
    assert_nn(a);
    assert_le(a, b);
    return (); }
```

The first confirms that the argument `a` is the cast of a nonnegative integer less than 2^{128}, by asserting that it is equal to the value of memory at the address `range_check_ptr`, which is assumed to point to a block of elements that have been verified to have this property. The second confirms that `a` is less than or equal to `b` by calling `assert_nn`(`b` − `a`), and the third function combines the previous two properties. The curly brackets mean that the argument `range_check_ptr` is passed to, updated by, and returned implicitly by these functions. We explain range checking in more detail in Section 5.

The Cairo compiler compiles these functions to assembly code and then to machine instructions. The assembly code corresponding to `assert_nn_le` looks as follows:

```
[ap] = [fp + (-5)]; ap++
[ap] = [fp + (-4)]; ap++
call rel -11
[ap] = [fp + (-4)]; ap++
[ap] = [fp + (-3)]; ap++
call rel -11
ret
```

The details are not important. The function calls are carried out by copying the arguments, including the implicit range check pointer, to the end of the global memory used so far. The arguments are referenced relative to the frame pointer, so [`fp` + (−5)] denotes the value in memory at the address `fp` − 5. The call instructions update the program counter and frame pointer so that execution continues in the subroutine, and the return at the end restores the frame pointer and updates the program counter to the next instruction in the calling routine.

Our tool generates a Lean description of this assembly code. The notation isn't pretty; we use tick marks and funny tokens to avoid conflicting with other tokens that may be in use. For example, the Lean description of the `assert_nn_le` assembly code is as follows:

```
def starkware.cairo.common.math.code_assert_nn_le : list F := [
  'assert_eq['dst[ap] === 'res['op1[fp+ -5]];ap++].to_nat,
  'assert_eq['dst[ap] === 'res['op1[fp+ -4]];ap++].to_nat,
  'call_rel['op1[imm]].to_nat, -11,
  'assert_eq['dst[ap] === 'res['op1[fp+ -4]];ap++].to_nat,
  'assert_eq['dst[ap] === 'res['op1[fp+ -3]];ap++].to_nat,
  'call_rel['op1[imm]].to_nat, -11,
  'ret[].to_nat ]
```

As developers, we only had to read such code for debugging, and we found the notation convenient. Our Lean representation is adequate in the sense that the assembly instructions can be transformed to machine instructions, which can, in turn, be transformed to the 63-bit numeric representations which are then cast to the finite field `F`. Users can evaluate definitions like the one above in Lean and check that the resulting numeric values are the same ones produced by the Cairo compiler, and hence are the same ones used in the STARK certificate generated by the Cairo runner. Our soundness proofs start with the assumption that these values are stored in memory and that the program counter is set accordingly.

The file `soundness/assembly.lean` establishes a small-step semantics for reasoning about instructions at the assembly level. For example, variants of the Cairo `call` instruction allow specifying the address of the target in various ways, either as an absolute or relative address, which can in turn be given as an immediate value or read from memory with an offset from either the allocation pointer or frame pointer. The theorem describing the behavior of this instruction is as follows:

```
theorem next_state_call {F : Type*} [field F] (mem : F → F)
  (s t : register_state F) (op0 : op0_spec) (res : res_spec) (call_abs : bool) :
(call_instr call_abs res).next_state mem s t ↔
  (t.pc = jump_pc s call_abs (compute_res mem s (op0_spec.ap_plus 1) res) ∧
    t.ap = s.ap + 2 ∧
    t.fp = s.ap + 2 ∧
    mem (s.ap + 1) = bump_pc s res.to_op1.op1_imm ∧
    mem s.ap = s.fp)
```

Read this as follows: given that the CPU registers are in state `s` and given the contents of memory `mem`, the call instruction with boolean flag `call_abs` and operand specifications `op0` and `res` results in the new state `t`, where the program counter is updated as indicated, the relevant return address and the current frame pointer are stored in memory at the current allocation pointer, the allocation pointer is increased by two, and the frame pointer is increased by two. The details of the computations `jump_pc`, `compute_res`, and `bump_pc` are not important. What *is* important is that for concrete values of `op0`, `res`, and `call_abs`, Lean's tactics (a term rewriter, a numeric evaluator, etc.) are powerful enough to compute specific values and *prove* that the functions have those values. For example, if a specific call instruction decreases the program counter by 100, Lean can prove that the `next_state` relation holds for a suitable state `t` with `t.pc = s.pc - 100`. This allows us to reason about the behavior of a block of assembly code by stepping through each instruction in turn. The proof of the `next_state_call` and others like it are fiddly but straightforward: it is just a matter of unfolding the definitions of the assembly language instructions and then relating the resulting machine instructions to the semantics defined in `cpu.lean`.

4 From Cairo Code to Assembly Code

Consider the procedure `assert_nn_le`, which takes field elements `a` and `b` and asserts that they are casts of integers in a certain range such that the one corresponding to `a` is less than or equal to the one corresponding to `b`. More precisely, the desired specification is as follows:

```
def spec_assert_nn_le (mem : F → F) (κ : ℕ)
  (range_check_ptr a b ρ_range_check_ptr : F) : Prop :=
∃ m n : ℕ, m < rc_bound F ∧ n < rc_bound F ∧ a = ↑m ∧ b = ↑(m + n)
```

The argument `ρ_range_check_ptr` denotes the return value of `assert_nn_le`, which is implicit in the Cairo code. We often use unicode characters, which are allowed in Lean but not Cairo, to ensure that identifiers that we introduce in specifications and proofs do not clash with the identifiers that we take from Cairo. The up arrows denote casts to the field `F`. Here the value of `rc_bound F` is assumed to be 2^{128}, so m and n represent 128-bit unsigned integers. The autogenerated specification of `assert_nn_le` merely says that the Cairo code calls the two auxiliary functions `assert_nn` and `assert_le`:

```
def auto_spec_assert_nn_le (mem : F → F) (κ : ℕ)
    (range_check_ptr a b ρ_range_check_ptr : F) : Prop :=
∃ (κ₁ : ℕ) (range_check_ptr₁ : F),
  spec_assert_nn mem κ₁ range_check_ptr a range_check_ptr₁ ∧
∃ (κ₂ : ℕ) (range_check_ptr₂ : F),
  spec_assert_le mem κ₂ range_check_ptr₁ a b range_check_ptr₂ ∧
κ₁ + κ₂ + 7 ≤ κ ∧
ρ_range_check_ptr = range_check_ptr₂
```

The role of κ, κ_1, and κ_2 will be discussed in Section 5. Notice that the autogenerated specification for `assert_nn_le` refers to the *user* specifications of `assert_nn` and `assert_le` rather than the autogenerated ones. Interleaving the two types of specifications is crucial for handling recursion, since our autogenerated specification of a recursive function invokes the user specification to describe the effects of the recursive calls. We handle loops in a similar way. More importantly, our approach means that when users have to reason about the autogenerated specification, they can make use of their own specifications of the dependencies. This enables them to verify complex programs in a modular way.

With the autogenerated specification in hand, the user's task is to write their own specification of `spec_assert_nn_le` and prove that it follows from the autogenerated one. Our verification tool then uses that in the proof of the following theorem, which asserts that the machine code meets the user specification:

```
theorem auto_sound_assert_nn_le
  (range_check_ptr a b : F)
  (h_mem : mem_at mem code_assert_nn_le σ.pc)
  (h_mem_0 : mem_at mem code_assert_nn (σ.pc - 9))
  (h_mem_1 : mem_at mem code_assert_le (σ.pc - 5))
  (hin_range_check_ptr : range_check_ptr = mem (σ.fp - 5))
  (hin_a : a = mem (σ.fp - 4))
  (hin_b : b = mem (σ.fp - 3)) :
ensures mem σ (λ κ τ,
  τ.pc = mem (σ.fp - 1) ∧ τ.fp = mem (σ.fp - 2) ∧ τ.ap = σ.ap + 14 ∧
  ∃ μ ≤ κ,
    rc_ensures mem (rc_bound F) μ (mem (σ.fp - 5)) (mem (τ.ap - 1))
    (spec_assert_nn_le mem κ range_check_ptr a b (mem (τ.ap - 1))))
```

The theorem asserts that, given the contents of memory `mem` and the register state σ, if the code for `assert_nn_le` is in memory at the program counter, the code for the dependencies are in place as well, and the arguments to the function are stored in memory in the expected locations indexed by the frame pointer, then any halting computation eventually returns to the calling function (restoring the program counter and frame pointer according to the Cairo language calling conventions) and ensures that the user specification holds, assuming that certain auxiliary locations in memory have been range checked. Once again, we promise to explain range checking in Section 5.

The proof of `auto_sound_assert_nn_le` establishes the correctness of the autogenerated specification and then applies the user-supplied theorem that this implies the user's specification. Generally speaking, the user doesn't need to see the Lean description of the assembly code, the theorem `auto_sound_assert_nn_le`, or the proof of correctness. The autogenerated specifications, the user specifications, and the proof that the former imply the latter are stored in the same directory as the Cairo code. The Lean descriptions of the assembly code and the correctness proofs are kept in a separate folder, tucked out of sight.

Our verification tool has the task of extracting the autogenerated specifications and constructing the correctness proofs. The Cairo compiler, which is written in Python, produces a number of data structures that we are able to make use of once the compilation is complete. These contain, for example, a dictionary of namespaced identifiers. Whenever we needed additional information, we added hooks that, in verification mode, are called to log that information. For example, a compound assertion like `x = 3 * y + 4 * z` translates to a list of atomic assertions, and our verification tool has access to the original equation and the code points that mark the beginning and end of the list of assertions.

As we will discuss in Section 5, the Cairo language uses two sorts of variables: local variables are indexed with offset from the frame pointer, and global variables are indexed offset from the allocation pointer. When a function takes values `a b c : F` as arguments, the compiler places these values in memory just before the allocation pointer when the procedure is called. For the most part, the verifier keeps the nitty-gritty memory allocation issues hidden from the user, and mediates between variable names and the machine semantics with with equations like `a = mem` (σ`.ap − 3`). The Cairo language also allows the definition of compound structures, and we define the corresponding structures in Lean and interpret references to memory accordingly.

Our verifier uses Dijkstra's weakest preconditions [10] to read off a specification. The process is straightforward, modulo the fact that the verifier also has to construct Lean proofs that prove that these specifications are met. That requires unpacking the meaning of each machine instruction, using the theorems described in the previous section. Unpacking the `mem_at` predicate tells us which instruction is present at each memory location. We then use special-purpose tactics (small-scale automation written in Lean) to unpack the effect of each instruction. For example, suppose at a given point in the proof we need to show that executing the code at program counter σ`.pc + 5` ensures that a certain result holds, and we know that the instruction at that location corresponds to a certain instruction with an immediate value. Applying the relevant tactic leaves us the goal of showing that executing the code at program counter σ`.pc + 7` ensures the desired result, with the other registers and the proof context updated to reflect the result of executing the instruction. This reduction is justified by appealing to the meaning of the `ensures` predicate and the specification of the machine semantics. In this way, our tactics carry out a kind of symbolic execution of the assembly code, and register the effects in the proof context.

The verifier's task is then to parse each high-level Cairo instruction, generate a specification of its behavior, and construct the corresponding part of the correctness proof, which shows that the corresponding assembly instructions implement the high-level ones.

- A variable declaration like `tempvar b = a + 5` translates to an existential quantifier in the specification, $\exists$ `b`, `b` $=$ `a` $+ 5 \wedge \ldots$. The correctness proof instantiates the existential quantifier to the corresponding memory location and maintains this correspondence.
- An equality assertion in the program translates to an equality assertion in the specification. Such an assertion generally translates to one or more assembly-level assertions, and the correctness proof involves reconstructing the compound equality statement from the components.
- Cairo programs can have labels and both conditional and unconditional jumps. The corresponding machine code has the expected effect of (conditionally) modifying the program counter. In some settings, the correctness proof only needs to record this change to the program counter and continue stepping through the instructions starting at the new pc. But for handling jumps that coalesce control flow, and loops in particular, we analyze the control flow into blocks and break the specification and correctness proof up accordingly. We describe this process further below.
- A conditional jump is implicit in a Cairo `if` ... `then` ... `else` construct. This translates to a disjunction in the specification and the definition of a block where the branches flow together.
- A subroutine call to another procedure translates to an assertion, in the autogenerated specification, that the user specification of the target procedure holds of the arguments and the return value. The call instruction stores the current program pointer and frame pointer in memory and jumps to the location of the target procedure. The return instruction restores the frame pointer and jumps back to the calling procedure. The correctness proof invokes the correctness theorem for the target procedure as well as the assumption that the procedure is in memory at the expected location.

The description so far presupposes that the control flow has no cycles. To handle recursive calls and loops, we do not have to prove termination; the STARK certificate assures a skeptical verifier that the program has terminated, so we need only show that, given that fact, the specification is met. (This is commonly characterized as the difference between *partial correctness* and *total correctness*.) The claim `ensures mem` σ `P` is equivalent to $\forall$ `b`, `ensuresb b mem` σ `P`, where `ensuresb b mem` σ `P` says that every halting execution sequence from state σ with at most `b` steps eventually reaches a state that satisfies `P`. We can prove the latter by induction on `b`, generalizing over states σ with program counter pointing to the relevant code.

For functions that call themselves recursively, we modify the default user specification so that it is trivially true, and place it *before* the autogenerated specification. The autogenerated specification asserts the play-by-play description alluded to above, except that it uses the user specification to characterize the recursive calls. The user is free to write any specification they want, provided they show that the autogenerated specification implies the user specification. In short, the user has to show that the play-by-play characterization implies their own characterization, assuming their characterization holds at downstream calls. The correctness proof uses this together with the inductive hypothesis at downstream calls.

A similar method handles loops. Our verification tool begins by analyzing the control-flow graph [2], dividing the code into basic blocks, without any jumps or labels. A block starts at the beginning of a function or at a label, and ends with a jump, a return, or a flow to a label. Any block that can be entered from more than one other block receives a separate specification in the specification file, and cycles that arise in a topological sort are handled in a manner similar to recursive function calls. In practice, the user can specify that the execution has an effect conditional on an invariant holding at the entry point. Verification of the full user specification then requires showing that the invariant holds at the first entry point and that it is maintained on re-entry.

5 Memory Management and Range Checks

In this section, we discuss aspects of the Cairo programming language that stem specifically from its intended use toward verifying computations on blockchain. Encoding execution traces efficiently required keeping the machine model simple, which is why StarkWare's engineers settled on a CPU with only three registers and read-only memory. The cost of certification on blockchain scales with the number of steps in the execution trace together with the number of memory accesses. The fact that memory is read-only makes the verification task easier: we do not need to worry about processes overwriting each other's memory. The Cairo language allows procedures to declare two types of temporary variables, namely, relative to the frame pointer (fp) or allocation pointer (ap). It is a quirk of the Cairo language that references to ap-based variables can be revoked when the compiler cannot reliably track the effects of intermediate commands and function calls on the ap, for example, when different flows of control may result in different changes to the ap. Our verification relies on the compiler's internal record of its ability to track these changes.

The STARK encoding is most efficient when memory is assigned in a continuous block. The Cairo compiler is tightly coupled with a Cairo *runner*, whose task is to allocate memory and assign values to ensure that the Cairo program runs to completion. The Cairo whitepaper describes the methods that are used to simulate conventional memory models. The processor uses the frame pointer to point to the base of a procedure's local memory and the allocation pointer to point to the next available position in global memory. Local variables are kept in the same contiguous block. When one procedure calls another, the program counter and frame pointer are stored in global memory, the allocation pointer is updated, and the frame pointer is set equal to the allocation pointer. When the procedure returns, the program counter and frame pointer are restored.

For efficiency, a local procedure sometimes has to access values that are stored in global memory, which is to say, they are indexed relative to the allocation pointer. This is challenging because the allocation pointer is constantly changing. For example, when one procedure calls another, upon the return the allocation pointer may have changed. Moreover, the new value of the allocation pointer cannot always be predicted at compile time; for example, different flows of control through an if-then-else can result in different changes to its value. The compiler uses a *flow tracker* that keeps track of these changes as best it can, allowing the programmer to refer to the same global variables throughout. Our verification tool does not have to know much about how the flow tracker works, but it needs to make use of the results. For example, if the value of a variable `x` is `mem (ap + 1)` before a subroutine call and the allocation pointer `ap'` on return is equal to `ap + 3`, our Lean proofs need to use the identity `ap' = ap + 3` to translate an assertion involving `mem (ap' − 2)` into an assertion about `x`. Our verification tool claims and proves the relevant identities while stepping through the code, and uses those identities as rewriting rules when verifying assertions.

A more striking difference between programming in Cairo and programming in an ordinary programming language is that the most fundamental data type consists of values of a finite field. One can add and multiply field elements, but there is no machine instruction that compares the order of two elements. To meet high-level specifications that are stated in terms of integers, the STARK encoding uses cryptographic primitives to verify that a specified range of memory has been *range checked*, which is to say, the corresponding field elements are casts of integers in the interval $[0, 2^{128})$. Our previous verification of the STARK encoding [4] shows that the STARK certificate guarantees (with high probability) that the specified

memory locations have indeed been range checked. A Cairo program can make use of this fact by taking, as input, a pointer to a location in the block of range-checked memory, making assertions about a sequence of values at that location, and returning (in addition to its ordinary return values) an updated pointer to the next unused element. Both the user specification and the autogenerated specification are of the form "assuming the values between ... and ... have been range-checked, the following holds: ... ." These hypotheses have to be used inside the correctness proofs, to justify the assertions that particular values have been range checked. The hypotheses also have to be threaded through procedure calls and combined appropriately, so the specification of a top-level function comes with a range-check hypothesis that covers all the recursive calls. This top-level hypothesis is justified by the STARK certificate.

Our verification tool handles all this plumbing. For example, recall the Cairo function `assert_nn`, which asserts that the argument `a` is the cast of an integer in $[0, 2^{128})$.

```
func assert_nn{range_check_ptr}(a) {
  a = [range_check_ptr];
  let range_check_ptr = range_check_ptr + 1;
  return (); }
```

The curly brackets in `{range_check_ptr}` indicate that the value should implicitly be returned among the other return values. (In this case, there aren't any others.) The user-written specification of this function is as follows:

```
def spec_assert_nn (mem : F → F) (κ : ℕ)
    (range_check_ptr a ρ_range_check_ptr : F) : Prop :=
  ∃ n : ℕ, n < rc_bound F ∧ a = ↑n
```

Recall that the annotation `↑n` casts the natural number `n` to the underlying field `F`. Our verification tool generates the following specification:

```
def auto_spec_assert_nn (mem : F → F) (κ : ℕ)
    (range_check_ptr a ρ_range_check_ptr : F) : Prop :=
  a = mem (range_check_ptr) ∧
  is_range_checked (rc_bound F) a ∧
  ∃ range_check_ptr₁ : F, range_check_ptr₁ = range_check_ptr + 1 ∧
  3 ≤ κ ∧
  ρ_range_check_ptr = range_check_ptr₁
```

Here, $\texttt{range\_check\_ptr}_1$ is the updated version of `range_check_ptr`, and the last line specifies that this is the function's sole return value. Our tool detects the reference to `range_check_ptr` in the Cairo code and adds `is_range_checked (rc_bound F) a` to the autogenerated specification, generating an obligation in the correctness proof that the user does not have to see. The user has to prove that `spec_assert_nn` follows from `auto_spec_assert_nn`, but this follows immediately from the conjunct `is_range_checked (rc_bound F) a` in the autogenerated specification.

The ultimate correctness theorem is stated as follows:

```
theorem auto_sound_assert_nn
  (range_check_ptr a : F)
  (h_mem : mem_at mem code_assert_nn σ.pc)
  (hin_range_check_ptr : range_check_ptr = mem (σ.fp - 4))
  (hin_a : a = mem (σ.fp - 3)) :
ensures mem σ (λ κ τ,
  τ.pc = mem (σ.fp - 1) ∧ τ.fp = mem (σ.fp - 2) ∧ τ.ap = σ.ap + 1 ∧
  ∃ μ ≤ κ, rc_ensures mem (rc_bound F) μ (mem (σ.fp - 4)) (mem (τ.ap - 1))
    (spec_assert_nn mem κ range_check_ptr a (mem (τ.ap - 1))))
```

In this specification, the assertions $\tau.\texttt{pc} = \texttt{mem}\ (\sigma.\texttt{fp} - 1)$ and $\tau.\texttt{fp} = \texttt{mem}\ (\sigma.\texttt{fp} - 2)$ assert that the program counter and frame pointer have been restored correctly when the function returns. Our verification tool learns from the flow tracker that any path through this code updates the allocation pointer by one, and so it also establishes that fact, i.e. $\tau.\texttt{ap} = \sigma.\texttt{ap} + 1$, to make that information accessible when reasoning about procedures that call it. The `rc_ensures` clause in the conclusion says that if the block of memory between $\texttt{mem}\ (\sigma.\texttt{fp} - 4)$ and $\texttt{mem}\ (\tau.\texttt{ap} - 1)$ is range-checked then the user specification holds. (We will return to the role of μ in a moment.) Here $\sigma.\texttt{fp}$ is the value of the frame pointer when the function is called, $\tau.\texttt{ap}$ refers to the value of the allocation pointer upon return, and $\texttt{mem}\ (\sigma.\texttt{fp} - 4)$ and $\texttt{mem}\ (\tau.\texttt{ap} - 1)$ are, respectively, the location of the argument `range_check_ptr` and the return value, which is supposed to be the updated range check pointer.

We can now explain the role of κ and μ. Recall that $\texttt{mem}\ (\sigma.\texttt{fp} - 4)$ and $\texttt{mem}\ (\tau.\texttt{ap} - 1)$ are field elements. A first guess as to how to specify that the range of memory values between those two locations is range checked is to say that there is a natural number μ such that $\texttt{mem}\ (\tau.\texttt{ap} - 1) = \texttt{mem}\ (\sigma.\texttt{fp} - 4) + \uparrow\mu$ and for every $\texttt{i} < \mu$, the value in memory at address $\texttt{mem}\ (\tau.\texttt{ap} - 1) + \texttt{i}$ is range checked. But this specification is problematic: if the equation holds for some small value of μ, it also holds for μ plus the characteristic of the underlying field. Our correctness proof needs to use the fact that the total number of range-checked elements does not wrap around the finite field. We achieve this by asserting that μ is, moreover, bounded by the number of steps κ in the execution trace, which is made public in the STARK certification and is always smaller than the characteristic of the field. In the case of range checks, the bounds are handled entirely by the verifier and the user need not worry about them. But we have found that some Cairo specifications require similar reasoning about bounds on the length of the execution, and for those rare occasions, we have exposed the parameter κ in the user-facing specifications.

The virtue of our verification tool is that the user can be oblivious to most of the implementation details we have just described, such as the handling of the range check pointers and the way that variables, arguments, and return values are stored in memory. The user writes the Cairo procedure `assert_nn` and is given the specification `auto_spec_assert_nn`. The user then writes the specification `spec_assert_nn` and proves that `spec_assert_nn` follows from `auto_spec_assert_nn`. The correctness proof can be checked behind the scenes. From that moment on, `spec_assert_nn` is all that users need to know about the behavior of `assert_nn`, from the point of view of proving properties of Cairo functions that use it. In the next section, we will show that this scales to the verification of more complex programs.

6 Validating Digital Signatures

Any elliptic curve over a field of characteristic not equal to 2 or 3 can be described as the set of solutions to an equation $y^2 = x^3 + ax + b$, the so-called *affine* points, together with one additional *point at infinity*. The set of such points has the structure of an abelian group where the zero is defined to be the point at and addition between affine points defined as follows:

- To add (x, y) to itself, let $s = (3x^2 + a)/2y$, let $x' = s^2 - 2x$, and let $y' = s(x - x') - y$. Then $(x, y) + (x, y) = (x', y')$. This is known as *point doubling*.
- $(x, y) + (x, -y) = 0$, that is, the point at infinity. In other words, $-(x, y) = (x, -y)$.
- Otherwise, to add (x_0, y_0) and (x_1, y_1), let $s = (y_0 - y_1)/(x_0 - x_1)$, let $x' = s^2 - x_0 - x_1$, and let $y' = s(x_0 - x') - y_0$. Then $(x_0, y_0) + (x_1, y_1) = (x', y')$.

It is not hard to prove that with addition, negation, and zero so defined, the structure satisfies all the axioms for an abelian group other than associativity. Proving associativity is trickier, though it can be done with brute-force algebraic computations in computer algebra systems, and various approaches have been used in the interactive theorem proving literature to establish the result formally [25, 5, 12, 15, 3].

The study of elliptic curves over the complex numbers originated in the nineteenth century, where the addition law has a geometric interpretation. The topic is fundamental to contemporary number theory. Elliptic curves over a *finite field* are widely used in cryptography today, on the grounds that for any nonzero point x, the map $n \mapsto n \cdot x$ (that is, n-fold sum of x with itself) is easy to compute but, as far as we know, difficult to invert. This forms the basis for the *elliptic curve digital signature algorithm* (ECDSA). ECDSA provides a protocol by which a sender can generate a pair consisting of a public key and a private key, publish the public key, and then send messages in such a way that a receiver can verify that the message was sent by the holder of the private key and that the message has not been changed.

The Cairo library contains functions that support ECDSA over the secp256k1 elliptic curve, that is, the curve $y^2 = x^3 + 7$ over the finite field of integers modulo the prime $p = 2^{256} - 2^{32} - 977$. For reasons we will shortly explain, the calculations are subtle. We have proved the correctness of the Cairo functions implementing the elliptic curve operations efficiently, as well as a Cairo procedure for validating secp signatures. Figure 1 shows the Cairo procedure for recovering the public key of the sender from a digitally signed message, and Figure 2 shows the correctness proof that we have obtained in Lean. The rest of this section is devoted to describing the formalization and the resulting theorem.

```
func recover_public_key{range_check_ptr}(msg_hash: BigInt3,
    r: BigInt3, s: BigInt3, v: felt) -> (public_key_point: EcPoint) {
  alloc_locals;
  let (local r_point: EcPoint) = get_point_from_x(x=r, v=v);
  let (generator_point: EcPoint) = get_generator_point();
  let (u1: BigInt3) = div_mod_n(msg_hash, r);
  let (u2: BigInt3) = div_mod_n(s, r);
  let (point1) = ec_mul(generator_point, u1);
  let (minus_point1) = ec_negate(point1);
  let (point2) = ec_mul(r_point, u2);
  let (public_key_point) = ec_add(minus_point1, point2);
  return (public_key_point); }
```

Figure 1 Cairo procedure for recovering an secp public key.

```
def spec_recover_public_key (mem : F → F) (κ : ℕ)
  (range_check_ptr : F) (msg_hash r s : BigInt3 F)
  (v ρ_range_check_ptr : F) (ρ_public_key_point : EcPoint F) : Prop :=
∀ (secpF : Type) [secp_field secpF], by exactI
r ≠ ⟨0, 0, 0⟩ →
∀ ir : bigint3, ir.bounded (3 * BASE - 1) → r = ir.toBigInt3 →
∀ is : bigint3, is.bounded (3 * BASE - 1) → s = is.toBigInt3 →
∀ imsg : bigint3, imsg.bounded (3 * BASE - 1) → msg_hash = imsg.toBigInt3 →
∃ nv : ℕ, nv < rc_bound F ∧ v = ↑nv ∧
∃ iu1 iu2 : ℤ,
  iu1 * ir.val ≡ imsg.val [ZMOD secp_n] ∧
  iu2 * ir.val ≡ is.val [ZMOD secp_n] ∧
∃ ny : ℕ, ny < SECP_PRIME ∧ nv ≡ ny [MOD 2] ∧
∃ h_on_ec : @on_ec secpF _ (ir.val, ny),
∃ hpoint  : BddECPointData secpF ρ_public_key_point,
  hpoint.toECPoint =
    -(iu1 · (gen_point_data F secpF).toECPoint) +
      iu2 · ECPoint.AffinePoint ⟨ir.val, ny, h_on_ec⟩
```

Figure 2 Correctness of the digital signature validation.

To start with, in the file `elliptic_curves.lean`, we define the secp curve over an arbitrary finite field of odd characteristic, define the group operations, and prove that they form a group, modulo a proof that the group law is associative. Lean allows us to insert a `sorry` placeholder for the missing proof of associativity; this is the only `sorry` in our development. David Kurniadi Angdinata and Junyan Xu have recently verified, in Lean, that the elliptic curve law forms a group, in impressive generality [3]. This will allow us to eliminate the `sorry`.

The files `constants.cairo`, `bigint.cairo`, `field.cairo`, and `ec.cairo` implement the operations over the secp curve, culminating in an efficient procedure to carry out scalar multiplication. The main reason that the code is subtle is that it requires calculations in the field $\mathbb{Z}/p\mathbb{Z}$, where p is the secp prime, which is even larger than (and different from!) the characteristic of the field that underlies the Cairo machine model. The Cairo implementation thus represents a value x in $\mathbb{Z}/p\mathbb{Z}$ by three field elements, each of which is checked to be the cast of an integer in a certain range. We impose additional bounds and hypotheses on these representations, and ensure that they are maintained by the calculations.

In greater detail, the Cairo code defines a constant `BASE` equal to 2^{86} and a structure `BigInt3 {d0: felt, d1: felt, d2: felt}`, with the intention that the field elements `d0`, `d1`, and `d2` will always be casts of integers `i0`, `i1`, and `i2`, respectively, with absolute values in $[0, 3 \cdot \texttt{BASE})$. These are intended to represent the value $\texttt{i0} + \texttt{i1} \cdot \texttt{BASE} + \texttt{i2} \cdot \texttt{BASE}^2$. Note that the values `i0`, `i1`, and `i2`, may be larger than `BASE`, so these representations are not unique. Our specification files define a Lean structure `bigint3` := (`i0 i1 i2` : $\mathbb{Z}$), as well as a predicate `bigint3.bounded i b` that says that each of the three limbs of the `bigint3` denoted by `i` is bounded in absolute value by `b`. Our Lean verification has to mediate between at least three different representations:

- Elements x of the secp field of integers modulo the secp prime number.
- Triples (i_0, i_1, i_2) of integers, suitably bounded, that represent such elements.
- Triples of elements (d_0, d_1, d_2) of the underlying field `F` of the Cairo machine model, assumed or checked to be casts of such integers.

Field operations like addition and multiplication on the secp field correspond to addition and multiplication on the integer representations modulo the secp prime. These in turn are carried out by Cairo code on the triples of field elements, with care to ensure that the results track the corresponding operations on suitable integer representations.

An element of the secp curve consists of a pair (x, y) of elements of the secp field satisfying $y^2 = x^3 + 7$ or the special point at infinity. These are represented in the Cairo code by a structure `EcPoint` given by `x: BigInt3, y: BigInt3`, with the point at infinity represented by any pair with `x` equal to the triple $\langle 0, 0, 0\rangle$. (This works because 7 is not a square modulo the secp prime.) We therefore use the following data structure to express that `pt : EcPoint` represents a point on the curve.

```
structure BddECPointData (secpF : Type*) [field secpF] (pt : EcPoint F) :=
  (ix iy : bigint3)
  (ixbdd : ix.bounded (3 * BASE - 1))
  (iybdd : iy.bounded (3 * BASE - 1))
  (ptxeq : pt.x = ix.toBigInt3)
  (ptyeq : pt.y = iy.toBigInt3)
  (onEC : pt.x = ⟨0, 0, 0⟩ ∨ (iy.val : secpF)^2 = (ix.val : secpF)^3 + 7)
```

The specification of a procedure that takes an `EcPoint` as input generally also assumes that the `EcPoint` is equipped with such data, and the specification of a procedure that *ouputs* an `EcPoint` generally *proves* the existence of the corresponding data.

We can now explain the Cairo code in Figure 1 and the specification in Figure 2. The digital signature method used by Cairo requires that the sender and receiver agree on the elliptic curve they are using and on a message hash function. They also fix a point G on the curve that generates the group, which has a known prime order n. The sender applies a hash function to the message to obtain an integer m, and the method provides a recipe for the sender to generate a triple (r, s, v) where r and s are integers and v is an additional bit. The recipient of the message and the signature applies the hash function to obtain m as well, checks to make sure $r \neq 0$, then finds a point (r, y) on the elliptic curve by finding the residues y satisfying $y^2 = r^3 + 7$ in the secp field and choosing the one that has the same parity as v. The receiver then computes $u_1 = r^{-1} \cdot m$ and $u_2 = r^{-1} \cdot s$, where these operations take place in the group of residues modulo n. Using scalar multiplication, the receiver calculates $Q = -u_1 \cdot G + u_2 \cdot (r, y)$, a point on the elliptic curve. If the value Q matches the sender's public key, the receiver has the desired confirmation that the message m has been sent by the sender.

The procedure `recover_public_key` carries out exactly the calculation of Q. It takes as input elements `msg_hash`, `r`, `s`, and `v` in the Cairo field. In the specification, the first three are assumed to be casts of suitably bounded integers `imsg`, `ir`, and `is`. In other words, these assumptions should be guaranteed by the calling procedure. It then checks that `v` is the cast of a suitably bounded natural number `nv` (this representation is necessarily unique), and it confirms the existence of data `hpoint` representing the point $-u_1 \cdot G + u_2 \cdot (r, y)$ in the calculation above.

The implementation of the procedure requires subtle calculations and checks. The best explanation of what the intermediate calculations are supposed to achieve are given by the Lean specification files themselves. The formalization uses the library file `math.cairo`, which runs about 450 lines of code; our Lean specifications of those functions, as well as our proofs of our own specifications from the autogenerated ones, comprises about 1,150 lines of code. The secp validation procedure runs about 800 lines of Cairo code and our specification files run about 3,200 lines of Lean code, on top of about 150 lines in our definition of the elliptic

curve group. The dependency chain of `recover_public_key` consists of 24 Cairo functions, which compile to about 900 lines of assembly code, i.e. 900 field elements. Our autogenerated correctness proofs run about 7,500 lines of Lean code.

Verifying an early version of the secp code turned up two errors that were independently caught and fixed by the software engineers. The verification later turned up an error that they missed, having to do with the use of the parameter `v` in `recover_public_key`. The error, which does not allow the prover to fake a signature but does allow it to claim that a valid signature is invalid, was fixed in the next Cairo release. Beyond that, the verification provided the software engineers with welcome reassurance. Despite extensive code review, they recognized that there were a number of places where small errors may have crept into the code, and they were able to breathe a sigh of relief when the verification was complete.

7 Methodology

We have reported on the means we have developed to deal with quirks of the Cairo architecture that stem from the need to encode Cairo computations efficiently in a STARK certificate. Beyond that, many of the methods we have used are routine for software verification. But some aspects of the way we have implemented these methods are notable, since they have enabled us to put the methods to use in a production setting. In this section, we discuss some of the pragmatic choices we have made and assess their effectiveness.

It is notable that our end-to-end correctness proofs are carried out within a single foundational proof assistant. Systems such as Why3 [13], Dafny [18], and F^* [24] extract verification conditions from imperative programs, but they do not generally verify those conditions with respect to a mathematical specification of a machine model. These systems also tend to rely on automation, like SMT solvers, that has to be trusted. In contrast, all our theorems are stated in the context of a precise axiomatic foundation and the proofs are checked by a small trusted kernel, for which independent reference checkers are available. This provides a high degree of confidence that the machine code meets its high-level specifications. Similar approaches to verifying code with respect to machine semantics include MM0 [7] and [21].

Another advantage of embedding the verification in a foundational proof assistant is that the availability of an ambient mathematical library [20] means that we can make use of any mathematical concepts that are needed to make sense of the high-level specification. Our verification of the digital signature recovery algorithm required reasoning about elliptic curves, as well as dealing with bounds and casts of integers to a finite field. We were able to carry out this reasoning in the same proof assistant that we used to carry out low-level reasoning about the machine code.

It is notable that the development of our tooling did not hamper the development of the Cairo compiler or its library. When we began our project, the compiler was already being used in production, and it is still under continuous development. Requesting substantial changes to the compiler code base would have slowed our efforts, requiring not only coordination with the compiler team but also extensive code review. With our approach, we were able to work under the radar, harvesting just enough data from the compiler for us to construct our proofs. For example, we found that justifying equality assertions between compound terms did not require a detailed understanding of the process by which the compiler carried out the calculations; it was enough to simply keep track of the intermediate assertions and pass those equations to Lean's simplifier.

An alternative approach to end-to-end verification is to verify a compiler with respect to a deeply embedded semantics. This is the approach taken by CompCert [19], CakeML [17, 1], and the Bedrock project (e.g. [8]). But the Cairo language is still evolving and there is no formal specification of its semantics, even though the meaning of a Cairo program is intuitively clear in general. Our approach gives us the freedom to generate specifications with confidence that they are correct, since they are backed up by formal proof. Producing proofs of correctness at compile time avoids having to model parts of the compiler that are irrelevant to correctness, and it does not require us to find a clean separation between those parts and the ones that are. It also avoids the need to verify behaviors that don't arise in practice. For example, Cairo allows for arbitrary labels and jumps, and programmers are free to write whatever spaghetti code they want. Our verification tooling is designed to work on regular control flow graphs, and will simply fail otherwise. This leaves the decision with Cairo developers as to whether to revise their Cairo code to fit our verification model, to verify their code by hand, or to leave it formally unverified. Thus our approach provides tools that are effective in practice without dictating or constraining the language development.

Perhaps most striking is our decision to construct correctness proofs by generating Lean source code that is then elaborated and checked by the same Lean process that elaborates and checks hand-written proofs. This means that our tool automatically constructs long, complex proofs in a system that has been carefully designed to support synergetic user interaction. This may seem odd and counterproductive. But we found that the compilation process is deterministic enough to make it possible to construct these proofs, and that we could make use of similarly deterministic and predictable automation in Lean.

Moreover, we found that the approach supports an efficient workflow for the developers of the verification tool as well as for users of the tool who wish to verify their Cairo specifications. To verify a Cairo program, a user runs our verification tool on the main file, which can import other Cairo program files. Our tool calls the compiler to compile the program and then generates a Lean description of the compiled code, a Lean specification file for each Cairo source file, and proofs that the compiled code meets the specifications. By convention, the specification files, which are typically the only formal content the user needs to inspect and modify, are kept with the source files. For example, a Cairo file `foo.cairo` gives rise to a specification file `foo_spec.lean` in the same directory. The remaining files are kept in a `verification` folder in the directory containing the main Cairo source file. Compiling the files immediately after the tool is run confirms that the compiled code meets the autogenerated default specifications. Compiling them again after the user adds their own specifications to the spec files and proves that they follow from the autogenerated specifications ensures that the code meets their specifications. The correctness proofs do nothing more than apply the theorem that says that the autogenerated specification implies the user one, so if the initial correctness proofs have already been checked and Lean accepts the proofs in the user specification file, it is unlikely that the subsequent check of the correctness proofs will fail. For the application described in Section 6, compiling and checking all the files in Lean requires only a few minutes on an ordinary desktop. Compiling and checking the specification files alone, with the user specifications and correctness proofs, takes less than two minutes from scratch. In practice, the user files are checked incrementally in real time, as the user types them into the editor.

This results in a congenial workflow. After first running the verification tool, a user can compile the files in the verification folder to confirm that the specifications are well formed and the correctness proofs are valid. The user can then focus on writing the specifications and proving them correct. We have arranged it so that if the verification tool is run again

in the presence of existing specification files, we do not overwrite any of the user-supplied content. We only add or change the autogenerated specifications, as well as the arguments to the specifications when the arguments to the corresponding Cairo functions change. (The tool leaves comments in the file so that the user can see what has changed.) That way, when the Cairo code changes, the user only needs to make corresponding changes to the specifications. Moreover, when verifying another Cairo file with overlapping dependencies, one can make use of the same specifications. This has made it possible for us to verify the Cairo library one step at a time.

Our approach has also had important benefits for the development of the verification tool. We started our project by compiling simple programs, extracting Lean descriptions of the compiled code, and writing and proving specifications by hand. This helped us determine what the autogenerated specifications should look like and taught us how to construct the correctness proofs. We then simply had to write Python code that did the same thing automatically. We were able to iteratively extend the tool to handle other aspects of the Cairo language: if-then-else blocks, recursive calls, structures, loops, and so on. As we worked through files in the Cairo library, whenever we came across a feature the verifier was not equipped to handle, we could figure out how to handle the feature manually, and then extend the tool to handle that and future instances. Debugging was similarly straightforward: whenever one of our autogenerated proofs failed, we could open the file, go to the error, and use Lean's rich editor interface to inspect the proof state. Once we figured out how to repair the error manually, it was generally not hard to modify the verification tool to produce the desired behavior automatically. Our generated code is structured, commented, and readable. It slightly more verbose and formulaic than proofs one would write by hand, but it is otherwise similar.

In sum, formal verification requires a synergetic combination of automation and user interaction. One of our most important findings is that using automation to generate formal content that can be inspected and modified interactively is a remarkably powerful and effective means to that end.

8 Conclusions and Related Work

We have presented a means of verifying functional correctness of programs written in the Cairo language with respect to a low-level machine model, and we have demonstrated its practical use with a case study, in which we have verified a Cairo library procedure for validating cryptographic signatures. We have similarly verified other fundamental components of the Cairo library, including a procedure that Cairo programmers can use to simulate the behavior of read-write dictionaries in Cairo's read-only memory model [14, Section 8.5.2].

In Section 7, we have already cited some other approaches toward verifying a functional specification down to machine code, and in Section 6, we cited various formalizations of the associativity of the group law for elliptic curves. In recent years, there has been extensive work on verification of cryptographic primitives [11, 22, 23, 26], including the kind of digital signature recovery described here. As we have explained, however, verification of Cairo programs requires dealing with specific features of the language and machine model, and it is notable that we have achieved end-to-end verification in a foundational proof assistant. Because Cairo programs are used extensively to carry out financial transactions, and because they are carefully optimized to reduce the cost of on-chain verification, having workable means of verifying their correctness is essential.

References

1 Oskar Abrahamsson, Son Ho, Hrutvik Kanabar, Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Yong Kiam Tan. Proof-producing synthesis of CakeML from monadic HOL functions. *J. Autom. Reason.*, 64(7):1287–1306, 2020. doi:10.1007/s10817-020-09559-8.

2 Frances E. Allen. Control flow analysis. *SIGPLAN Notices.*, 5(7):1–19, July 1970. doi:10.1145/390013.808479.

3 David Kurniadi Angdinata and Junyan Xu. An elementary formal proof of the group law on weierstrass elliptic curves in any characteristic. In Adam Naumowicz and René Thiemann, editors, *Interactive Theorem Proving (ITP) 2023*, 2023.

4 Jeremy Avigad, Lior Goldberg, David Levit, Yoav Seginer, and Alon Titelman. A verified algebraic representation of Cairo program execution. In Andrei Popescu and Steve Zdancewic, editors, *Certified Programs and Proofs (CPP) 2022*, pages 153–165. ACM, 2022. doi:10.1145/3497775.3503675.

5 Evmorfia-Iro Bartzia and Pierre-Yves Strub. A formal library for elliptic curves in the Coq proof assistant. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving (ITP) 2014*, pages 77–92. Springer, 2014. doi:10.1007/978-3-319-08970-6_6.

6 Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptol. ePrint Arch.*, 2018:46, 2018. URL: http://eprint.iacr.org/2018/046.

7 Mario Carneiro. Metamath Zero: Designing a theorem prover prover. In Christoph Benzmüller and Bruce R. Miller, editors, *Intelligent Computer Mathematics (CICM) 2020*, pages 71–88. Springer, 2020. doi:10.1007/978-3-030-53518-6_5.

8 Adam Chlipala. The Bedrock structured programming system: combining generative metaprogramming and hoare logic in an extensible program verifier. In Greg Morrisett and Tarmo Uustalu, editors, *International Conference on Functional Programming (ICFP) 2013*, pages 391–402. ACM, 2013. doi:10.1145/2500365.2500592.

9 Leonardo Mendonça 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, *Conference on Automated Deduction (CADE) 2015*, pages 378–388. Springer, Berlin, 2015. doi:10.1007/978-3-319-21401-6_26.

10 Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975. doi:10.1145/360933.360975.

11 Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic: With proofs, without compromises. *ACM SIGOPS Oper. Syst. Rev.*, 54(1):23–30, 2020. doi:10.1145/3421473.3421477.

12 Andrew Erbsen. *Crafting Certified Elliptic Curve Cryptography Implementations in Coq*. PhD thesis, Massachusetts Institute of Technology, 2017.

13 Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — Where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *European Symposium on Programming (ESOP) 2013: Programming Languages and Systems*, pages 125–128. Springer, 2013. doi:10.1007/978-3-642-37036-6_8.

14 Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo — a Turing-complete STARK-friendly CPU architecture. Cryptology ePrint Archive, Report 2021/1063, 2021. URL: https://ia.cr/2021/1063.

15 Thomas C. Hales and Rodrigo Raya. Formal proof of the group law for edwards elliptic curves. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *International Joint Conference on Automated Reasoning (IJCAR) 2020*, pages 254–269. Springer, 2020. doi:10.1007/978-3-030-51054-1_15.

16 StarkWare Industries. Cairo. https://www.cairo-lang.org/.

17 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *Principles of*

Programming Languages (POPL) 2014, pages 179–192. ACM, 2014. doi:10.1145/2535838.2535841.

18 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) 2010- 16th*, pages 348–370. Springer, 2010. doi:10.1007/978-3-642-17511-4_20.

19 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. doi:10.1145/1538788.1538814.

20 The mathlib community. The Lean mathematical library. In Jasmin Blanchette and Catalin Hritcu, editors, *Certified Programs and Proofs (CPP) 2020*, pages 367–381. ACM, 2020. doi:10.1145/3372885.3373824.

21 Magnus O. Myreen, Konrad Slind, and Michael J. C. Gordon. Extensible proof-producing compilation. In Oege de Moor and Michael I. Schwartzbach, editors, *Compiler Construction (CC) 2009*, pages 2–16. Springer, 2009. doi:10.1007/978-3-642-00722-4_2.

22 Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella Béguelin. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *IEEE Symposium on Security and Privacy (SP) 2020*, pages 983–1002. IEEE, 2020. doi:10.1109/SP40000.2020.00114.

23 Peter Schwabe, Benoît Viguier, Timmy Weerwag, and Freek Wiedijk. A Coq proof of the correctness of X25519 in TweetNaCl. In *Computer Security Foundations Symposium (CSF) 2021*, pages 1–16. IEEE, 2021. doi:10.1109/CSF51468.2021.00023.

24 Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. *J. Funct. Program.*, 23(4):402–451, 2013. doi:10.1017/S0956796813000142.

25 Laurent Théry. Proving the group law for elliptic curves formally. Technical Report RT-0330, INRIA, 2007. URL: https://hal.inria.fr/inria-00129237.

26 Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Conference on Computer and Communications Security (CCS) 2017*, pages 1789–1806. ACM, 2017. doi:10.1145/3133956.3134043.

No Unification Variable Left Behind: Fully Grounding Type Inference for the HDM System

Roger Bosman
KU Leuven, Belgium

Georgios Karachalias
Tweag, Paris, France

Tom Schrijvers
KU Leuven, Belgium

Abstract

The Hindley-Damas-Milner (HDM) system provides polymorphism, a key feature of functional programming languages such as Haskell and OCaml. It does so through a type inference algorithm, whose soundness and completeness have been well-studied and proven both manually (on paper) and mechanically (in a proof assistant). Earlier research has focused on the problem of inferring the type of a top-level expression. Yet, in practice, we also may wish to infer the type of subexpressions, either for the sake of elaboration into an explicitly-typed target language, or for reporting those types back to the programmer. One key difference between these two problems is the treatment of underconstrained types: in the former, unification variables that do not affect the overall type need not be instantiated. However, in the latter, instantiating all unification variables is essential, because unification variables are internal to the algorithm and should not leak into the output.

We present an algorithm for the HDM system that explicitly tracks *the scope* of all unification variables. In addition to solving the *subexpression type reconstruction* problem described above, it can be used as a basis for elaboration algorithms, including those that implement elaboration-based features such as type classes. The algorithm implements input and output contexts, as well as the novel concept of *full contexts*, which significantly simplifies the state-passing of traditional algorithms. The algorithm has been formalised and proven sound and complete using the Coq proof assistant.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Software and its engineering → Correctness

Keywords and phrases type inference, mechanization, let-polymorphism

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.8

Supplementary Material *Software (Source Code)*: `github.com/rogerbosman/hdm-fully-grounding` archived at `swh:1:dir:32c6b22d2de66001bf4c0ab5e255481de6561daa`

Funding This work was partly funded by KU Leuven project C14/20/079#55685055.

Acknowledgements We would like to thank Steven Keuchel for their help and insights about Coq, and their comments about a draft of this paper.

1 Introduction

Classic unification-based type inference algorithms for the Hindley–Damas–Milner (HDM) system such as algorithm $\mathcal{W}$ [7] solve the *type inference problem*. That is, they determine whether programs that lack type signatures are well-typed or not, by assigning every subterm the most general type possible (an unconstrained unification variable) and solving any type constraints that arise. Programs are well-typed if and only if all constraints can be solved.

However, depending on the setting, we would like to not only verify that a program is well-typed but also determine the type of every subterm. The canonical example of this is elaboration to System F [13, 19], but the problem arises in other settings as well. For


© Roger Bosman, Georgios Karachalias, and Tom Schrijvers;
licensed under Creative Commons License CC-BY 4.0

example, to aid development, real-world implementations of programming languages often allow developers to query the types of subterms, either via a REPL[1] like GHCi [22] or in a GUI-based editor, for example by supporting [24, 17] the Language Server Protocol [4]. We name this the *subterm type reconstruction problem*.

An important way the type inference and subterm type reconstruction problem differ is in how they treat underconstrained types (i.e. types with unconstrained parts). Consider the following program below.

$\textbf{let}\ x = (\lambda f.\ \texttt{unit})\ (\lambda y.\ y)\ \textbf{in}\ \ldots$

Observe that the type of y is not subject to any constraints: since $\lambda y.y$ is passed to a function that discards its argument and instead returns `unit`, it is never applied an argument, nor is its output used, either which would impose constraints. For type checking this unconstrained type is not a problem: the program is well-typed regardless of y's type. However, this situation is problematic for subterm type reconstruction, because we need to provide types for both f and y. We may only output *fully ground* types; unification variables are internal to the algorithm and should not be returned. Thus, to ground these types, we must instantiate all remaining unification variables. Generally, there are two options: (1) to generalise over the remaining variables, or (2) to default them to an arbitrary type (e.g. `Unit`).

(1) $\textbf{let}\ x = \Lambda a.\ (\lambda f : a \to a.\ \texttt{unit})\ (\lambda y : a.\ y)\ \textbf{in}\ \ldots$

(2) $\textbf{let}\ x = (\lambda f : \texttt{Unit} \to \texttt{Unit}.\ \texttt{unit})\ (\lambda y : \texttt{Unit}.\ y)\ \textbf{in}\ \ldots$

Crucially, the type of the overall expression may not determine the instantiation, as type variables may not occur in this type. Consider again the example above. Since $(\lambda f.\ \texttt{unit})\ (\lambda y.\ y)$ beta-reduces to `unit`, x's type is `Unit`. Hence, the type of y does not occur in x's type. Therefore, additional machinery is needed to keep track of unsolved unification variables and apply whichever *grounding strategy* has been chosen. While solutions to this problem are not necessarily complicated in practice, implementations are often ad hoc, making reasoning about their correctness hard.

In this paper, we address this very issue. We present algorithm $\mathcal{R}$, a *fully grounding* type inference algorithm for the HDM system. The algorithm explicitly tracks the scope of unification variables, which allows for fully grounding type inference, meaning we can infer fully ground types for all subexpressions. Since type grounding is internal to algorithm $\mathcal{R}$, its correctness proof (which we have mechanised in the Coq proof assistant [23]) carries over to the grounding strategy as well. As far as we know, we are the first to mechanically formalize a type inference algorithm for the HDM system that includes type grounding.

The algorithm utilizes in- and output contexts in the style of Dunfield and Krishnaswami [10] as well as a novel approach to unification, using a concept we dub *full contexts*. Here, contexts always contain all existing unification variables. Traditionally, inference algorithm thread through a substitution to reflect equalities found during unification in other branches of the derivation. With our approach, we avoid this threading: when an equality $\alpha := \tau$ is found, α can immediately be substituted for τ in the current context. Since the context is full, no further occurrences of α exist, and the equality can be discharged in one go.

In summary, the specific contributions of this paper are:

- This paper presents a new, *fully grounding* type inference algorithm $\mathcal{R}$ for ML-style polymorphism. The algorithm keeps track of all unification variables and their scope and uses the novel concept of full contexts to discharge all unifications in one go.

[1] Read–Eval–Print Loop

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathcal{W}} e_1 : \tau, \theta_1 \\ \theta_1\Gamma \vdash_{\mathcal{W}} e_2 : \tau_2, \theta_2 \\ a \# \theta_2\theta_1\Gamma \\ \theta_3 = \mathsf{unify}(\theta_2\tau \sim \tau_2 \to a)\end{array}}{\Gamma \vdash_{\mathcal{W}} e_1 e_2 : \theta_3 a, \theta_3\theta_2\theta_1}\ \mathcal{W}\text{-App} \qquad \frac{\Gamma \vdash_{\mathcal{W}} e : \tau, \theta \quad \overline{a} = fv(\tau) \setminus fv(\theta\Gamma)}{\Gamma \vdash_{\mathcal{W}} e : \forall \overline{a}.\tau, \theta}\ \mathcal{W}\text{-Gen}$$

$$\frac{\begin{array}{c}\Psi_{in} \vdash e_1 : [A_1]T \dashv \Psi_1 \\ \Psi_1; \{[A_1]T\} \vdash e_2 : [A_2]T_1 \dashv \Psi_2; \{[A_1']T'\} \\ \widehat{\alpha} \# \Psi_2; (A_1', A_2) \\ \Psi_2; (A_1', A_2, \widehat{\alpha}); \{\widehat{\alpha}\} \vdash T' \sim T_1 \to \widehat{\alpha} \dashv \Psi_{out}; A_{out}; \{T_{out}\}\end{array}}{\Psi_{in} \vdash e_1\ e_2 : [A_{out}]T_{out} \dashv \Psi_{out}}\ \textsc{App} \qquad \frac{\Psi_{in} \vdash e : [A]T \dashv \Psi_{out} \quad \mathsf{gen}(T, A) = S}{\Psi_{in} \vdash e : S \dashv \Psi_{out}}\ \textsc{Gen}$$

Figure 1 Application and generalisation of algorithm $\mathcal{W}$ (top) and algorithm $\mathcal{R}$ (bottom).

- We have mechanised both algorithm $\mathcal{R}$ as well as its correctness proof in the Coq proof assistant. Since algorithm $\mathcal{R}$ is fully grounding, we are – to our knowledge – the first to mechanically prove the correctness of an inference algorithm that features grounding. We admit an axiom about unification (see Section 5.3) and about the declarative specification (see Section 6.3).

2 Overview

This section describes the difference between unification-based algorithms like algorithm $\mathcal{W}$ and our algorithm $\mathcal{R}$. We first describe how algorithm $\mathcal{W}$ loses track of unconstrained type variables. We then propose our algorithm $\mathcal{R}$, which explicitly tracks the scope of unification variables, and show how this information yields fully grounding type inference.

Algorithm $\mathcal{W}$

Unification-based algorithms like algorithm $\mathcal{W}$ derive equality constraints at application sites $e_1\ e_2$. Rule $\mathcal{W}$-App of Figure 1 describes algorithm $\mathcal{W}$ in the case of applications.

Let us apply this to the example $(\lambda f.\ \texttt{unit})\ (\lambda y.\ y)$ (shown in Section 1) under an empty context. First, we infer the type $a_1 \to \texttt{Unit}$ for $\lambda f.\ \texttt{unit}$. Then, we infer the type $a_2 \to a_2$ for $(\lambda y.\ y)$. Both steps result in empty unifiers θ_1, θ_2. Then, with a_3 fresh, we unify $a_1 \to \texttt{Unit}$ with $(a_2 \to a_2) \to a_3$, yielding $\theta = (a_3 := \texttt{Unit}, a_1 := a_2 \to a_2)$. Finally, we return $\theta(a_3)$, which equates to `Unit`. Since algorithm $\mathcal{W}$ only returns the function's result type `Unit`, it loses track of free variables that only occur in the parameter's type (i.e. a_2). As a_2 is no longer reachable, it will not be further constrained and will remain unsolved.

Algorithm $\mathcal{W}$'s generalisation logic, extracted as $\mathcal{W}$-Gen[2] in Figure 1, turns an expression's monotype into a type scheme. In our running example, since the monotype `Unit` does not contain any free variables, algorithm $\mathcal{W}$ generalises over the empty list, which simply yields the `Unit` type scheme. Observe in particular that the unsolved unification variable a_2 is not generalised over. Hence, the type for $\lambda y.\ y$ remains $a_2 \to a_2$, but we do not know in which context a_2 is defined, and whether or where it can be generalised.

[2] Normally, this logic would be incorporated as part of the rule for let expressions.

x	Variables		a	Skolem type variables
Terms	e	$::=$	$x \mid \texttt{unit} \mid \lambda x.e \mid e_1\ e_2 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2$	
Monotypes	τ	$::=$	$a \mid \texttt{Unit} \mid \tau_1 \to \tau_2$	
Type Schemes	σ	$::=$	$\tau \mid \forall a.\sigma$	
Scoping/Typing Context	Γ	$::=$	$\bullet \mid \Gamma; a \mid \Gamma; x : \sigma$	

Figure 2 Syntax of the Declarative Specification.

Algorithm $\mathcal{R}$

Our algorithm solves this problem by not only inferring the type T of an expression but also a list of unification variables that are in scope for T. By "in scope" we mean those variables that are safe to generalise over. Note that this list need not be a superset or subset of the free unification variables of T. We denote unification variables as $\widehat{\alpha}$ with A denoting a list of $\widehat{\alpha}$. Furthermore, we use $[A]T$ to denote the type T having A in scope.

Algorithm $\mathcal{R}$ utilises in- and output contexts [10] as well as the notion of *full contexts* to avoid having to pass around unifiers θ. We postpone fully introducing algorithm $\mathcal{R}$ to Section 4.2. For now, we present an informal preview of the application of algorithm $\mathcal{R}$ to the same example as covered above, highlighting how algorithm $\mathcal{R}$ infers fully ground types, and showing the benefit of full contexts.

For the application $(\lambda f.\ \texttt{unit})\ (\lambda y.\ y)$, in App, we first derive the type $[\widehat{\alpha}_1](\widehat{\alpha}_1 \to \texttt{Unit})$ for $\lambda f.\ \texttt{unit}$. Then, we infer the type $[\widehat{\alpha}_2](\widehat{\alpha}_2 \to \widehat{\alpha}_2)$ for $\lambda y.\ y$. Here our notion of full contexts comes in: instead of deriving a unifier that needs to be applied to the type of $\lambda f.\ \texttt{unit}$, we instead append $[\widehat{\alpha}_1](\widehat{\alpha}_1 \to \texttt{Unit})$ to the input context of the inference on $\lambda y.\ y$, and obtain a possibly further instantiated $[A_1']T'$ from the output context (as seen in rule App in Figure 1). Here, $[A_1']T' = [\widehat{\alpha}_1](\widehat{\alpha}_1 \to \texttt{Unit})$.

With $\widehat{\alpha}_3$ fresh, we unify $[\widehat{\alpha}_1](\widehat{\alpha}_1 \to \texttt{Unit})$ with $[\widehat{\alpha}_3, \widehat{\alpha}_2]((\widehat{\alpha}_2 \to \widehat{\alpha}_2) \to \widehat{\alpha}_3)$. Again, we apply our notion of full contexts, appending all variables in scope for our to-be-unified types ($[\widehat{\alpha}_3, \widehat{\alpha}_1, \widehat{\alpha}_2]$) to the context, allowing us to retrieve a possibly further instantiated A_{out} from the output context. Here, $A_{out} = [\widehat{\alpha}_2]$. Furthermore, we append $\widehat{\alpha}_3$ once more, now occurring as a type instead of an in-scope unification variable. Since $\widehat{\alpha}_3$ enjoys any substitution occurring during unification, we obtain the possibly further instantiated T_{out} from the output context. Here, $T_{out} = \texttt{Unit}$. Finally, we return $[\widehat{\alpha}_2]\texttt{Unit}$. Observe that, even though we are dropping the argument type, we are **not dropping the variables in scope of the argument type**. Generalisation, as displayed in Gen, of type $[\widehat{\alpha}_2]\texttt{Unit}$ is (almost) trivial.

To conclude this section, we have shown that our algorithm $\mathcal{R}$ not only infers a type T, but also a list of type variables A in scope for T. This way it can infer a fully ground type for every subterm. The following sections formally introduce algorithm $\mathcal{R}$.

3 Declarative System

Before we present our algorithm, we present the declarative system that serves as its specification. The declarative system is essentially the syntax-directed system of Clement et al. [6], with two changes. First, like System F [13, 19], we explicitly track type variables in an ordered context. Consequently, we only generalise over variables that occur at the end of the context (i.e., not occurring to the left of term variable bindings). The second change is a purely syntactic one: we have extracted generalisation into a separate judgment.

$\boxed{\Gamma \Vdash_{\texttt{mono}} e : \tau}$ $\boxed{\Gamma \Vdash_{\texttt{poly}} e : \sigma}$ Term Typing

$$\frac{(x : \sigma) \in \Gamma \qquad \Gamma \Vdash \sigma \geq \tau}{\Gamma \Vdash_{\texttt{mono}} x : \tau}\ \textsc{TmVar} \qquad \frac{}{\Gamma \Vdash_{\texttt{mono}} \texttt{unit} : \texttt{Unit}}\ \textsc{TmUnit}$$

$$\frac{\Gamma \Vdash_{\texttt{ty}} \tau_1 \quad \Gamma; x : \tau_1 \Vdash_{\texttt{mono}} e : \tau_2}{\Gamma \Vdash_{\texttt{mono}} \lambda x.e : \tau_1 \to \tau_2}\ \textsc{TmAbs} \qquad \frac{\Gamma \Vdash_{\texttt{mono}} e_1 : \tau_1 \to \tau_2 \quad \Gamma \Vdash_{\texttt{mono}} e_2 : \tau_1}{\Gamma \Vdash_{\texttt{mono}} e_1\ e_2 : \tau_2}\ \textsc{TmApp}$$

$$\frac{\Gamma \Vdash_{\texttt{poly}} e_1 : \sigma \quad \Gamma; x : \sigma \Vdash_{\texttt{mono}} e_2 : \tau}{\Gamma \Vdash_{\texttt{mono}} (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) : \tau}\ \textsc{TmLet} \qquad \frac{\overline{a} \# \Gamma \qquad \Gamma; \overline{a} \Vdash_{\texttt{mono}} e : \tau \quad \mathsf{gen}(\tau, \overline{a}) = \sigma}{\Gamma \Vdash_{\texttt{poly}} e : \sigma}\ \textsc{TmGen}$$

$\boxed{\Gamma \Vdash_{\texttt{ty}} \sigma}$ Type Well-formedness

$$\frac{a \in \Gamma}{\Gamma \Vdash_{\texttt{ty}} a} \qquad \frac{}{\Gamma \Vdash_{\texttt{ty}} \texttt{Unit}} \qquad \frac{\Gamma \Vdash_{\texttt{ty}} \tau_1 \qquad \Gamma \Vdash_{\texttt{ty}} \tau_2}{\Gamma \Vdash_{\texttt{ty}} \tau_1 \to \tau_2} \qquad \frac{\Gamma; a \Vdash_{\texttt{ty}} \sigma}{\Gamma \Vdash_{\texttt{ty}} \forall a.\sigma}$$

$\boxed{\texttt{wf}(\Gamma)}$ Scoping/Typing Context Well-formedness

$$\frac{\texttt{wf}(\Gamma)}{\texttt{wf}(\Gamma; \bullet)} \qquad \frac{\texttt{wf}(\Gamma) \qquad a \notin \Gamma}{\texttt{wf}(\Gamma, a)} \qquad \frac{\texttt{wf}(\Gamma) \qquad \Gamma \Vdash_{\texttt{ty}} \sigma}{\texttt{wf}(\Gamma; x : \sigma)}$$

$\boxed{\Gamma \Vdash \sigma_1 \geq \sigma_2}$ Type Subsumption

$$\frac{}{\Gamma \Vdash \tau \geq \tau} \qquad \frac{a \# \Gamma \qquad \Gamma; a \Vdash \sigma_1 \geq \sigma_2}{\Gamma \Vdash \sigma_1 \geq \forall a.\sigma_2} \qquad \frac{\Gamma \Vdash_{\texttt{ty}} \tau_1 \qquad \Gamma \Vdash [\tau_1/a]\sigma \geq \tau_2}{\Gamma \Vdash \forall a.\sigma \geq \tau_2}$$

Figure 3 Typing of the Declarative Specification.

3.1 Syntax

Figure 2 displays the syntax of the declarative system. The terms and types are as given by Damas and Milner [7]. Terms consist of term variables, `unit` values, lambda abstractions, applications, and let-bindings. Type schemes are in Skolem normal form, consisting of a number of quantifiers in front of a monotype. Finally, contexts Γ track the scope of type and term variables that are in scope of an expression.

3.2 Typing

Figure 3 displays the typing rules of our declarative system. As stated, we have extracted the generalisation logic in a separate judgment, giving rise to both a monomorphic typing judgment $\Gamma \Vdash_{\texttt{mono}} e : \tau$, and a polymorphic judgment $\Gamma \Vdash_{\texttt{poly}} e : \sigma$, the latter of which is exclusively used in the typing rule for let-bindings TmLet. Rule TmGen uses the auxiliary function $\mathsf{gen}(\tau, \overline{a})$, which generalises the passed τ over the passed $\overline{a}$ in the usual way. The type- and context well-formedness judgments $\Gamma \vdash_{\texttt{ty}} \sigma$ and $\texttt{wf}(\Gamma)$ are standard. Finally, rule TmVar uses type subsumption [7, 6] to instantiate a type scheme. Since subsumption is only used in this manner, we could have given it the signature $\Gamma \Vdash \sigma \geq \tau$ and omitted the middle rule. Yet, the advantage of the subsumption rules in Figure 3 is that subsumption

x Variables a Skolem type variables $\widehat{\alpha}$ Existential type variables

Terms	e	::=	$x \mid \texttt{unit} \mid \lambda x.e \mid e_1\ e_2 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2$
Monotypes	T	::=	$a \mid \widehat{\alpha} \mid \texttt{Unit} \mid T_1 \to T_2$
Type Schemes	S	::=	$T \mid \forall a.S$
Local Existential Context	A	::=	$\bullet \mid A, \widehat{\alpha}$
Scoping/Typing Context	Ψ	::=	$\bullet \mid \Psi; a \mid \Psi; A \mid \Psi; x : S \mid \Psi; \{[A]S\}$
Type Equalities	E	::=	$\bullet \mid T_1 \sim T_2, E$
	$\{S\}$	$\doteq$	$\{[\bullet]S\}$

Figure 4 Syntax of Algorithm $\mathcal{R}$.

proofs can be done in multiple parts and combined using transitivity.

4 Algorithmic System

We now introduce algorithm $\mathcal{R}$. We discuss its syntax, rules and unification algorithm.

4.1 Syntax

Figure 4 displays the syntax used by algorithm $\mathcal{R}$. Observe that we now have two kinds of type variables: like our declarative system we have (Skolem) type variables representing types generalised over by a type scheme. We have added unification variables $\widehat{\alpha}$, which we refer to as existential type variables. Like Skolem type variables they are placeholders which can be substituted for other types. Accordingly, monotypes T may now also take the form of an existential type variable.

Contexts Ψ differ from their declarative counterparts in two significant ways. First, besides Skolem type variables, contexts also track the scope of existential type variables, similar to [10, 30]. However, unlike Skolem type variables, they are not simply appended as individual variables, but instead come in a list-like structure A. As unification may both split and solve existential type variables, reasoning about ranges of existential type variables traditionally [10] requires adding markers to the context. By putting them in a list we obtain the same reasoning power, without having to add explicit markers.

Secondly, types with their list of existential variables in scope may live in the context as an *invisible* object $\{[A]S\}$. These invisible objects, when combined with input and output contexts, are the essence behind *full contexts*, which we already introduced in Section 2. These allow us to append As and Ss on the context in branches of the inference algorithm that normally would not have them in scope. Invisible objects are invisible to membership $\in$, but visible to both substitution and fresh variable generation $\#$.

4.2 Inference algorithm

Figure 5 shows the rules of algorithm $\mathcal{R}$. Its main judgments feature in and output contexts, where the output context consists of the input context subjected to all unifications made in the derivation, which means sequences A may shrink or grow and substitutions may be made, but their basic structure is the same.

Rule Var looks up a variable in the context, and instantiates polytype S to $[A]T$ using instantiation, discussed below. Rule Unit is trivial.

$\boxed{\Psi_{in} \vdash e : [A]T \dashv \Psi_{out}}$ Type Inference

$$\frac{(x : S) \in \Psi \qquad \Psi \vdash S \geq [A]T}{\Psi \vdash x : [A]T \dashv \Psi}\ \textsc{Var} \qquad \frac{}{\Psi \vdash \texttt{unit} : [\bullet]\texttt{Unit} \dashv \Psi}\ \textsc{Unit}$$

$$\frac{\widehat{\alpha}\#\Psi_{in} \quad \Psi_{in}; \widehat{\alpha}; x : \widehat{\alpha} \vdash e : [A_2]T_2 \dashv \Psi_{out}; A_1; x : T_1}{\Psi_{in} \vdash \lambda x.e : [A_1, A_2](T_1 \to T_2) \dashv \Psi_{out}}\ \textsc{Abs} \qquad \frac{\Psi_{in} \vdash e_1 : S \dashv \Psi \quad \Psi; x : S \vdash e_2 : [A]T \dashv \Psi_{out}; x : S'}{\Psi_{in} \vdash (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) : [A]T \dashv \Psi_{out}}\ \textsc{Let}$$

$$\frac{\Psi_{in} \vdash e_1 : [A_1]T \dashv \Psi_1 \qquad \Psi_1; \{[A_1]T\} \vdash e_2 : [A_2]T_1 \dashv \Psi_2; \{[A_1']T'\} \quad \widehat{\alpha}\#\Psi_2; (A_1', A_2) \qquad \Psi_2; (A_1', A_2, \widehat{\alpha}); \{\widehat{\alpha}\} \vdash T' \sim T_1 \to \widehat{\alpha} \dashv \Psi_{out}; A_{out}; \{T_{out}\}}{\Psi_{in} \vdash e_1\ e_2 : [A_{out}]T_{out} \dashv \Psi_{out}}\ \textsc{App}$$

$\boxed{\Psi_{in} \vdash e : S \dashv \Psi_{out}}$ Generalization

$$\frac{\Psi_{in} \vdash e : [A]T \dashv \Psi_{out} \qquad \mathsf{gen}(T, A) = S}{\Psi_{in} \vdash e : S \dashv \Psi_{out}}\ \textsc{Gen}$$

$\boxed{\Psi \vdash_{\texttt{ty}} S}$ Type Well-formedness

$$\frac{a \in \Psi}{\Psi \vdash_{\texttt{ty}} a} \qquad \frac{\widehat{\alpha} \in \Psi}{\Psi \vdash_{\texttt{ty}} \widehat{\alpha}} \qquad \frac{}{\Psi \vdash_{\texttt{ty}} \texttt{Unit}} \qquad \frac{\Psi \vdash_{\texttt{ty}} T_1 \qquad \Psi \vdash_{\texttt{ty}} T_2}{\Psi \vdash_{\texttt{ty}} T_1 \to T_2} \qquad \frac{\Psi; a \vdash_{\texttt{ty}} S}{\Psi \vdash_{\texttt{ty}} \forall a.S}$$

$\boxed{\texttt{wf}(\Psi)}$ Scoping/Typing Context Well-formedness

$$\frac{\texttt{wf}(\Psi)}{\texttt{wf}(\Psi; \bullet)} \qquad \frac{\texttt{wf}(\Psi) \qquad A\#\Psi}{\texttt{wf}(\Psi, A)} \qquad \frac{\texttt{wf}(\Psi) \qquad \Psi \vdash_{\texttt{ty}} S}{\texttt{wf}(\Psi; x : S)} \qquad \frac{\texttt{wf}(\Psi; A) \qquad \Psi; A \vdash_{\texttt{ty}} T}{\texttt{wf}(\Psi; \{[A]T\})}$$

$\boxed{\Psi \vdash S \geq [A]T}$ Polymorphic Type Instantiation

$$\frac{}{\Psi \vdash T \geq [\bullet]T}\ \textsc{InstMono} \qquad \frac{\widehat{\alpha}\#\Psi \qquad \Psi; (\widehat{\alpha}) \vdash [\widehat{\alpha}/a]S \geq [A]T}{\Psi \vdash \forall a.S \geq [(\widehat{\alpha}), A]T}\ \textsc{InstPoly}$$

Figure 5 Typing of Algorithm $\mathcal{R}$.

While Abs may visually look different from conventional abstraction typing rules, it follows the same approach, with added machinery to derive the list of existential type variables in scope. Term variable x is assigned a fresh existential variable $\widehat{\alpha}$; this assignment is added to the context as well as (the singleton list) $\widehat{\alpha}$. We utilize full contexts to let $[\widehat{\alpha}]\widehat{\alpha}$ enjoy any unifications made during the recursive inference by appending them to the input context and obtaining the possibly further instantiated $[A_1]T_1$ from the output context.

Rule App, as already discussed in Section 2, first infers a type $[A_1]T$ for e_1. Inference proceeds on e_2, with the input environment extended with $[A_1]T$, by using an invisible object. By usage of this invisible object we ensure that we can safely extend the context with $[A_1]T$, because it does not bring either A_1 or T into scope. We now unify e_1's type with a function consisting of e_2's type as argument, and fresh variable $\widehat{\alpha}$ as result. We do so under an environment extend with all existential variables in scope for both types being unified, as well as $\widehat{\alpha}$, occurring as a type, instead of an in-scope variable. For this second occurrence of $\widehat{\alpha}$ we again use an invisible object, which avoids us bringing $\widehat{\alpha}$ into scope twice. We obtain

the results from the unification's output.

Rule Gen, as already discussed in Section 2, is (almost) trivial: based on the recursive, monomorphic inference, we generalise T over A. Note that we do not derive a list of variables in scope of S: since we generalise over all existential variables in scope, this list would always be empty. Finally, we have rule Let, which first infers a polytype using Gen. Inference proceeds on e_2, on which the output is based.

Type Instantiation

Type instantiation is of form $\Psi \vdash S \geq [A]T$, where context Ψ and polytype S are inputs, and the monomorphic instance T and list in scope A are outputs. Essentially, type instantiation takes a type of form $\overline{\forall a}^i.T$, removes all quantifiers, and generates a fresh existential type variable $\widehat{\alpha}^i$ for each Skolem type variable a^i, and returns $[\overline{\widehat{\alpha}}^i]([\overline{\widehat{\alpha}}^i/\overline{a}^i]T)$. For example, the `fst` projection of pairs instantiates to $\bullet \vdash \forall a_1.\forall a_2.(a_1, a_2) \rightarrow a_1 \geq [\widehat{\alpha}_1, \widehat{\alpha}_2](\widehat{\alpha}_1, \widehat{\alpha}_2) \rightarrow \widehat{\alpha}_1$.

Well-formedness

Type well-formedness for the algorithmic system is a moderate extension of the declarative one, adding a single rule that checks if existential type variables $\widehat{\alpha}$ are in the context Ψ. Observe that, since objects are invisible to set membership $\in$, $\{[\widehat{\alpha}]\texttt{Unit}\} \not\vdash_{\texttt{ty}} \widehat{\alpha}$.

Contexts are well-formed iff all contained existential type variables are unique and all contained types are well-formed w.r.t. the context to their left, with any A enclosed in an invisible object temporarily added to the context. The notation $A\#\Psi$ ensures not only that A is fresh w.r.t. Ψ, but also that all $\widehat{\alpha}$ in A are fresh w.r.t. each other. Since objects *are* visible to freshness $\#$, context $\{[\widehat{\alpha}]\texttt{Unit}\}; \widehat{\alpha}$ is ill-formed. Another interesting detail is that, while contexts Ψ may contain Skolem type variables a (and this is used to verify the well-formedness of types), well-formed contexts may not contain any Skolem type variables.

4.3 Unification

Figure 6 displays our unification algorithm. The judgment $\Psi_{in} \vdash E \dashv \Psi_{out}$ unifies a list of constraints E of form $T_1 \sim T_2$ under input context Ψ_{in} and produces an output context Ψ_{out}. It can be viewed as the transitive closure of the single-step unification judgment $\Psi_1 \vdash E_1 \longrightarrow \Psi_2 \vdash E_2$, restricted to those sequences that end in $\bullet$.

Hole Notation

We use the syntax $\Psi[\widehat{\alpha}]$ to denote the context $\Psi_L; (A_L, \widehat{\alpha}, A_R); \Psi_R$, where $\Psi[]$ is the context $\Psi_L; (A_L, A_R); \Psi_R$. A multi-hole notation splits the context into more parts. For example, $\Psi[\widehat{\alpha}_1][\widehat{\alpha}_2]$ means $\Psi_1; (A_1, \widehat{\alpha}_1, A_2, \widehat{\alpha}_2, A_3); \Psi_2$ or $\Psi_1; (A_1, \widehat{\alpha}_1, A_2); \Psi_2; (A_3, \widehat{\alpha}_2, A_4); \Psi_3$. Note that hole notation does not split invisible objects.

Single-step Unification

The single-step unification algorithm essentially is a subset of Zhao et al.'s [30], taking only the cases that apply. Rules 1 and 2 simply discharge already-solved constraints. Rule 3 splits constraints on function types. Rules 7 and 8 deal with constraints on two existential type variables. Since our contexts are ordered, we avoid existential type variables escaping their scope by always substituting away the rightmost variable. Rules 9 and 10 solve constraints with an existential variable on one side, and `Unit` on the other.

$\boxed{\Psi_{in} \vdash E \dashv \Psi_{out}}$ Unification Algorithm

$$\frac{}{\Psi \vdash \bullet \dashv \Psi}\ \textsc{SolNil} \qquad \frac{\Psi_{in} \vdash T_1 \sim T_2, E \longrightarrow \Psi \vdash E \qquad \Psi \vdash E \dashv \Psi_{out}}{\Psi_{in} \vdash T_1 \sim T_2, E \dashv \Psi_{out}}\ \textsc{SolCons}$$

$\boxed{\Psi_1 \vdash E_1 \longrightarrow \Psi_2 \vdash E_2}$ Unification Algorithm (Single-step)

$$\begin{array}{rcl}
\Psi \vdash \texttt{Unit} \sim \texttt{Unit}, E & \longrightarrow_1 & \Psi \vdash E \\
\Psi \vdash \widehat{\alpha} \sim \widehat{\alpha}, E & \longrightarrow_2 & \Psi \vdash E \\
\Psi \vdash (T_1 \to T_2) \sim (T_3 \to T_4), E & \longrightarrow_3 & \Psi \vdash T_1 \sim T_3, T_2 \sim T_4, E \\
\Psi[\widehat{\alpha}] \vdash \widehat{\alpha} \sim (T_1 \to T_2), E & \longrightarrow_4 & [\widehat{\alpha}_1 \to \widehat{\alpha}_2/\widehat{\alpha}](\Psi[\widehat{\alpha}_1, \widehat{\alpha}_2] \vdash (\widehat{\alpha}_1 \to \widehat{\alpha}_2) \sim (T_1 \to T_2), E) \\
 & & \quad \text{where } \widehat{\alpha} \notin fv(T_1 \to T_2) \text{ and } \widehat{\alpha}_1, \widehat{\alpha}_2 \# \Psi[\widehat{\alpha}] \\
\Psi[\widehat{\alpha}] \vdash (T_1 \to T_2) \sim \widehat{\alpha}, E & \longrightarrow_5 & [\widehat{\alpha}_1 \to \widehat{\alpha}_2/\widehat{\alpha}](\Psi[\widehat{\alpha}_1, \widehat{\alpha}_2] \vdash (T_1 \to T_2) \sim (\widehat{\alpha}_1 \to \widehat{\alpha}_2), E) \\
 & & \quad \text{where } \widehat{\alpha} \notin fv(T_1 \to T_2) \text{ and } \widehat{\alpha}_1, \widehat{\alpha}_2 \# \Psi[\widehat{\alpha}] \\
\Psi[\widehat{\alpha}_1][\widehat{\alpha}_2] \vdash \widehat{\alpha}_1 \sim \widehat{\alpha}_2, E & \longrightarrow_7 & [\widehat{\alpha}_1/\widehat{\alpha}_2](\Psi[\widehat{\alpha}_1][] \vdash E) \\
\Psi[\widehat{\alpha}_1][\widehat{\alpha}_2] \vdash \widehat{\alpha}_2 \sim \widehat{\alpha}_1, E & \longrightarrow_8 & [\widehat{\alpha}_1/\widehat{\alpha}_2](\Psi[\widehat{\alpha}_1][] \vdash E) \\
\Psi[\widehat{\alpha}] \vdash \widehat{\alpha} \sim \texttt{Unit}, E & \longrightarrow_9 & [\texttt{Unit}/\widehat{\alpha}](\Psi \vdash E) \\
\Psi[\widehat{\alpha}] \vdash \texttt{Unit} \sim \widehat{\alpha}, E & \longrightarrow_{10} & [\texttt{Unit}/\widehat{\alpha}](\Psi \vdash E)
\end{array}$$

Figure 6 Unification Algorithm.

Finally, rules 4 and 5 solve constraints with an existential variable $\widehat{\alpha}$ on one side, and a function type $T_1 \to T_2$ on the other. Because our contexts are ordered, and both T_1 and T_2 may contain existential variables to the left of $\widehat{\alpha}$, we do not directly unify $\widehat{\alpha} := T_1 \to T_2$, but instead split $\widehat{\alpha}$ into a function type $\widehat{\alpha}_1 \to \widehat{\alpha}_2$, where $\widehat{\alpha}_1$ and $\widehat{\alpha}_2$ are fresh w.r.t. the context. This way, rules 7 and 8 may correctly determine which existential variable to eliminate. Because of our notion of *full* contexts, after substitution we can discharge the fact that $\widehat{\alpha} := \widehat{\alpha}_1 \to \widehat{\alpha}_2$, since no other occurrences of $\widehat{\alpha}$ exist. Finally, to ensure termination, we require $\widehat{\alpha}$ does not occur in $T_1 \to T_2$.

5 Metatheory

To reason about how declarative and algorithmic derivations relate, we first need a way of converting between them. We do so through context instantiation, which takes an algorithmic context and converts it to a declarative one. However, this instantiation leaves us with a problem: what to do about invisible objects? To make reasoning about the declarative system easier, we extend declarative contexts Γ with a rule for objects $\Gamma; \{[\overline{a}]\sigma\}$, and assert we can rewrite these away.

▶ **Definition 1.** $\Gamma_1 \equiv_{a,x} \Gamma_2 \triangleq (\forall a, a \in \Gamma_1 \iff a \in \Gamma_2) \ \wedge \ (\forall (x : \sigma), (x : \sigma) \in \Gamma_1 \iff (x : \sigma) \in \Gamma_2)$

▶ **Lemma 2.** *If* $\Gamma_1 \Vdash_{\texttt{mono}} e : \tau$ *and* $\Gamma_1 \equiv_{a,x} \Gamma_2$*, then* $\Gamma_2 \Vdash_{\texttt{mono}} e : \tau$*.*

5.1 Context instantiation

Figure 7 shows simplified context instantiation rules, which implicitly coerce Ψs to Γs and allow for the appending of a and A. They are meant to convey the intuition; their actual full definition can be found in the supplementary materials.

$\boxed{\Psi \rightsquigarrow \Gamma}$ Context instantiation

$$\frac{}{\Gamma \rightsquigarrow \Gamma} \qquad \frac{\Gamma; \overline{a} \vdash_{\mathtt{ty}} \tau \quad \Gamma; \overline{a}; [\tau/\widehat{\alpha}]A; \Psi \rightsquigarrow \Gamma'}{\Gamma; (\widehat{\alpha}; A); \Psi \rightsquigarrow \Gamma'} \qquad \frac{\Gamma; \overline{a}_1; \overline{a}_2 \vdash_{\mathtt{ty}} \tau \quad \Gamma; \{[\overline{a}_1; \overline{a}_2; A][\tau/\widehat{\alpha}]T\}; \Psi \rightsquigarrow \Gamma'}{\Gamma; \{[\overline{a}_1; \widehat{\alpha}; A]T\}; \Psi \rightsquigarrow \Gamma'}$$

Figure 7 Context instantiation.

For existential type variables outside invisible objects, we choose a sequence of Skolem type variables $\overline{a}$ and a declarative type τ that is well-typed w.r.t. the *already-instantiated* context Γ to its left as well as the chosen sequence $\overline{a}$. We proceed by replacing $\widehat{\alpha}$ by $\overline{a}$, and substituting τ for $\widehat{\alpha}$ in the remaining, still-to-be instantiated Ψ to its right. For $\widehat{\alpha}$'s in invisible objects the logic is similar, but the generated sequences A and substitutions stay local to the object itself.

5.2 Soundness

Using context instantiation, we can formulate the soundness of the algorithmic system. We want to show that, for every closed algorithmic derivation, *any* instantiation leads to a valid derivation in the declarative system.

▶ **Theorem 3** (Soundness of the algorithmic system). *If* $\bullet \vdash e : [A]T \dashv \bullet$ *then for all* $A; \{T\} \rightsquigarrow \{\tau\}$ *we have that* $\bullet \Vdash_{\mathtt{mono}} e : \tau$.

This formulation is too weak to prove directly. Instead, we prove a more general variant, from which soundness follows.

▶ **Lemma 4.** *Given* $\mathtt{wf}(\Psi_{in})$*:*
1. *If* $\Psi_{in} \vdash e : [A]T \dashv \Psi_{out}$ *then for all* $\Psi_{out}; A; \{T\} \rightsquigarrow \Gamma; \{\tau\}$ *we have that* $\Gamma \Vdash_{\mathtt{mono}} e : \tau$.
2. *If* $\Psi_{in} \vdash e : S \dashv \Psi_{out}$ *then for all* $\Psi_{out}; \{S\} \rightsquigarrow \Gamma; \{\sigma\}$ *we have that* $\Gamma \Vdash_{\mathtt{poly}} e : \sigma$.

The proof proceeds by mutual induction on the monomorphic and polymorphic algorithmic typing judgments. As the given instantiation instantiates the *output* context, we reason backwards through the algorithm. As a consequence, for rules App and Gen that have multiple recursive hypotheses, to invoke the induction hypotheses the second time we must produce an instantiation of the intermediate context from the instantiation of the output context. To allow for this, we have proven several lemmas about the backwards preservation of instantiation.

▶ **Lemma 5.** *Both typing judgments and unification preserve instantiation. That is:*
1. *If* $\Psi_{in} \vdash e : [A]T \dashv \Psi_{out}$ *and* $\Psi_{out} \rightsquigarrow \Gamma$*, then* $\Psi_{in} \rightsquigarrow \Gamma$.
2. *If* $\Psi_{in} \vdash e : S \dashv \Psi_{out}$ *and* $\Psi_{out} \rightsquigarrow \Gamma$*, then* $\Psi_{in} \rightsquigarrow \Gamma$.
3. *If* $\Psi_{in} \vdash E \dashv \Psi_{out}$ *and* $\Psi_{out} \rightsquigarrow \Gamma$*, then* $\Psi_{in} \rightsquigarrow \Gamma$.

5.3 Completeness

Completeness states that, for any declarative derivation, there exists an algorithmic derivation that instantiates to it.

▶ **Theorem 6** (Completeness of the algorithmic system). *For each declarative derivation there exists an algorithmic derivation that instantiates to it. That is,*
1. *If* $\bullet \Vdash_{\mathtt{mono}} e : \tau$ *then there exists* A T *such that* $A; \{T\} \rightsquigarrow \{\tau\}$ *and* $\bullet \vdash e : [A]T \dashv \bullet$.

2. *If* $\bullet \Vdash_{\texttt{poly}} e : \sigma$ *then there exists* σ' *such that* $\bullet \vdash e : \sigma' \dashv \bullet$ *and* $\Vdash \sigma' \geq \sigma$.

Observe that (2) from Theorem 6 asserts that a polytype σ' *not containing any existential type variables* is inferred. In other words, σ' is fully ground. Again, we proceed by proving a more general lemma.

▶ **Lemma 7.** *Given* $\texttt{wf}(\Psi_{in})$*:*

1. *If* $\Gamma \Vdash_{\texttt{mono}} e : \tau$, $\Gamma' \leqq_{a,x} \Gamma$, *and* $\Psi_{in}; A_{in} \rightsquigarrow \Gamma'$, *then there exists* T A Ψ_{out} Γ'' $\overline{a}$ *s.t.* $\Gamma' = \Gamma''; \overline{a}$, $\Psi_{in} \vdash e : [A]T \dashv \Psi_{out}$ *and* $\Psi_{out}; A; \{T\}; A_{in} \rightsquigarrow \Gamma''; \{\tau\}; \overline{a}$.
2. *If* $\Gamma \Vdash_{\texttt{poly}} e : \sigma$, $\Gamma' \leqq_{a,x} \Gamma$, *and* $\Psi_{in}; A_{in} \rightsquigarrow \Gamma'$, *then there exists* S σ' Ψ_{out} Γ'' $\overline{a}$ *s.t.* $\Gamma' = \Gamma''; \overline{a}$, $\Psi_{in} \vdash e : S \dashv \Psi_{out}$, $\Psi_{out}; \{S\}; A_{in} \rightsquigarrow \Gamma''; \{\sigma'\}; \overline{a}$, *and* $\Gamma' \Vdash \sigma' \geq \sigma$.

Here, $\Gamma_1 \leqq_{a,x} \Gamma_2$ iff two conditions hold. First, the contexts must contain the same type variables in the same order. Second, their term bindings $(x : S)$ must (1) bind the same names in the same order to (2) types that are related by subsumption under Γ_1.

Finally, we admit the following property about unification:

▶ **Axiom 8.** *If a unifier exists, unification succeeds. That is, if* $\theta T_1 = \theta T_2$ *and* $\Psi_{in} \rightsquigarrow \Gamma$ *then there exists* Ψ_{out} *such that* $\Psi_{in} \vdash T_1 \sim T_2 \dashv \Psi_{out}$ *and* $\Psi_{out} \rightsquigarrow \Gamma$.

5.4 Decidability

In our algorithm there is only one part of which decidability is not obvious: unification. Hence, we prove its decidability here.

▶ **Theorem 9** (Decidability of unification). *Given* $\forall T_1 T_2.\ T_1 \sim T_2 \in E \implies (\Psi_{in} \vdash_{\texttt{ty}} T_1 \wedge \Psi_{in} \vdash_{\texttt{ty}} T_2)$*, it is decidable whether there exists a* Ψ_{out} *such that* $\Psi_{in} \vdash E \dashv \Psi_{out}$.

The proof proceeds by induction on the lexicographic measure $\langle |\Psi_{in}|_{\widehat{\alpha}}, |E| + 2 * |E|_{\rightarrow} \rangle$, representing the number of existential type variables in Ψ_{in} and the length and number of function arrows in E, respectively. All rules directly reduce this measure, except for rules 4 and 5. For these, we need an additional lemma, from which these cases follow. Let us categorize lists of constraints where one side is an existential type variable that does not occur in the rest of the list as E_i, and assert that we can solve any head of pattern E_i without increasing the length of the tail.

$$
\begin{array}{rcll}
E_i & ::= & \bullet & \\
 & | & \widehat{\alpha} \sim T, E_i & \text{with } \widehat{\alpha} \notin E_i \\
 & | & T \sim \widehat{\alpha}, E_i & \text{with } \widehat{\alpha} \notin E_i
\end{array}
$$

▶ **Lemma 10** (Solving E_i). *For all* Ψ_{in} E_i E *there exist* Ψ_{out} E' *such that* $\Psi_{in} \vdash E_i + E \longrightarrow^* \Psi_{out} \vdash E'$ *and* $|\Psi_{out}|_{\widehat{\alpha}} = |\Psi_{in}|_{\widehat{\alpha}} - |E_i|$.

Proof. By induction on $\langle |E_i| + 2 * |E_i|_{\rightarrow} \rangle$. Rules 1, 9 and 10 do not apply. The rest directly reduce the measure, except for (again) rules 4 and 5. We consider rule 4, where $E_i = \widehat{\alpha} \sim (T_1 \rightarrow T_2), E_i'$. It must be immediately followed by rule 3, which gives us $\Psi_{in}[\widehat{\alpha}] \vdash \widehat{\alpha} \sim (T_1 \rightarrow T_2), E_i, E \longrightarrow^* [\widehat{\alpha}_1 \rightarrow \widehat{\alpha}_2]\Psi_{in}[\widehat{\alpha}_1, \widehat{\alpha}_2] \vdash \widehat{\alpha}_1 \sim T_1, \widehat{\alpha}_2 \sim T_2, E_i, E$. Because we know $\widehat{\alpha} \notin E_i'$, we know any substitution of $\widehat{\alpha}$ on E_i does not increase $|E_i|_{\rightarrow}$. Even though we have added an existential variable, we end up with a decreased measure because we have eliminated an arrow, which counts for two. ◀

6 Mechanization

We have mechanised both the declarative specification presented in Section 3 as well as the algorithmic system presented in Section 4 in the Coq proof assistant [23]. Furthermore, we have proven the algorithmic system sound and correct w.r.t. the declarative speciation following the approach described in Section 5. The mechanization is implemented by generating definitions with Ott [20] and its backend [29] for the locally nameless representation [2, 5, 15]. To reason about the locally nameless representation, we have generated many useful lemmas with LNgen [1]. The mechanisation consists of ±700 handwritten lines of Ott DSL, ±10 000 lines of handwritten Coq code, ±900 lines of Coq code generated by Ott, and ±6 800 lines of Coq code generated by LNgen.

We start this section with a discussion of these tools and the locally nameless representation. Then, we discuss the major points of difference between what is presented in the paper and the formalization. The mechanisation as well as an exhaustive list of the delta between the paper and the mechanization are available in the supplementary material, as well as at `https://github.com/rogerbosman/hdm-fully-grounding`.

6.1 Ott

Our mechanization uses the Sewel et al.'s Ott [20] DSL to express both the syntax and inference rules in this paper and generate corresponding (LaTeX and) Coq definitions, as well as boilerplate definitions such as substitutions and free variable functions. As Sewell et al. already argue the general benefits of Ott, here we focus only on the aspects that we found particularly useful.

Typically, manually written LaTeX specifications make notational liberties that do not translate well to Coq. For example, we have taken such a liberty in the environment instantiation judgment as discussed in Section 5.1. Ott rejects such ill-typed definitions. Hence, it forces well-typed formulations that can be translated to Coq, but are more verbose in LaTeX. As a compromise, we have stuck to the Ott-generated LaTeX during the development and have manually produced a cleaned-up version for this paper.[3]

A clear advantage of the Ott-generated outputs is that they both have the same single source of truth. Thus, the LaTeX output can be used to reason about the Coq output. Another substantial advantage is that Ott takes care of generating boilerplate definitions such as free variable functions and substitutions.

6.2 The locally nameless representation

Formalizations that contain abstraction must represent variables in some way. Typically, variables are either referred to by explicit name – which suffers from the lack of built-in α-equivalence, and have issues such as shadowing – or a nameless representation such as De Bruijn indices [8], which are sensitive to the context in which they are defined, requiring *shifting* operations whenever such changes occur.

The locally nameless representation combines the two approaches: it uses a named representation for free variables, and a nameless representation for locally bound variables. As a consequence, each alpha-equivalence class of closed lambda terms has a unique representation. At the same time, terms are less sensitive to changes in their context. For example, the lambda expression $\lambda x.\, x\ y$ is represented as $\lambda.\, 0y$, because x is locally bound, while y is

[3] We describe the difference in Section 6.4.

free. This implies a well-formedness condition, namely that every nameless variable has a corresponding abstraction, in other words, that nameless variables are not free. This condition is called *locally closed*.

A locally bound variable can be converted to a named, free variable through *opening*, where any reference to the outermost abstraction is replaced by a named variable. We use e^x to denote opening term e with name x. It's dual is *closing*. We use ${}^{\backslash x}e$ to denote closing e w.r.t. x. Our mechanization uses the locally nameless representation for both the declarative and algorithmic term variables x and for the Skolem type variables a. Since existential type variables $\widehat{\alpha}$ do not have a matching abstraction, they are always free, and thus use the named representation.

Cofinite Quantification

To preserve the locally closed property, whenever we go under a binder, we have to open the term with some named variable quantified over in some way. There are several ways to go about this. One way would be to use existential quantification, where we assert that there exists some name not in the free variables of the term being opened. Consider rule ∀WF-EX below, which applies this principle to the well-formedness of declarative type schemes.

$$\frac{\exists a.(a \notin fv(\sigma) \cup fv(\Gamma) \qquad \Gamma; a \vdash_{\mathtt{ty}} \sigma^a)}{\Gamma \vdash_{\mathtt{ty}} \forall.\sigma}\ \forall\text{WF-EX} \qquad \frac{\exists L.\forall a.a \notin L.\ \Gamma; a \vdash_{\mathtt{ty}} \sigma^a}{\Gamma \vdash_{\mathtt{ty}} \forall.\sigma}\ \forall\text{WF-COF}$$

As described by Aydemir et al. [2], existential quantification is weak as an elimination form. For example, since eliminating this rule only gives well-formedness for one particular name, renaming lemmas are required for deriving well-formedness over any other name.

Universal quantification suffers from the opposite problem: it can be cumbersome to prove the well-formedness of *any* variable satisfying the freshness constaints. In particular, sometimes we want to exclude more variables than just those in $fv(\sigma) \cup fv(\Gamma)$.

Cofinite quantification, as displayed by rule ∀WF-COF above, offers exactly this. Here, we quantify universally over any name not in some existentially quantified set L. This elimination form is much stronger than with existential quantification, because we know well-formedness to hold for any $a \notin L$, instead of just one, avoiding, in general, the need for renaming lemmas. Yet, as an introduction form, it is much easier to use than with universal quantification, because it allows us to exclude finitely many names, instead of just the fixed set of free variables. While cofinite quantification is not free of quirks (particularly the control flow of quantification), which we describe below, in general, it strikes the best balance.

Ott's Locally Nameless Backend & LNgen

One drawback of cofinite quantification is that implementation details of the variable representation leak to the LaTeX inference rules. Here, Ott's locally nameless backend [29] comes in handy: it automatically converts inference rules as specified in Sections 3 and 4 to those that use a (cofinitely quantified) locally nameless presentation *for Coq only*. The LaTeX definitions render as the original specification.

By default, Ott's locally nameless backend generates definitions for opening terms, but not for closing them. Weirich's Ott fork [27] adds the generation of these closing definitions.

The opening and closing operations are subject to various laws. One of these, which will become relevant later, is the following.

▶ **Proposition 11** (Subsitution as Open and Close). *Subsitution can be defined in terms of* open *and* close. *That is,* $[T/a]S = ({}^{\backslash a}S)^T$.

Proposition 11 as well as many others are automatically generated and proven by LNgen [1], which bases itself on the Ott specification. Our mechanization uses these laws extensively.

6.3 Quirks of the locally nameless representation

As with any variable representation, some quirks arise. We cover three here.

Generalisation

First is the gen function used in Gen in Figure 5, and its definition[4] is displayed below.

$$\begin{aligned}
&\mathsf{gen}(S, \quad \bullet \quad, _) = S \\
&\mathsf{gen}(S, (A; \widehat{\alpha}), L) = \textbf{let } S' = \mathsf{gen}(S, A, L),\ a \# fv(S') \cup L \\
&\qquad\qquad\qquad\qquad\quad \textbf{in } \forall.^{\backslash a}([a/\widehat{\alpha}]\ S')
\end{aligned}$$

Since variable closing closes nameless Skolem type variables a only, we first substitute in a freshly generated one, only to close it away immediately after. While it would be possible to manually define a closing operation that replaces (named) existential type variables with unnamed Skolem type variables, we would lose the ability to reason over them with the laws generated by LNgen. While we cannot completely avoid having to manually replicate some of these in some instances, here we can avoid doing so. Fortunately, because of these same LN-generated laws, reasoning about this is straightforward. If we open the generalised term with some T, we get $({}^{\backslash a}([a/\widehat{\alpha}]\ S'))^{T}$. By Proposition 11, this can be rewritten into $[T/a][a/\widehat{\alpha}]\ S'$, which simplifies to $[T/\widehat{\alpha}]\ S'$.

Lists of variables

Rule TmGen in Figure 2 quantifies over a list of variables $\overline{a}$. Quantifying cofinitely over and opening with a list of type variables instead of a singular variable requires additional machinery and is not supported by Ott. Attempts at patching the generated definitions manually were unsuccessful (we discuss this again in Section 8). As a consequence, the list of variables $\overline{a}$ is quantified existentially, which is why we used an axiom in our proof of the weakening lemma for declarative typing judgments.

Control Flow

When inducting over typing derivations, we have existentially quantified sets of variables L, and universally quantified variables fresh w.r.t. L. Sets L flow downwards from the induction hypothesis to the conclusion. Yet, variables flow upwards from the conclusion to the induction hypothesis. Consider the abstraction case for completeness, which essentially consists of proving the following implication.

$$\begin{aligned}
&(\exists L. \forall x. x \notin L \implies \exists\ \Psi_{out}\ A\ T_2.\ \Psi_{in}; [\widehat{\alpha}]; x : \widehat{\alpha} \vdash e^x : [A_2]T_2 \dashv \Psi_{out}; A_1; x : T_1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \Psi_{out}; A_1; x : T_1; A_2; \{T_2\} \rightsquigarrow \Gamma; x : \tau_1; \{\tau_2\}) \\
&\implies \Psi_{in} \vdash \lambda.e : [A_1; A_2]T_1 \to T_2 \dashv \Psi_{out} \wedge \Psi_{out}; A_1; A_2; \{T_1 \to T_2\} \rightsquigarrow \Gamma; \{\tau_1 \to \tau_2\}
\end{aligned}$$

[4] Observe that gen is parametrised with a third argument, unspecified in Gen, which is included in the set w.r.t. fresh variables are generated, i.e. $a \# fv(S') \cup L$. Since fresh variables are immediately closed away, the generalised term is not affected by a choice for L. It is helpful proving the commutativity of generalisation with for example substitution of existential type variables.

There is a problem here. Since we only obtain the term variable to open e with after applying the Abs constructor in the right branch of the conclusion, we do not have access to it in the left branch of the conclusion. Since the IH existentially quantifies objects that occur in both branches of the conclusion, we cannot simply apply the IH twice, once per branch. While the IH can probably be strengthened to shift the $\forall x$ to each of its two branches, we found it easier to apply the IH to a sufficiently fresh variable before splitting the conclusion. This leaves us with a typing derivation opened with a different term than required. However, this can be remedied straightforwardly with the following renaming lemma.

▶ **Lemma 12.** $\Psi_{in} \vdash e : [A]T \dashv \Psi_{out} \implies [y/x]\Psi_{in} \vdash [y/x]e : [A]T \dashv [y/x]\Psi_{out}$

6.4 Delta between the paper and the mechanization

We cover the two most important differences between the system as presented in this paper and the mechanization.

Unification

To facilitate easier reasoning over unification, the mechanisation's single-step unification judgment rules do not apply the substitution directly, but instead output the substitution as a third output, giving unification the form $\Psi_1 \vdash E_1 \longrightarrow \Psi_2 \vdash E_2, \gamma$, where γ has form $[\overline{T}/\overline{\widehat{\alpha}}]$. Note that single steps return either the empty list, or a singleton list. The auxiliary judgment $\Psi_{in} \vdash E \dashv \Psi_{out}, \gamma$ takes the substitution generated by the single-step judgment, applies it to the step's result (yielding the same result as the paper's single-step judgment), and then combines it with the inductive result. Finally, $\Psi_{in} \vdash E \dashv \Psi_{out}$ is defined in terms of this auxiliary judgment by simply discarding the substitution.

Context Instantiation

The instantiation as presented in Section 5.1 contains notation that is not properly translatable to an inductive type. We present instantiation in this manner to obtain a simpler overview of the logic of context instantiation. The instantiation in the mechanization can be obtained by applying the following three transformations.

First, instead of concatenating the already-processed Γ with the yet-to-be processed Ψ, we define instantiation inductively on Ψ, where we pattern match on the different heads of Ψ, process the tail, and then add the processed head. This means that when generating a substitution for an existential type variable $\widehat{\alpha}$ we do not have access to the yet-to-be processed Ψ, since now $\widehat{\alpha}$ is at the head. Therefore, we flip the control flow, instead deriving a substitution θ of form $[\overline{\tau}/\overline{\widehat{\alpha}}]$, and apply it to any bound type later. This yields a signature of $\Gamma \rightsquigarrow \Psi, \theta$.

Then we split out the instantiation of A's in a dedicated judgment, $A \rightsquigarrow \overline{a}, \theta$. Finally, to make it easier to reason about instantiation, we add a context Γ_{in}, θ_{in} such that the following holds.

▶ **Theorem 13** (Splitting and merging context instantiation). *Context instantiation judgments can be split and merged. That is:*

- $\Gamma_{in}, \theta_{in} \vdash \Psi_1; \Psi_2 \rightsquigarrow \Gamma, \theta \implies \exists\, \Gamma_1\ \Gamma_2\ \theta_1\ \theta_2,\ \Gamma = \Gamma_1; \Gamma_2 \wedge \theta = \theta_2; \theta_1$
  $\wedge\ \Gamma_{in}, \theta_{in} \vdash \Psi_1 \rightsquigarrow \Gamma_1, \theta_1 \wedge \Gamma_{in}; \Gamma_1, \theta_1; \theta_{in} \vdash \Psi_2 \rightsquigarrow \Gamma_2, \theta_2.$
- $\Gamma_{in}, \theta_{in} \vdash \Psi_1 \rightsquigarrow \Gamma_1, \theta_1 \wedge \Gamma_{in}; \Gamma_1, \theta_1; \theta_{in} \vdash \Psi_2 \rightsquigarrow \Gamma_2, \theta_2 \implies$
  $\Gamma_{in}, \theta_{in} \vdash \Psi_1; \Psi_2 \rightsquigarrow \Gamma_1; \Gamma_2, \theta_2; \theta_2.$

7 Related Work

The algorithm presented in this paper extends a long line of work on the inference of the HDM system [9, 16, 12, 14]. Yet, a surprisingly small amount of work addresses the issue of underconstrained type variables.

Pottier [18] gives an (not formalised) elaboration algorithm which inspects the accumulated constraints to determine the list of variables in scope of types. In the appendix, they identify the problem of potentially unnecessary quantification. They address this with a non-deterministic specification that "magically" chooses which variables to abstract over.

Vytiniotis et al. advocate [25] removing the generalisation of lets altogether, citing unwanted interactions and needless complexity in context of generalising types with constraints arising from, for example, type classes or GADTs [28]. They observe that removing let generalisation would not be a significant restriction, since most programs do not utilize this functionality. Yet, removing let generalisation would not address the problem of underconstrained types: they would still need to be dealt with, only now by defaulting, since generalisation is no longer an option.

Zhao et al. mechanised [30] an algorithm for Dunfield and Krishnaswami's [10] type system featuring higher-rank polymorphism. However, since these systems are bidirectional, it is left to the programmer to decide which type variable should be generalised over where, if at all. Yet, we have taken a great deal of inspiration from both these works, adopting the in- and output contexts from Dunfield and Krishnaswami, and manner of tracking existential type variables and approach to unification from Zhao et al.

Zhao et al. rewrote Dunfield and Krishnaswami's algorithmic system, citing the lack of support by their proof assistant of choice (Abella [11]) as one of their reasons. Since we are not using any built-in variable binding support (like what is supported by Abella), we did not encounter such limitations. Thus, we were able to maintain the tree-like structure of Dunfield and Krishnaswami instead of the flatter, list-based approach of Zhao et al.

8 Conclusion

In this paper we have presented algorithm $\mathcal{R}$: the first mechanically verified, fully grounding type inference algorithm for the HDM system. The contribution features the novel approach to unification by using full contexts, in which the current context always represents the *entire* context. The algorithm lays the foundation for formalizing algorithms that require determining types for every subterm.

While any variable representation will have its quirks, the quirks of locally nameless as discussed in Section 6.3 make us wonder if a fully nameless representation would be easier to work with. Our design choice of a separate judgment for generalisation did not turn out well. This approach requires mutual induction on the monomorphic and polymorphic typing judgments, which is a nuisance. Furthermore, Coq not being able to generate this mutual induction scheme is what left us unable to manually patch the inference rule for generalisation to quantify the list of variables $\overline{a}$ cofinitely, as discussed in Section 6.3.

One particularly interesting future area of work is the extension of the algorithm with elaboration to an explicitly typed language like System F, potentially extended with elaboration-based features such as Go's structural subtyping system [21] or type classes [26], whose coherence has been proven on paper in a bidirectional setting [3], but – as far as we know – not yet in the HDM system. Since formalizing these algorithms requires reasoning about the scope of existential variables, our work should serve as a solid starting point.

References

1 Brian Aydemir and Stephanie Weirich. LNgen: Tool support for locally nameless representations. Technical report, Department of Computer and Information Science, University of Pennsylvania, 2010.

2 Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 3–15. ACM, 2008. doi:10.1145/1328438.1328443.

3 Gert-Jan Bottu, Ningning Xie, Koar Marntirosian, and Tom Schrijvers. Coherence of type class resolution. *Proc. ACM Program. Lang.*, 3(ICFP):91:1–91:28, 2019. doi:10.1145/3341695.

4 Hendrik Bünder. Decoupling language and editor - the impact of the language server protocol on textual domain-specific languages. In Slimane Hammoudi, Luís Ferreira Pires, and Bran Selic, editors, *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2019, Prague, Czech Republic, February 20-22, 2019*, pages 129–140. SciTePress, 2019. doi:10.5220/0007556301310142.

5 Arthur Charguéraud. The locally nameless representation. *J. Autom. Reason.*, 49(3):363–408, 2012. doi:10.1007/s10817-011-9225-2.

6 Dominique Clément, Joëlle Despeyroux, Th. Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ml. In William L. Scherlis, John H. Williams, and Richard P. Gabriel, editors, *Proceedings of the 1986 ACM Conference on LISP and Functional Programming, LFP 1986, Cambridge, Massachusetts, USA, August 4-6, 1986*, pages 13–27. ACM, 1986. doi:10.1145/319838.319847.

7 Luís Damas and Robin Milner. Principal type-schemes for functional programs. In Richard A. DeMillo, editor, *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*, pages 207–212. ACM Press, 1982. doi:10.1145/582153.582176.

8 N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. doi:10.1016/1385-7258(72)90034-0.

9 Catherine Dubois and Valérie Ménissier-Morain. Certification of a type inference tool for ML: damas-milner within coq. *J. Autom. Reason.*, 23(3-4):319–346, 1999. doi:10.1023/A:1006285817788.

10 Jana Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 429–442. ACM, 2013. doi:10.1145/2500365.2500582.

11 Andrew Gacek. The abella interactive theorem prover (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 154–161. Springer, 2008. doi:10.1007/978-3-540-71070-7_13.

12 Jacques Garrigue. A certified implementation of ML with structural polymorphism and recursive types. *Math. Struct. Comput. Sci.*, 25(4):867–891, 2015. doi:10.1017/S0960129513000066.

13 Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Éditeur inconnu, 1972.

14 Adam Gundry, Conor McBride, and James McKinna. Type inference in context. In Venanzio Capretta and James Chapman, editors, *Proceedings of the 3rd ACM SIGPLAN Workshop on Mathematically Structured Functional Programming, MSFP@ICFP 2010, Baltimore, MD, USA, September 25, 2010*, pages 43–54. ACM, 2010. doi:10.1145/1863597.1863608.

15 Conor McBride and James McKinna. Functional pearl: i am not a number-i am a free variable. In Henrik Nilsson, editor, *Proceedings of the ACM SIGPLAN Workshop on Haskell,*

Haskell 2004, Snowbird, UT, USA, September 22-22, 2004, pages 1–9. ACM, 2004. doi:10.1145/1017472.1017477.

16 Wolfgang Naraschewski and Tobias Nipkow. Type inference verified: Algorithm W in isabelle/hol. *J. Autom. Reason.*, 23(3-4):299–318, 1999. doi:10.1023/A:1006277616879.

17 Andrey Popp, Rusty Key, Louis Roché, Oleksiy Golovko, Rudi Grinberg, Sacha Ayoun, cannorin, Ulugbek Abdullaev, Thibaut Mattio, and Max Lantas. ocaml-lsp-server 1.15.1-5.0 – opam, January 2023. URL: https://opam.ocaml.org/packages/ocaml-lsp-server/ocaml-lsp-server.1.15.1-5.0/.

18 François Pottier. Hindley-milner elaboration in applicative style: functional pearl. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 203–212. ACM, 2014. doi:10.1145/2628136.2628145.

19 John C. Reynolds. Towards a theory of type structure. In Bernard J. Robinet, editor, *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974. doi:10.1007/3-540-06859-7_148.

20 Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Tom Ridge, Susmit Sarkar, and Rok Strnisa. Ott: effective tool support for the working semanticist. In Ralf Hinze and Norman Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 1–12. ACM, 2007. doi:10.1145/1291151.1291155.

21 Martin Sulzmann and Stefan Wehr. A dictionary-passing translation of featherweight go. In Hakjoo Oh, editor, *Programming Languages and Systems - 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17-18, 2021, Proceedings*, volume 13008 of *Lecture Notes in Computer Science*, pages 102–120. Springer, 2021. doi:10.1007/978-3-030-89051-3_7.

22 GHC Team. Using GHCi - GHC User's Guide 9.4.4. URL: https://downloads.haskell.org/ghc/9.4.4/docs/users_guide/index.html.

23 The Coq Development Team. The coq proof assistant, September 2022. doi:10.5281/zenodo.7313584.

24 The Haskell IDE Team. haskell-language-server documentation. URL: https://haskell-language-server.readthedocs.io/en/latest/.

25 Dimitrios Vytiniotis, Simon Peyton Jones, and Tom Schrijvers. Let should not be generalized. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '10, pages 39–50, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1708016.1708023.

26 Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 60–76. ACM Press, 1989. doi:10.1145/75277.75283.

27 Stephanie Weirich. Github repository: sweirich/ott, April 2022. URL: https://github.com/sweirich/ott/tree/aa65f53ea0587223662aaad9c48cb0770549f018.

28 Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In Alex Aiken and Greg Morrisett, editors, *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, pages 224–235. ACM, 2003. doi:10.1145/604131.604150.

29 Francesco Zappa Nardelli. A locally-nameless backend for ott, March 2009. URL: https://fzn.fr/projects/ln_ott/.

30 Jinxu Zhao, Bruno C. d. S. Oliveira, and Tom Schrijvers. A mechanical formalization of higher-ranked polymorphic type inference. *Proc. ACM Program. Lang.*, 3(ICFP):112:1–112:29, 2019. doi:10.1145/3341716.

Automated Theorem Proving for Metamath

Mario Carneiro
Carnegie Mellon University, Pittsburgh, PA, USA

Chad E. Brown
Czech Technical University in Prague, Czech Republic

Josef Urban
Czech Technical University in Prague, Czech Republic

Abstract

Metamath is a proof assistant that keeps surprising outsiders by its combination of a very minimalist design with a large library of advanced results, ranking high on the Freek Wiedijk's 100 list. In this work, we develop several translations of the Metamath logic and its large set-theoretical library into higher-order and first-order TPTP formats for automated theorem provers (ATPs). We show that state-of-the-art ATPs can prove 68% of the Metamath problems automatically when using the premises that were used in the human-written Metamath proofs. Finally, we add proof reconstruction and premise selection methods and combine the components into the first hammer system for Metamath.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Theory of computation → Higher order logic; Theory of computation → Logic and verification

Keywords and phrases Metamath, Automated theorem proving, Interactive theorem proving, Formal proof assistants, proof discovery

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.9

Supplementary Material *Software*: `https://github.com/ai4reason/mm-atp-benchmark`
Software: `https://github.com/digama0/mm-hammer`

Funding This work was partially supported by the European Regional Development Fund under the Czech project AI&Reasoning no. CZ.02.1.01/0.0/0.0/15_003/0000466 (CB, JU), Amazon Research Awards (CB, JU), the Czech MEYS under the ERC CZ project *POSTMAN* no. LL1902 (CB), and the EU ICT-48 2020 project TAILOR no. 952215 (JU).
Mario Carneiro: Supported by the Hoskinson Center for Formal Mathematics at CMU.

1 Introduction

Metamath [20] is a formal system developed by Norman Megill in 1990. Its largest database, `set.mm`[1], has 40338 theorems in ZFC set theory, including a diverse range of topics including analysis, topology, graph theory, number theory, Hilbert spaces, and it continues to grow steadily due to its small but active community. In the space of theorem prover languages, it is one of the simplest, by design.

Metamath is one of the last formal proof systems with a large mathematical library that have not yet been translated to today's automated theorem provers (ATPs) [23]. Such translations between ITPs and ATPs are one of the main parts of *hammer systems* [4], which have become popular in the recent years, especially in the Isabelle community [21, 22, 3, 18, 8]. Hammer systems today exist also for the Coq [7, 9], HOL [10, 15], and Mizar [27, 16, 14] proof assistants. The goal of this work is to provide the first such ATP translation for Metamath, and to do the first evaluation of the potential of state-of-the-art ATP systems on

[1] `https://github.com/metamath/set.mm`


© Mario Carneiro, Chad E. Brown, and Josef Urban;
licensed under Creative Commons License CC-BY 4.0

the translated Metamath library. This also results in a new large mathematical benchmark for ATP systems. We also build other components of the first full Metamath hammer here, such as proof reconstruction and premise selection [1].

The rest of the paper is structured as follows. Section 2 provides a brief summary of Metamath. In Section 3, we describe a translation of Metamath to HOL via the Metamath Zero (MM0) system. This is then used as an input to several versions of translation to the higher-order TPTP (TH0) format [11] in Section 4. We then describe an alternative translation to first-order class theory in Section 5. Section 6 describes the resulting large new benchmark of higher-order and first-order ATP problems obtained by the translation, and Section 7 provides the first evaluation of higher-order and first-order ATPs on the benchmark. We show that with higher time limits, the ATP systems can jointly prove 68% of the problems when using the premises provided in the human written Metamath proofs. This is a very encouraging result, both for the development of hammers for Metamath, and for the developers of ATP systems. In section 8 we discuss how IVY proofs are reconstructed to Metamath, and section 9 has some interesting examples of theorems the provers were able to get.

2 Metamath

Development in Metamath is generally facilitated by a proof assistant, and unlike many theorem provers there is no proof assistant that has a monopoly on the job; the most common proof assistants in use are mmj2 by Mel O'Cat and MM-PA which is bundled with the original `metamath.exe` verifier by Norman Megill. A `.mm` Metamath file does not contain proof scripts, but rather it is a textual format for complete and fully explicit proof objects. This makes it very attractive as a data source, because very little is required to parse and validate the theorems from the file.

The name "Metamath" comes from "metavariable mathematics," because the core concept is the pervasive use of metavariables over an object logic. For example, theorem ax6e[2] asserts $\vdash \exists x\ x = y$, but x and y are metavariables ranging over variables in FOL. Depending on whether x and y are taken to be the same or different FOL variables, we get two α-equivalence classes of FOL theorems from this single Metamath theorem: $\vdash \exists u\ u = v$ asserts that there exists an element equal to some fixed free variable v, and $\vdash \exists u\ u = u$ asserts that there exists an element equal to itself. These are both true statements, and the original Metamath theorem includes both of these as substitution instances.

This ability for a Metamath theorem to encode multiple α-equivalence classes of FOL theorems is known in the Metamath community as "bundling," and it poses a problem for translation to plain FOL or HOL.

3 Translating Metamath to HOL via MM0

Metamath Zero [5] is a formal system developed by M. Carneiro with a logic somewhat intermediate between Metamath and HOL. It positions itself as an interchange language between other proof systems, and in particular the MM0 toolchain[3] implements a translation from Metamath to MM0 that addresses exactly this bundling issue. MM0 requires that all

[2] `https://us.metamath.org/mpeuni/ax6e.html`
[3] `https://github.com/digama0/mm0`

theorems are fully unbundled, meaning that a theorem like ax6e has to be split into two theorems:

- ax6e: $\vdash \exists x\ x = y$
- ax6e_b: $\vdash \exists x\ x = x$

Each theorem now has a straightforward rendering as a theorem in FOL.

Another problem that the translator addresses is parsing of math expressions. In metamath, the native representation of statements is as sequences of constant and variable tokens, so a verifier does not truly need to know how to break the statement into formula constructors – this is encoded as part of the proof itself. In MM0, the native representation instead uses trees of expression constructors, which is a better fit for traditional ATPs. So ax6e would actually be translated as (wex x (wceq (cv x) (cv y))) which now encodes the parse tree of $\exists x\ x = y$ (including the invisible cv coercion from set variables to class expressions).

The MM0 toolchain also has a translator from MM0 to HOL, in a lisp-based format. The main mismatch at this level is that MM0 variables represent open terms, and so they have to be transformed into higher order variables to match HOL semantics. For example, the Metamath theorem axi4:[4] $\vdash (\forall x\ \varphi \to \varphi)$ is translated to the MM0 theorem:

$$\mathsf{axi4}\ \{x : \mathsf{setvar}\}\ (\varphi : \mathsf{wff}\ x) \colon (\mathsf{wi}\ (\mathsf{wal}\ x\ \varphi)\ \varphi)$$

where the binders encode that x is a first order set variable and φ is a second order wff variable that is allowed to depend on x.[5] It is translated to the following HOL theorem:[6]

$$\mathsf{axi4} : \forall\ (\varphi : \mathsf{setvar} \to \mathsf{wff})\ (x : \mathsf{setvar}).\ \ (\mathsf{wi}\ (\mathsf{wal}\ (\lambda x : \mathsf{setvar}.\ \varphi\ x))\ (\varphi\ x))$$

This is written in lisp concrete syntax as:

```
(theorem "axi4"
  (for ("ph" ("setvar" "wff")))
  (for)
  (for ("x3" "setvar"))
  ("wi"
    ("wal" (fn ("x3" "setvar") ("ph" "x3")))
    ("ph" "x3"))
  ("sp" (fn ("x3" "setvar") ("ph" "x3")) "x3"))
```

The three `(for)` blocks are for introducing binders for the second order variables, then the hypotheses (of which there are none in this example), and finally the first order variables, and then the type of the theorem as above. The last expression, (sp $(\lambda x : \mathsf{setvar}.\ \varphi\ x)\ x$), is the proof, encoded as a lambda calculus term. (The proof in this case is short because axi4 is just an alias of theorem sp, but usually this will include many theorem applications.)

By chaining all these translations on `set.mm` we obtain a `set.lisp` file containing statements and complete proofs for every theorem in `set.mm`.

---

[4] `https://us.metamath.org/mpeuni/axi4.html`

[5] Note that "wff" is a common abbreviation for "well-formed formula."

[6] Names of costructors that return wffs begin with the letter w. For example, wi is the wff constructor for implication and wal is the wff constructor for universal quantification (over sets).

4 Translating Metamath in HOL to TH0

From the higher-order representation of set.mm we have several options for how to create TH0 problems for automated theorem provers. We consider specifically three versions (denoted as *v1*, *v2* and *v3* below) of this last part of the translation. To describe the difference between these three versions, we consider a few example theorems from set.mm.

4.1 Translation v1

Each set.mm theorem consists of a finite list of premises[7] (wffs) and a conclusion (a wff). The premises and conclusion may depend on certain variables ranging over wffs, classes and sets. Wff variables and class variables may depend on sets and these dependencies are made explicit in the higher-order translation of set.mm. Set variables may also be locally bound (while variables representing wffs and classes are global to the theorem). In the source higher-order logic (after translating set.mm as above and before translating to TH0) there are three base types: wff, class and setvar. We also have function types $\alpha \to \beta$ as usual. There are a variety of constructors for wffs and classes. For the examples we only need these few, given here with their source higher-order types:

- wi : wff → wff → wff (implication)
- wa : wff → wff → wff (conjunction)
- wb : wff → wff → wff (equivalence)
- w3a : wff → wff → wff → wff (tertiary conjunction)
- wceq : class → class → wff (equality on classes)
- wcel : class → class → wff (membership on classes)
- wal : (setvar → wff) → wff (universal quantification)
- wsb : (setvar → wff) → (setvar → wff) (substitution)
- cab : (setvar → wff) → class (class abstraction)[8]

In set.mm a set variable can be used as a class. After translating to higher-order, we need a corresponding way to coerce a set variable to be a class. This is given by sv of type setvar → class.

When translating to TH0, we can use the builtin type o (`$o` in ASCII) for wff and the builtin type ι (`$i` in ASCII) for setvar. Also, we can use the type $\iota \to o$ for class. So the translated types (for all versions of the translations) are as follows:

- wi : $o \to o \to o$
- wa : $o \to o \to o$
- wb : $o \to o \to o$
- w3a : $o \to o \to o \to o$
- wceq : $(\iota \to o) \to (\iota \to o) \to o$
- wcel : $(\iota \to o) \to (\iota \to o) \to o$
- wal : $(\iota \to o) \to o$
- wsb : $(\iota \to o) \to \iota \to o$
- cab : $(\iota \to o) \to \iota \to o$
- sv : $\iota \to \iota \to o$

[7] Here the words *premise* and *conclusion* are used in the meaning of *antecedent* and *succedent* of a sequent. In particular, *premise* does not mean here *"another toplevel fact used in the proof"* (premise selection terminology).

[8] This is a constructor that returns a class. Names of such constructors usually begin with c.

There are over 1200 other constructors we will not explicitly mention here. There are no constructors (even among the unmentioned ones) returning ι, so that when translating we can always assume terms of type ι are variables.

Consider the theorem sylan9eq.[9] The theorem depends on two wff variables φ and ψ and three class variables A, B and C. The theorem has two premises: wi φ (wceq A B) and wi ψ (wceq B C). The conclusion of the theorem is wi (wa φ ψ) (wceq A C). In the v1 translation this theorem is translated into the closed proposition

$$\forall\,(\varphi\ \psi : o)\ (A\ B\ C : \iota \to o).\ \mathsf{wi}\ \varphi\ (\mathsf{wceq}\ A\ B) \to \mathsf{wi}\ \psi\ (\mathsf{wceq}\ B\ C) \to \mathsf{wi}\ (\mathsf{wa}\ \varphi\ \psi)\ (\mathsf{wceq}\ A\ C).$$

This closed proposition is, of course, not provable in isolation. When creating the TH0 problem we examine the set.mm proof to determine the axioms and previous theorems used in the proof. In the case of sylan9eq two previous results are used: syl2an[10] and eqtr.[11] These two previous results can be assumed to have been translated earlier to yield the following two closed propositions:

- $\mathsf{syl2an} : \forall\,(\varphi\ \psi\ \xi\ \theta\ \tau : o).\ \mathsf{wi}\ \varphi\ \psi \to \mathsf{wi}\ \tau\ \xi \to \mathsf{wi}\ (\mathsf{wa}\ \psi\ \xi)\ \theta \to \mathsf{wi}\ (\mathsf{wa}\ \varphi\ \tau)\ \theta$
- $\mathsf{eqtr} : \forall\,(A\ B\ C : \iota \to o).\ \mathsf{wi}\ (\mathsf{wa}\ (\mathsf{wceq}\ A\ B)\ (\mathsf{wceq}\ B\ C))\ (\mathsf{wceq}\ A\ C)$

The TH0 problem given by v1 translation of sylan9eq declares syl2an and eqtr (in the form shown above) as axioms and declares the v1 translation of the theorem (as shown above) as the conjecture. (The reader can easily check that the conjecture follows from the two axioms.)

We next consider the theorem axi4. This example has no premises and one wff variable φ and one set variable x. In this case, the wff φ has a dependence on a set variable and so has type $\iota \to o$ (as opposed to o in the previous example). The conclusion of the theorem is wi (wal ($\lambda x.\varphi\ x$)) ($\varphi\ x$). The v1 translation quantifies over φ and x to create the closed proposition $\forall\varphi : \iota \to o.\forall x : \iota.\mathsf{wi}\ (\mathsf{wal}\ (\lambda x.\varphi\ x))\ (\varphi\ x)$. The set.mm proof uses one previous result, called sp,[12] which the v1 translation of is precisely the same as the closed proposition above, making the TH0 theorem proving problem trivial (the single axiom is the same as the conjecture).

4.2 Translation v2

Since wi, wa and wceq are intended to correspond to implication, conjunction and equality (on classes), we also created translations that make use of this intention. Translation v2 behaves as v1 except that for 10 constructors (corresponding to true, false, implication, conjunction, equivalence, negation, disjunction, equality, universal quantification and existential quantification) are translated using their intended meaning. In particular wi φ ψ translate as $\varphi' \to \psi'$ (where φ' is the v2 translation of φ and ψ' is the v2 translation of ψ). Likewise, wa φ ψ and wb φ ψ translate as $\varphi' \wedge \psi'$ and $\varphi' \leftrightarrow \psi'$. Similarly, wceq A B translates to $A' = B'$ where A' and B' are the v2 translation of A and B. For the universal quantifier, wal ($\lambda x.\ \varphi$) is translated to $\forall x.\ \varphi'$ where φ' is the v2 translation of φ.[13]

[9] https://us.metamath.org/mpeuni/sylan9eq.html
[10] https://us.metamath.org/mpeuni/syl2an.html
[11] https://us.metamath.org/mpeuni/eqtr.html
[12] https://us.metamath.org/mpeuni/sp.html
[13] We can assume the argument is of the form $\lambda x.\varphi$ by η-expansion.

4.3 Translation v3

Translation v3 behaves as v2 except that 11 more constructors are translated using their intended meaning, including w3a, wsb and cab. The v3 translation of w3a $\varphi\ \psi\ \xi$ is $\varphi' \wedge \psi' \wedge \xi'$ where φ', ψ' and ξ' are the v3 translations of φ, ψ and ξ. The v3 translation of wsb $(\lambda x.\ \varphi)\ y$ is $(\lambda x.\ \varphi')\ y$ which β-reduces to φ'^{x}_{y}.[14] This corresponds to substituting y for x in the (translation of the) formula φ. A term of the form cab $(\lambda x.\ \varphi)$ is meant to return the class $\{x \mid \varphi\}$. Since classes are predicates of type $\iota \to o$ and membership of a set in a class corresponds to application of the predicate to the set, the v3 translation of cab $(\lambda x.\ \varphi)\ y$ is simply taken to be φ'^{x}_{y}, treating it essentially the same way as wsb. Using η-reduction, we can also say wsb $(\lambda x.\ \varphi)$ and cab $(\lambda x.\ \varphi)$ v3 translate to $\lambda x.\ \varphi'$.

The v2 and v3 translations of the conjecture for sylan9eq are both

$$\forall\,(\varphi\ \psi : o)\ (A\ B : \iota \to o).\ \ (\varphi \to A = B) \to (\psi \to B = C) \to \varphi \wedge \psi \to A = C.$$

The two TH0 problems (for v2 and v3) also include the translation of the two dependencies syl2an and eqtr as axioms, though these are no longer needed for the proof.

4.4 More Examples

The v2 and v3 translations of the conjecture for axi4 are both

$$\forall\,(\varphi : \iota \to o)\ (x : \iota).\ \ (\forall x.\ \varphi\ x) \to \varphi\ x.$$

Again, the dependency is also translated and included in as an axiom in the TH0 problem, though the axiom is no longer needed to prove the conjecture.

To see the distinction between the v2 and v3 translations, we briefly consider three more small examples: rp_simp2,[15] sbt[16] and abbi2i.[17] The three translations of the conjecture for rp_simp2 are as follows:

- v1: $\forall\,(\varphi\ \psi\ \xi : o).\ \ \mathsf{wi}\ (\mathsf{w3a}\ \varphi\ \psi\ \xi)\ \psi$
- v2: $\forall\,(\varphi\ \psi\ \xi : o).\ \ \mathsf{w3a}\ \varphi\ \psi\ \xi\ \to\ \psi$
- v3: $\forall\,(\varphi\ \psi\ \xi : o).\ \ \varphi\ \wedge\ \psi\ \wedge\ \xi\ \to\ \psi$

The three translations of the conjecture for sbt are as follows:

- v1 and v2: $\forall\,(\varphi : \iota \to \iota \to o).\ \ (\forall x\ y : \iota.\ \varphi\ x\ y) \to \forall y : \iota.\ \mathsf{wsb}\ (\lambda z.\varphi\ z\ y)\ y$
- v3: $\forall\,(\varphi : \iota \to \iota \to o).\ \ (\forall x\ y : \iota.\ \varphi\ x\ y) \to \forall y : \iota.\ \varphi\ y\ y$

The three translations of the conjecture for abbi2i are as follows:

- v1: $\forall\,(\varphi : \iota \to o)\ (A : \iota \to o).\ \ (\forall x : \iota.\ \mathsf{wb}\ (\mathsf{wcel}\ (\mathsf{cv}\ x)\ A)\ (\varphi\ x)) \to \mathsf{wceq}\ A\ (\mathsf{cab}\ (\lambda x.\varphi x))$
- v2: $\forall\,(\varphi : \iota \to o)\ (A : \iota \to o).\ \ (\forall x : \iota.\ (\mathsf{wcel}\ (\mathsf{cv}\ x)\ A)\ \leftrightarrow\ (\varphi\ x)) \to A\ =\ \mathsf{cab}\ (\lambda x.\varphi x)$
- v3: $\forall\,(\varphi : \iota \to o)\ (A : \iota \to o).\ \ (\forall x : \iota.\ (\mathsf{wcel}\ (\mathsf{cv}\ x)\ A)\ \leftrightarrow\ (\varphi\ x)) \to A\ =\ (\lambda x.\varphi x)$

4.5 Why three translations?

The translations v1-v3 represent increasingly "deep" interpretation of the Metamath formulas. One might wonder why we are considering all of them, instead of just using the best one – clearly giving the prover more information is a good idea. (And as 7 will show, this is largely correct.) However, there are three complicating factors that make it not a completely one sided tradeoff:

---

[14] We can assume the second argument is a variable y since it has type ι.

[15] https://us.metamath.org/mpeuni/rp-simp2.html

[16] https://us.metamath.org/mpeuni/sbt.html

[17] https://us.metamath.org/mpeuni/abbi2i.html

- If the prover is given more information, it has more options, and this can cause it to run away and prove the wrong things.
- Also following from the previous point, the prover will be more "creative" with the more deeply embedded proofs, performing more normalization and often resulting in longer proofs than if it is forced to play by Metamath rules.
- Most pertinently for our hammer system, the deeper translations require more of the prover's mechanisms to be translatable back to Metamath proofs, and since only the simpler mechanisms have reconstruction implemented for them, using the deeper translations can cause the reconstruction rate to decrease, even though the ATP has a higher success rate, since it will come up with proofs outside the translatable fragment.

5 Translating Metamath in HOL to First-Order Class Theory

We additionally translated the higher-order representation of set.mm into first-order theorem proving problems by interpreting propositions and terms as classes. This allows us to compare the performance of first-order and higher-order ATPs on problems coming from a common source. In addition we use the first-order prover Prover9 [19] to obtain IVY proof objects which we use to reconstruct Metamath proofs.

Recall that the type of cab in the HOL representation is $(\mathsf{setvar} \to \mathsf{wff}) \to \mathsf{class}$. In the resulting TH0 for the v1 and v2 translations terms with cab at the head could always be assumed to be of the form $\mathsf{cab}\ (\lambda x.\ \varphi)$ (by η-expansion if necessary). We do not have such term level binders in first-order terms, so we must find an alternative method to handle such binders. Every occurrence of a wff or class will be under n setvar binders, binding $x_0, \dots, x_{n-1} \in V$, where V is the class of all sets. Instead of making the binders explicit in the logic, we translate a wff φ in context $x_0, \dots, x_{n-1}$ as the class $\{((x_0, \dots, x_{n-1}), \emptyset) \mid \varphi\}$ (i.e., the class of all n-tuples of sets for which φ holds, with an associated dummy value of $\emptyset$). Likewise we translate a class A in context $x_0, \dots, x_{n-1}$ as the class $\{((x_0, \dots, x_{n-1}), y) \mid y \in A\}$. The operator cab takes a wff in context $x_0, \dots, x_{n-1}, x_n$ to a class in context $x_0, \dots, x_{n-1}$, where n need not be known in advance. In particular we take $\mathsf{cab}(B)$ to be the class

$$\{((x_0, \dots, x_{n-1}), x_n) \mid n \in \omega,\ ((x_0, \dots, x_{n-1}, x_n), \emptyset) \in B\}.$$

For example, suppose we have a HOL version of a Metamath term of the form $\mathsf{cab}\ (\lambda x_n.\ \varphi)$ in a context $x_0, \dots, x_{n-1}$. We can translate φ in context $x_0, \dots, x_{n-1}, x_n$ to obtain a first-order term φ' representing the class $\{((x_0, \dots, x_{n-1}, x_n), \emptyset) \mid \varphi'\}$. We then translate $\mathsf{cab}\ (\lambda x_n.\ \varphi)$ simply to be the first-order term $\mathsf{cab}(\varphi')$, corresponding to the class

$$\{((x_0, \dots, x_{n-1}), x_n) \mid ((x_0, \dots, x_{n-1}, x_n), \emptyset) \in \varphi'\}.$$

The distinction between sets and proper classes is not useful as we will generally be concerned with sets in a context of a certain length. For example, the proper class $\{((x_0, x_1), y) \mid y \in x_1\}$ can be considered a set in a context of length 2 since if $x_0, x_1 \in V$ are fixed, then $\{y \mid y \in x_1\}$ is a set. We say a class A is a set in a context of length n if $\{y \mid ((x_0, \dots, x_{n-1}), y) \in A\}$ is a set for every $x_0, \dots, x_{n-1} \in V$. Note that a class can be a set in a context of different lengths. For example, the empty class is a set in a context of length n for every $n \in \omega$. Consider the Metamath wff wtru (corresponding to the true wff). This will be translated to a first-order constant wtru intended to be the class $\{((x_0, \dots, x_{n-1}), \emptyset) \mid x_0, \dots, x_{n-1} \in V\}$. Note that this is both a proper class and a set in a context of length n for every $n \in \omega$.

In general Metamath constructors with functional arguments may change the length of the context, as cab does. However, most operations do not change the length of the context. For example, $\mathsf{wa}(\varphi, \psi)$ in context $x_0, \ldots, x_{n-1}$ is simply $\mathsf{wa}(\varphi', \psi')$ where φ and ψ are the translations of φ and ψ in context $x_0, \ldots, x_{n-1}$. The intended semantics of $\mathsf{wa}(B, C)$ is

$$\{((x_0, \ldots, x_{n-1}), \emptyset) \mid ((x_0, \ldots, x_{n-1}), \emptyset) \in B \wedge ((x_0, \ldots, x_{n-1}), \emptyset) \in C\}.$$

Note that if φ' is a class $\{((x_0, \ldots, x_{n-1}), \emptyset) \mid \varphi\}$, ψ' is a class $\{((x_0, \ldots, x_{n-1}), \emptyset) \mid \psi\}$ and $\mathsf{wa}(\varphi', \psi')$ is the class $\{((x_0, \ldots, x_{n-1}), \emptyset) \mid \varphi \wedge \psi\}$. As with cab, wa does not depend on the length of the context n.

As a consequence of treating setvar binders in this way, we must decide how to translate the (now implicitly) bound variables as first-order terms. We do this by simply having a constant $\mathtt{var}_n^i$ for each $i < n$, intended to correspond to the variable x_i in the context $x_0, \ldots, x_{n-1}$. Semantically, $\mathtt{var}_n^i$ is the class $\{((x_0, \ldots, x_{n-1}), y) \mid y \in x_i\}$.

Since many wff and class variables depend on a context of set variables, we also include functions $\mathtt{eval}_n^m$ of arity $m + 1$. The intention is that the first argument of $\mathtt{eval}_n^m$ is a class (intended to be in context $x_0, \ldots, x_{m-1}$) and the next m arguments are sets in a context of length n (intended to be in context $x_0, \ldots, x_{n-1}$). Since no Metamath constructor yields a set, the only sets available in context $x_0, \ldots, x_{n-1}$ are each x_i (represented in first-order by the constant $\mathtt{var}_n^i$ and possibly some universal first-order variables intended to range over sets. Each first-order variable Y intended to range over sets will occur as $\mathsf{cv}(Y)$ (the coercion sending sets to classes). We consider a slightly different semantics of cv in the first-order class theory translation than the higher-order case. For a set Y, we take $\mathsf{cv}(Y)$ to be the class

$$\{((x_0, \ldots, x_{n-1}), y) \mid n \in \omega, x_0, \ldots, x_{n-1} \in V, y \in Y\}.$$

That is, cv lifts a set to a class in an arbitrary context (with no dependence on the set variables in the context).

The result of applying $\mathtt{eval}_n^m$ to $m + 1$ arguments is a class (intended to be in context $x_0, \ldots, x_{n-1}$) obtained by composition. In particular, we define $\mathtt{eval}_n^m$ as

$$\begin{aligned}\mathtt{eval}_n^m(B, A_0, \ldots, A_{m-1}) = \\ \{(x_0, \ldots, x_{n-1}, z) \mid \exists y_0, \ldots, y_{m-1} \in V.\ (y_0, \ldots, y_{m-1}, z) \in B \\ \wedge (\forall y.\ y \in y_0 \Leftrightarrow (x_0, \ldots, x_{n-1}, y) \in A_0) \\ \wedge \cdots \\ \wedge (\forall y.\ y \in y_{m-1} \Leftrightarrow (x_0, \ldots, x_{n-1}, y) \in A_{m-1})\}.\end{aligned}$$

Note that if some A_i were not a set in a context of length n, then $\mathtt{eval}_n^m(B, A_0, \ldots, A_{m-1}) = \emptyset$ since a corresponding $y_i \in V$ would not exist. This is never relevant in practice as the only arguments to $\mathtt{eval}_n^m$ after the first argument are of the form $\mathtt{var}_n^i$ or $\mathsf{cv}(Y)$, as stated above.

In case we have a Metamath wff variable φ that depends on m set variables, then occurrences of φ in the higher-order representation will be applied to m arguments. To make this first-order we simply choose n to be the length of the context in which the wff occurs and translate as $\mathtt{eval}_n^m(\varphi', A_0, \ldots, A_{m-1})$ where φ' is a first-order variable corresponding to φ and A_i is the first-order term obtained by translating the arguments of φ.

Some special identities are easy to verify given the semantics described above, e.g.,

$$\mathtt{eval}_n^m(\mathtt{var}_m^i, A_0, \ldots, A_{m-1}) = A_i$$

and

$$\mathtt{eval}_n^m(\mathsf{wa}(B, C), A_0, \ldots, A_{m-1}) = \mathsf{wa}(\mathtt{eval}_n^m(B, A_0, \ldots, A_{m-1}), \mathtt{eval}_n^m(C, A_0, \ldots, A_{m-1}))$$

where we assume each A_i is a set in a context of length n. Some first order problems resulting from the translation only become provable if such identities are included. However, for now we have primarily focused on the problems solvable without such identities included.

As a final step to obtain a first-order atomic proposition, we apply a unary predicate $\mathtt{p}$ to a first-order term φ' (the translation of a Metamath wff φ in an empty context). The intention is that $\mathtt{p}(A)$ should be true precisely if $((), \emptyset) \in A$.

The hypotheses and conclusion of a Metamath axiom or theorem may have universally bound set variables. To translate these to terms we use a unary function $\mathtt{all}_n$. The intended semantics of $\mathtt{all}_n(A)$ is

$$\{((x_0, \ldots, x_{n-1}), \emptyset) \mid \forall x_n \in V.\ ((x_0, \ldots, x_{n-1}, x_n), \emptyset) \in A\}.$$

Again for some translated problems to be theorems we would need to include certain properties of $\mathtt{all}_n$, e.g.,

$$\forall Y.\ \mathtt{p}(Y) \Leftrightarrow \forall X.\ \mathtt{p}(\mathtt{eval}_0^1(Y, \mathsf{cv}(X))).$$

In practice we omit these extra properties for now.

We again consider the example theorem sylan9eq. In the first-order version the two wff variables φ and ψ and the two class variables A and B all range over classes (and hence are represented simply by first-order variables). The two premises translate to first-order terms terms $\mathsf{wi}(\varphi, \mathsf{wceq}(A, B))$ and $\mathsf{wi}(\psi, \mathsf{wceq}(B, C))$. which can then be used as arguments to $\mathtt{p}$ to obtain atomic propositions. The conclusion translates to the first order term $\mathsf{wi}(\mathsf{wa}(\varphi, \psi), \mathsf{wceq}(A, C))$. Combining the premises with the conclusion and quantifying over the variables yields the first-order sentence

$$\forall\, \varphi\ \psi\ A\ B\ C.\ \ \mathtt{p}(\mathsf{wi}(\varphi, \mathsf{wceq}(A, B))) \to \mathtt{p}(\mathsf{wi}(\psi, \mathsf{wceq}(B, C))) \to \mathtt{p}(\mathsf{wi}(\mathsf{wa}(\varphi, \psi), \mathsf{wceq}(A, C))).$$

As before the sentence is a consequence of syl2an and eqtr which translate to the first-order sentences

$$\forall\, \varphi\ \psi\ \xi\ \theta\ \tau.\ \ \mathtt{p}(\mathsf{wi}(\varphi, \psi)) \to \mathtt{p}(\mathsf{wi}(\tau, \xi)) \to \mathtt{p}(\mathsf{wi}(\mathsf{wa}(\psi, \xi), \theta)) \to \mathtt{p}(\mathsf{wi}(\mathsf{wa}(\varphi, \tau), \theta))$$

and

$$\forall\, A\ B\ C.\ \ \mathtt{p}(\mathsf{wi}(\mathsf{wa}(\mathsf{wceq}(A, B), \mathsf{wceq}(B, C)), \mathsf{wceq}(A, C))).$$

We also reconsider the theorem axi4. This translates to the first-order statement

$$\forall\, \varphi\ x.\ \ \mathtt{p}(\mathsf{wi}(\mathsf{wal}(\mathtt{eval}_1^1(\varphi, \mathtt{var}_1^0)), \mathtt{eval}_0^1(\varphi, x))).$$

Again the proof uses sp which translates to the same first-order statement (making the theorem proving problem trivial).

6 Benchmark

We use the translations to TPTP described in Sections 4 and 5 to create higher-order and first-order ATP problems. This is implemented in Lisp, as a program that reads the HOL Lisp representation of the MM0 (Section 3) version of set.mm as its input, and produces the corresponding ATP problem for each Metamath theorem proved in set.mm .

The version of set.mm we used corresponds to the git repo with a commit from June 24, 2022.[18] There are 40338 theorems with proofs in this version of set.mm. The translation to MM0 increases the number of theorems to 40556 (218 extra theorems) since some theorems also have α-degenerate versions that are used in their degenerated form later in the library. For example, the Metamath set.mm theorem `ax7v` is $x = y \to x = z \to y = z$ where x and y must be distinct variables. The MM0 version expands this into three versions:

- `ax7v`: $x = y \to x = z \to y = z$
- `ax7v_b` (an α-degenerate): $x = y \to x = x \to y = x$
- `ax7v_b1` (another α-degenerate): $x = y \to x = y \to y = y$

There are also three corresponding TH0 problems (in each of v1, v2 and v3).

The axioms of each TH0 problem are determined by the named facts (proof-external facts, premises) used in the MM0 proof. Note that when generating the problem for a given theorem we already have the TH0 formulas corresponding to the previous facts. Note also that the translation from Metamath to MM0 already distinguished between α-degenerates. While a Metamath proof may have depended on `ax7v`, its MM0 proof may depend on `ax7v` and one of its α-degenerates, say `ax7v_b`. In that case the TH0 problem would include axioms corresponding to `ax7v` and `ax7v_b` (but not `ax7v_b1`).

In the end we obtain 40556 TH0 problems for each of v1, v2 and v3.[19] This corresponds precisely to the 40556 MM0 theorems obtained by translating set.mm. To this we also add the first-order version produced by the translation described in Section 5.

7 Initial ATP Evaluation

7.1 Higher-order Evaluation

We first evaluate three top-performing[20] higher-order ATPs on the three higher-order versions of the problems using several values for timeout. Only one full evaluation is done on v1 of the problems, since we consider the encoding suboptimal. Indeed, E-HO 2.6 solves 77.65% (20352 vs 11456) more problems on v2 with v1 adding only 0.23% (46) more solutions to v2. All our experiments are performed on a server with 36 hyperthreading Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz cores and 256 GB of RAM. We use the following ATP systems:

- E prover version 2.6 [25, 24], run both in its default portfolio (auto-schedule) mode and with several strategies developed previously by strategy invention systems [26, 13] targeting ITP libraries [16, 12].
- Vampire version 5980 [17, 2], using mainly default (casc2020) higher-order portfolio. We have also briefly tested some Vampire strategies in a standalone mode.
- Zipperposition version CASC20 [6, 28], using its default CASC'20 portfolio. We have also tried several other Zipperposition settings.

For most of the experiments we have used a time limit of 60s. We have also used initially a lower 10s time limit, and later also higher time limits, especially to see the performance of the portfolio-based systems. The highest time used for an evaluation on the full dataset was 280s, and we have increased it to 600s and 1200s in several cases to see the improvement on unsolved problems by E, Vampire and Zipperposition. Of the 40556 problems the ATPs can in total solve 27436, i.e., 67.65%. The detailed results of the complete runs of the systems are shown in Table 1. Table 2 shows the top-5 greedy cover including incomplete runs done with high time limits on unsolved problems only.

[18] Specifically the commit d75c0dbe.
[19] The benchmark is publicly available at `https://github.com/ai4reason/mm-atp-benchmark` .
[20] `https://www.tptp.org/CASC/J11/WWWFiles/DivisionSummary1.html`

The highest performance is achieved by Zipperposition which in 280s solves 62.68% (25420) of the v3 problems, and 61.53% (24959) of the v2 problems. This (v2) drops to 57.99% (23518) when using half of the time only, i.e. 140s. Vampire solves 58.01% (23555) of the v3 problems in 280s which is a surprisingly good performance given the 2022 CASC results,[21] where Vampire loses more than 25% on Zipperposition (367 vs 460 problems solved in the CASC THF category). Note that none of the ATP developers have yet seen our benchmark and thus could not develop targeted ATP strategies on the problems, as is typically done for TPTP and CASC. Vampire also gains considerably by going from v2 to v3 and by increasing the time limit (the latter likely thanks to its large portfolio mode). It solves only 45.57% (18482) of the v2 problems in 60s, which increases to 52.08% (21123) of the v3 problems in 60s, and to 56.65% (22976) v3 problems in 120s. Furthermore, Vampire gains from running its strongest strategies with higher time limits: 7 of the strategies run separately for 60s on v3 add together 443 problems, raising Vampire's performance to 23998 problems (in general in 240s + 7*60s = 660s).

E prover outperforms Vampire on v2 in 60s (21001 solved by E vs 18482 by Vampire), and even more so in 10s (20352 vs 17160). Surprisingly, E's performance is lower on v3 compared to v2 (20799 vs 21001 in 60s). The performance however does not increase much with higher time limits as in the case of Vampire, indicating that Vampire makes better use of multiple strategies. We have however only evaluated the official E version 2.6, which seems to have been improved a lot very recently by E version 3.0 in the latest 2022 CASC results. We plan to evaluate E 3.0 when it is officially released.

Table 1 The complete runs of the systems on the benchmark, ordered by performance.

System	mode	version	time (s)	solved
Z	portfolio	v3	280	25420
Z	portfolio	v2	280	24959
V	portfolio	v3	280	23555
Z	portfolio	v2	140	23518
V	portfolio	v3	120	22976
V	portfolio	v3	60	21123
E	portfolio	v2	60	21001
E	portfolio	v3	60	20799
E	portfolio	v2	10	20352
E	strat. f17	v3	120	19782
E	strat. f17	v2	10	19624
V	portfolio	v2	60	18482
Z	fo-complete-basic	v2	10	17295
V	portfolio	v2	10	17160
Z	ho-pragmatic	v2	10	16115
E	portfolio	v1	10	11456

7.2 First-order runs

We also evaluate the first order translation, by running Vampire, E and Prover9 on these problems. Vampire proves 15938 of them, while E and Prover9 solve 15136 and 14693 respectively. The Vampire performance can be compared to its 60s performance on the v2 higher-order problems (18482). This likely again demonstrates the efficiency of the v2 and v3 higher-order translations, because practically none of the standard logical connectives are

[21] See footnote 20.

Table 2 The top 5 methods in the greedy sequence. Note that we use different (and also high) time limits and that the high-time runs are only done on previously unsolved problems.

System	mode	version	time (s)	added	sum
Z	portfolio	v3	280	25420	25420
V	portfolio	v3	600	960	26380
V	portfolio	v3	1200	415	26795
E	portfolio	v3	600	279	27074
Z	portfolio	v2	280	124	27198

mapped in a shallow way to their first-order logical counterparts in this first-order translation. Because of that, all these problems are also Horn, and they are also quite small due to the minimized premises. This is likely the reason why all the three ATPs perform similarly here, with Vampire and E unable to benefit from their strategies for large problems with large non-Horn clauses. The systems are also not very complementary, with E adding 148 and Prover9 adding 110 problems to Vampire. Prover9 adds 505 problems to E.

7.3 Premise Selection Experiments

7.3.1 Higher-order runs

We evaluate Vampire also on the large (*chainy*) versions of the higher-order v3 problems, using a 60s time limit. This emulates the hammer-style reasoning with the whole library that exists before a particular theorem was stated. Using its LTB (large-theory batch division of the CASC competition) portfolio, Vampire solves 8509 v3 problems, while its hol mode solves only 4013 v3 problems. This is relatively few compared to the performance on the benchmark with preselected premises.

Vampire uses SInE as its default premise selector. A number of learning-based premise selection methods typically improve on SInE if enough data about previous proofs are available to train on. For simplicity of use and comparison with other standard hammers we have decided to use here a fast implementation of the distance weighted k-nearest neighbor (k-NN), parameterized by several values of k (10, 20, 40, 80, 120, 160, 240) that typically work well with current ATPs. The first evaluation is done chronologically, by training k-NN incrementally on the human-written proofs as the library grows in time. This means that we always allow k-NN to see all the facts and proofs that precede the fact for which it is predicting the premises, but none of the facts and proofs that follow after that.

We use again the standard higher-order portfolio of Vampire and 60s time limit for each of the values of k. The results are shown in Table 3. The performance peaks at around 12k proved problems for the values of $k = 120, 160, 80$. This is more than 40% better than the best SInE result above (8509). All seven k-NN predictions solve together 14787 problems (in general in 7 minutes), with the top three most orthogonal slices (120, 240, and 20) solving 14113 (in general in 3 minutes).

Table 3 Vampire on k-NN premise selection slices.

Premises	10	20	40	80	120	160	240
V-thf v3	9112	10078	11060	11863	12043	11997	11582
V-fof v1	2600	4239	6294	8366	9416	9875	10352

7.3.2 First-order runs

To get a version that can be reconstructed in Metamath, we also evaluate Vampire on the premise selection slices of the large (chainy) first-order versions of the problems. The results for the 7 standard slices are again shown in Table 3. Unlike in the thf-v3 version, Vampires benefits here from increasingly large slices, so we add also slices with 480 and 960 premises to the fof-v1 premise selection. These slices solve 10726 and 10593 problems respectively, adding many new solutions. In total, the 9 first-order slices solve 12373 problems, with the top 4 most complementary slices (480, 960, 80, and 240) solving 12089 problems. This implies a 30% performance in the first-order v1 hammering setting, which we currently use for the first version of the proof reconstruction.

8 Proof Reconstruction

While Vampire has impressive solving capabilities, we were not able to get it to produce a proof object which was sufficiently detailed for our purposes, so we instead turned to the IVY proof format used by Prover9. Prover9 is not as powerful, but we can still use Vampire as a more precise relevance filter by using the lemmas from the proof it produces as input to Prover9, and process the resulting IVY proof.

IVY is a resolution-style proof format for doing classical reasoning, so it is not a priori obvious how to reconstruct these terms into a Metamath proof without a deep embedding. However, our input clauses and the conjecture are all Horn clauses (that is, they have at most one non-negated literal), and this makes all the difference.

IVY proofs consist of the following kind of proof steps:

- `input` steps refer to one of the hypotheses, except that instead of using $\forall \vec{x}.\ \overrightarrow{A_i(\vec{x})} \to B(\vec{x})$, the quantifiers are removed and the clauses are turned into disjunctions, as in $B(\vec{v}) \vee \bigvee_i \neg A_i(\vec{v})$, with the literals possibly reordered.
  - Because these inputs appear in the same order as they were given to the checker, they are easy to identify.
  - The conjecture is negated, so it turns into multiple inputs and the variables are skolemized: $\forall \vec{x}.\ \overrightarrow{A'_i(\vec{x})} \to B'$ becomes $A'_i(\vec{c})$ for each i, plus $\neg B'(\vec{c})$.
- `instantiate` steps refer to a previous step $p : \bigvee_i C_i$ plus a substitution mapping $\{v_i \mapsto t_i\}$ and results in a proof of $\bigvee_i (C_i[v_i \mapsto t_i])$.
- `resolve` steps specify $p : \bigvee_i C_i \vee P$ and $q : \bigvee_i D_i \vee \neg P$ (where P may appear in the middle of the disjunction but is identified by a path), and results in a proof of $\bigvee_i C_i \vee \bigvee_i D_i$.
- `propositional` steps prove an arbitrary clause Q from previous step $p : P$ where $P \to Q$ is a propositional tautology.
- IVY also supports `new_symbol`, `flip`, and `paramod` steps but these never appear in reconstructed proofs.

The key observation is that IVY never leaves the realm of Horn clauses in the proof. This is not syntactically a requirement – proofs can in principle involve arbitrary propositions – but we can see why it might happen with this kind of input:

- All the inputs are Horn clauses.
- `instantiate` or `resolve` on Horn clauses yield more Horn clauses.
- While `propositional` steps can yield non-Horn clauses in principle, this is mainly used for clause simplification, and there is a unique best clause that the solver will want to generate here, namely the input clause with duplicate hypotheses removed. That is, this is used only for simplifying $\neg A \vee \neg A \vee B$ to $\neg A \vee B$.
- The fact that the clauses have this restricted form is likely the reason why we do not observe the more advanced kinds of steps.

So our strategy for reconstruction is essentially to interpret these as proofs in minimal logic or terms in the simply typed lambda calculus, where $(\bigvee_i \neg A_i) \vee B$ is interpreted as $(\bigwedge_i A_i) \to B$ and $\bigvee_i \neg A_i$ is interpreted as $(\bigwedge_i A_i) \to \mathbf{F}$. The proof steps all have associated lambda terms:

- Hypothesis inputs are $H_i(\vec{v}) : (\bigwedge_i A_i(\vec{v})) \to B(\vec{v})$
- The conjecture is $\mathbf{thm} : B'(\vec{c}) \to \mathbf{F}$
- `instantiate`$(p, \{v_i \mapsto t_i\})$ is just $p[v_i \mapsto t_i]$
- For `resolve`(p, q): if $p : \bigwedge_i C_i \to P$ and $q : (\bigwedge_i D_i) \wedge P \to A$ or $q : (\bigwedge_i D_i) \wedge P \to \mathbf{F}$, then `resolve`$(p, q) := \lambda \overrightarrow{c_i}, \overrightarrow{d_i}.\ q(\overrightarrow{d_i}, p(\overrightarrow{c_i}))$.
- For `propositional` steps we spot the duplicates and generate a term like $\lambda x\, y.\ p(x, x, y)$.

Because the final result is a proof of `false`, we get a closed term of type $\mathbf{F}$ after translation, and we can normalize it to eliminate all the lambdas. Since the only constructor for $\mathbf{F}$ in this grammar is **thm**, the result will be of the form $\mathbf{thm}(p)$, where p is a proof structured out of applications of H_i to a substitution and a list of subproofs, which is exactly the form expected by Metamath proofs. So we strip the **thm** node and the result is a well formed proof.

8.1 Proof Objects

Table 4 shows the longest IVY and Metamath proof objects obtained in the experiments. This is for IVY measured by the number of proof steps, while for Metamath these are lines of the reconstructed proof.

Table 4 Length of the longest proof objects in IVY steps and Metamath lines.

Problem	mercolem6	tgbtwnconn1lem1	hdmap14lem9	isoas	lclkrlem2a
IVY	674	480	392	375	316
Problem	mercolem6	mercolem2	merlem5	mercolem7	minimp_sylsimp
Metamath	5660830	849	77	50	45

An outlier here is `mercolem6`, which is one lemma in the proof that Meredith's axiom

$$((\varphi \to \psi) \to (\bot \to \chi) \to \theta) \to (\theta \to \varphi) \to \tau \to \eta \to \varphi$$

is complete for propositional logic. Prover9 is able to return a proof with only 674 lines, but it balloons to a massive 5 660 830 lines after Metamath reconstruction, over 7 times the size of `set.mm`. The reason for this due to the normalization process described in section 8. Each Metamath proof step is exactly and only an application of a previous theorem, with substitutions for the variables, and then proofs for the hypotheses. That is, in IVY terminology we are structurally required to perform `instantiate` steps only on the leaves of the proof.

What happened in this proof is that Prover9 found a useful *lemma*, which has a long proof, and then applied it many times with different instantiations, and the Metamath proof is forced to replicate the subproof many times in order to push the instantiations to the leaves. This is essentially an artificial restriction caused by our implicit requirement that the hammer should generate *one proof*, rather than a sequence of lemmas leading up to the proof. In actual practice a user would split the proof at this useful lemma and refer to it. In fact, the name `mercolem6` indicates that this is lemma 6 of something, so this technique is already being used here.

Future versions of the hammer may include this kind of lemma generation, but we decided not to pursue it since it is extremely rare. Most of the time the cost of extracting these narrow lemmas is higher than the proof savings for applying them.

9 Examples

Three similar examples Zipperposition and E can prove in the *v3* representations are the set.mm theorems amgm2d[22], amgm3d[23] and amgm4d[24] comparing arithmetic and geometric means. The first theorem states that for positive reals A and B,

$$(A \cdot B)^{\frac{1}{2}} \leq \frac{A+B}{2}.$$

The next two theorems state

$$(A \cdot B \cdot C)^{\frac{1}{3}} \leq \frac{A+B+C}{3}$$

and

$$(A \cdot B \cdot C \cdot D)^{\frac{1}{4}} \leq \frac{A+B+C+D}{4}$$

for positive reals A, B, C and (in the last case) D. All three theorems are proven by making use of a lemma amgmlem[25] giving the property

$$(\Sigma^M F)^{\frac{1}{|A|}} \leq \frac{\Sigma^{\mathbb{C}} F}{|A|}$$

where A is a finite set, F is a function from A to positive reals, and Σ^Q is performs a binary operation from a given monoid Q to the images of F. In this case $\mathbb{C}$ is the complex field (thought of as its additive group here) and so $\Sigma^{\mathbb{C}}$ is ordinary summation. However, M is the multiplicative group of $\mathbb{C}$ and so Σ^M in the usual Π operator performing finitely many multiplications. In order for an ATP to prove the examples above, it must instantiate with appropriate values of A and F in the assumption amgmlem, essentially giving the appropriate n-tuple (for $n \in \{2,3,4\}$). In this case the n-tuples are represented as words and special theorems gsumws2[26], gsumws3[27] and gsumws4[28] give equations between applying Σ^Q to an appropriate length word and the summation of the "characters." Applying these theorems when proving amgm2d, amgm3d and amgm4d leads to the generation of the appropriate n-tuple (word) being constructed by the ATP via unification.

It is worth noting Zipperposition, E and Vampire could also prove the *v3* problems corresponding to gsumws2, gsumws3 and gsumws4. By contrast, none of the ATPs could prove the vital lemma amgmlem. Also, none of the ATPs could prove amgmw2d[29], a generalized version of amgm2d stating

$$A^P \cdot B^Q \leq A \cdot P + B \cdot Q$$

for positive reals A, B, P and Q such that $P + Q = 1$.

[22] https://us.metamath.org/mpeuni/amgm2d.html
[23] https://us.metamath.org/mpeuni/amgm3d.html
[24] https://us.metamath.org/mpeuni/amgm4d.html
[25] https://us.metamath.org/mpeuni/amgmw2d.html
[26] https://us.metamath.org/mpeuni/gsumws2.html
[27] https://us.metamath.org/mpeuni/gsumws3.html
[28] https://us.metamath.org/mpeuni/gsumws4.html
[29] https://us.metamath.org/mpeuni/amgmw2d.html

A different example Zipperposition and E can prove is zringunit[30]. This states A is a unit of the ring of integers if and only if A is an integer with norm 1 (i.e., A is -1 or 1). A previous result used in the proof is gzrngunit[31] which states the units of the ring of Gaussian integers is precisely those with norm 1. None of the ATPs were able to prove gzrngunit.

Several other interesting ATP proofs are available on our web page.[32] This includes E's higher-order proof of theorem `xmulneg1`[33] which has 127 steps in Metamath and takes 18131 given clause loops in 30 seconds to E.[34] It proves for extended reals that a product with a negative is the negative of the product:

```
xmulneg1 $p |- ( ( A e. RR* /\ B e. RR* ) -> ( -e A *e B ) = -e ( A *e B ) )
```

E also proves the `matinv` theorem in 12 seconds and 13052 given clause loops, which takes a 73-step proof in Metamath.[35] The theorem states that the inverse of a matrix is the adjunct of the matrix multiplied with the inverse of the determinant of the matrix if the determinant is a unit in the underlying ring:

```
matinv.a $e |- A = ( N Mat R ) $.
matinv.j $e |- J = ( N maAdju R ) $.
matinv.d $e |- D = ( N maDet R ) $.
matinv.b $e |- B = ( Base ‘ A ) $.
matinv.u $e |- U = ( Unit ‘ A ) $.
matinv.v $e |- V = ( Unit ‘ R ) $.
matinv.h $e |- H = ( invr ‘ R ) $.
matinv.i $e |- I = ( invr ‘ A ) $.
matinv.t $e |- .xb = ( .s ‘ A ) $.
matinv $p |- ( ( R e. CRing /\ M e. B /\ ( D ‘ M ) e. V ) ->
    ( M e. U /\ ( I ‘ M ) = ( ( H ‘ ( D ‘ M ) ) .xb ( J ‘ M ) ) ) )
```

Further impressive ATP proofs collected by us include theorems about integrals,[36] triangle inequality,[37] measure,[38] sums of vector spaces,[39] etc. These proofs typically take over one hundred steps in Metamath.

10 Hammer Tool

The mm-hammer tool is publicly available from our GitHub repository[40]. It packages the theorem proving, proof reconstruction and premise selection methods described above for the Metamath users. We provide there also an installer script that installs all the prerequisites (including Prover9 and Vampire).

11 Conclusion

We have developed the first translations of the Metamath set.mm library to the formats used by state-of-the-art higher-order and first-order automated theorem provers. Based on them, we have constructed several versions of a large new benchmark of 40556 mathematical ATP

[30] https://us.metamath.org/mpeuni/zringunit.html
[31] https://us.metamath.org/mpeuni/gzrngunit.html
[32] http://grid01.ciirc.cvut.cz/~mptp/mm_prf/
[33] https://us.metamath.org/mpeuni/xmulneg1.html
[34] http://grid01.ciirc.cvut.cz/~mptp/mm_prf/mmset12407_xmulneg1.p
[35] https://us.metamath.org/mpeuni/matinv.html
[36] https://us.metamath.org/mpeuni/ditgsplit.html
[37] https://us.metamath.org/mpeuni/isxmet2d.html
[38] https://us.metamath.org/mpeuni/sibfinima.html
[39] https://us.metamath.org/mpeuni/mapdlsm.html
[40] https://github.com/digama0/mm-hammer

problems based on set.mm . The initial evaluation of the ATPs is very encouraging. The strongest higher-order system (Zipperposition) proves 62.68% of the problems in 280s, and 57.99% of the problems in 140s. Even when using low (hammer-friendly) time limits, the higher-order ATPs are very useful, with E proving 50.18% of the problems in 10s. These are very encouraging results for providing ATP-based automation for the Metamath authors.

We have also developed the first version of a full hammer tool for Metamath and made it publicly available to the Metamath community. This includes mainly a proof reconstruction tool that imports the Prover9/IVY proof objects into Metamath. The tool already replays all 15k proofs that Prover9 can find when using human-based premises extracted from Metamath. Another component of the hammer is a real-time pipeline that translates Metamath user problems into first-order formats, and runs premise selectors and a portfolio of large-theory Vampires on the problems, followed by running Prover9/IVY on the Vampire-minimized problems when successful. The first version of the tool proves 30% of the Metamath theorems when running the ATPs on four premise selections in parallel for 60 seconds.

References

1 Jesse Alama, Tom Heskes, Daniel Kühlwein, Evgeni Tsivtsivadze, and Josef Urban. Premise selection for mathematics by corpus analysis and kernel methods. *J. Autom. Reasoning*, 52(2):191–213, 2014. doi:10.1007/s10817-013-9286-5.

2 Ahmed Bhayat and Giles Reger. A combinator-based superposition calculus for higher-order logic. In *IJCAR (1)*, volume 12166 of *Lecture Notes in Computer Science*, pages 278–296. Springer, 2020.

3 Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending Sledgehammer with SMT solvers. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *CADE*, volume 6803 of *LNCS*, pages 116–130. Springer, 2011. doi:10.1007/978-3-642-22438-6_11.

4 Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *J. Formalized Reasoning*, 9(1):101–148, 2016. doi:10.6092/issn.1972-5787/4593.

5 Mario Carneiro. Metamath zero: Designing a theorem prover prover. In *Intelligent Computer Mathematics: 13th International Conference, CICM 2020, Bertinoro, Italy, July 26–31, 2020, Proceedings*, pages 71–88, Berlin, Heidelberg, 2020. Springer-Verlag. doi:10.1007/978-3-030-53518-6_5.

6 Simon Cruanes. *Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond*. Theses, École polytechnique, September 2015. URL: https://hal.archives-ouvertes.fr/tel-01223502.

7 Lukasz Czajka and Cezary Kaliszyk. Hammer for coq: Automation for dependent type theory. *J. Autom. Reason.*, 61(1-4):423–453, 2018.

8 Martin Desharnais, Petar Vukmirović, Jasmin Blanchette, and Makarius Wenzel. Seventeen provers under the hammer, 2022. URL: https://matryoshka-project.github.io/pubs/seventeen.pdf.

9 Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. Smtcoq: A plug-in for integrating SMT solvers into coq. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 126–133. Springer, 2017. doi:10.1007/978-3-319-63390-9_7.

10 Thibault Gauthier and Cezary Kaliszyk. Premise selection and external provers for HOL4. In *Certified Programs and Proofs (CPP'15)*, LNCS. Springer, 2015. doi:10.1145/2676724.2693173.

11 Allen Van Gelder and Geoff Sutcliffe. Extending the TPTP language to higher-order logic with automated parser generation. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *LNCS*, pages 156–161. Springer, 2006. doi:10.1007/11814771_15.

12 Thomas C. Hales, John Harrison, Sean McLaughlin, Tobias Nipkow, Steven Obua, and Roland Zumkeller. A revision of the proof of the Kepler conjecture. *Discrete & Computational Geometry*, 44(1):1–34, 2010. doi:10.1007/s00454-009-9148-4.

13 Jan Jakubův and Josef Urban. BliStrTune: hierarchical invention of theorem proving strategies. 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 43–52. ACM, 2017. doi:10.1145/3018610.3018619.

14 Jan Jakubuv, Karel Chvalovský, Zarathustra Amadeus Goertzel, Cezary Kaliszyk, Mirek Olsák, Bartosz Piotrowski, Stephan Schulz, Martin Suda, and Josef Urban. Mizar 60 for mizar 50. *CoRR*, abs/2303.06686, 2023. doi:10.48550/arXiv.2303.06686.

15 Cezary Kaliszyk and Josef Urban. HOL(y)Hammer: Online ATP service for HOL Light. *Mathematics in Computer Science*, 9(1):5–22, 2015.

16 Cezary Kaliszyk and Josef Urban. MizAR 40 for Mizar 40. *Journal of Automated Reasoning*, 55(3):245–256, 2015. doi:10.1007/s10817-015-9330-8.

17 Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *CAV*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013. doi:10.1007/978-3-642-39799-8_1.

18 Daniel Kühlwein, Jasmin Christian Blanchette, Cezary Kaliszyk, and Josef Urban. MaSh: Machine learning for Sledgehammer. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *ITP 2013*, volume 7998 of *LNCS*, pages 35–50. Springer, 2013. doi:10.1007/978-3-642-39634-2_6.

19 William McCune. Prover9 and Mace4, 2005–2010. URL: http://www.cs.unm.edu/~mccune/prover9/.

20 Norman D. Megill and David A. Wheeler. *Metamath: A Computer Language for Mathematical Proofs*. Lulu Press, Morrisville, North Carolina, 2019. http://us.metamath.org/downloads/metamath.pdf.

21 Jia Meng and Lawrence C. Paulson. Translating higher-order clauses to first-order clauses. *J. Autom. Reasoning*, 40(1):35–60, 2008. doi:10.1007/s10817-007-9085-y.

22 Lawrence C. Paulson and Jasmin C. Blanchette. Three years of experience with Sledgehammer, a practical link between automated and interactive theorem provers. In Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska, editors, *Workshop on the Implementation of Logics (IWIL)*, volume 2 of *EPiC*, pages 1–11. EasyChair, 2010. Invited talk. URL: http://www.easychair.org/publications/paper/62805.

23 John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001. URL: https://www.sciencedirect.com/book/9780444508133/handbook-of-automated-reasoning.

24 Stephan Schulz. System description: E 1.8. In Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *LPAR*, volume 8312 of *Lecture Notes in Computer Science*, pages 735–743. Springer, 2013. doi:10.1007/978-3-642-45221-5_49.

25 Stephan Schulz, Simon Cruanes, and Petar Vukmirovic. Faster, higher, stronger: E 2.3. 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 495–507. Springer, 2019. doi:10.1007/978-3-030-29436-6_29.

26 Josef Urban. BliStr: The Blind Strategymaker. In Georg Gottlob, Geoff Sutcliffe, and Andrei Voronkov, editors, *Global Conference on Artificial Intelligence, GCAI 2015, Tbilisi, Georgia, October 16-19, 2015*, volume 36 of *EPiC Series in Computing*, pages 312–319. EasyChair, 2015. URL: http://www.easychair.org/publications/paper/BliStr_The_Blind_Strategymaker, doi:10.29007/8n7m.

27 Josef Urban, Piotr Rudnicki, and Geoff Sutcliffe. ATP and presentation service for Mizar formalizations. *J. Autom. Reasoning*, 50:229–241, 2013. doi:10.1007/s10817-012-9269-y.

28 Petar Vukmirovic, Alexander Bentkamp, Jasmin Blanchette, Simon Cruanes, Visa Nummelin, and Sophie Tourret. Making higher-order superposition work. In *CADE*, volume 12699 of *Lecture Notes in Computer Science*, pages 415–432. Springer, 2021.

Reimplementing Mizar in Rust

Mario Carneiro
Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

This paper describes a new open-source proof processing tool, `mizar-rs`, a wholesale reimplementation of core parts of the Mizar proof system, written in Rust. In particular, the "checker" and "analyzer" of Mizar are implemented, which together form the trusted core of Mizar. This is to our knowledge the first and only external implementation of these components. Thanks to the loose coupling of Mizar's passes, it is possible to use the checker as a drop-in replacement for the original, and we have used this to verify the entire MML in 11.8 minutes on 8 cores, a 4.8× speedup over the original Pascal implementation. Since Mizar is not designed to have a small trusted core, checking Mizar proofs entails following Mizar closely, so our ability to detect bugs is limited. Nevertheless, we were able to find multiple memory errors, four soundness bugs in the original (which were not being exploited in MML), in addition to one non-critical bug which was being exploited in 46 different MML articles. We hope to use this checker as a base for proof export tooling, as well as revitalizing development of the language.

2012 ACM Subject Classification Mathematics of computing → Mathematical software performance; Software and its engineering → Formal methods; Social and professional topics → Systems analysis and design

Keywords and phrases Mizar, proof checker, software, Rust

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.10

Related Version *Previous Version*: `https://arxiv.org/abs/2304.08391`

Supplementary Material *Software (Source Code)*: `https://github.com/digama0/mizar-rs/tree/itp2023/itp2023`, archived at `swh:1:dir:2522beed4f5fce87ce3193ff0359def1dcff1d7c`

Funding Work supported by the Hoskinson Center for Formal Mathematics at Carnegie Mellon.

Acknowledgements I would like to thank Josef Urban and Jeremy Avigad for their support and encouragement, and Adam Naumowicz and SUM for taking the courage to finally open-source Mizar, without which it would not have been possible to publish this paper and accompanying code.

1 Introduction

The Mizar language [3] is a proof language designed by Andrzej Trybulec in 1973 for writing and checking proofs in a block structured natural deduction style. The Mizar project more broadly has been devoted to the development of the language and tooling, in addition to the Mizar Mathematical Library (MML) [2], a compendium of "articles" on a variety of mathematical topics written in the Mizar language. The MML is one of the largest and oldest formal mathematical libraries in existence, containing (at time of writing) 1434 articles and over 65,000 theorems.

The "Mizar system" is a collection of tools for manipulating Mizar articles, used by authors to develop and check articles for correctness, and maintained by the Association of Mizar Users (SUM). Arguably the most important tool in the toolbox is `verifier`, which reads a Mizar article and checks it for logical correctness. The starting point for this work is the goal of extracting formal proofs from the `verifier` which can be checked by a tool not directly connected to Mizar. This turned out to be quite challenging, and this paper will explain how we achieved a slightly different goal – to build `mizar-rs`, a Rust program which checks proofs in the same manner as the `verifier`, with an eye for proof generation.


© Mario Carneiro;
licensed under Creative Commons License CC-BY 4.0

Why hasn't this been done before?

The first challenge is that there is no "spec" for the Mizar language: a Mizar proof is one that `verifier` accepts. Some of it is "obvious" things which users can figure out by playing with the language, and there are reference works at varying levels of detail [5, 10], but if we want to check the entire MML then approximations aren't going to cut it. There are some writings online which give the general sense of the algorithm, but there is nowhere to look for details except the source code.

This leads to the second challenge, which is that the source code for the Mizar system is not publicly available, or at least was not while this project was under development.[1] It was only made available to members of the SUM. Luckily, it is possible to obtain a membership, but the price of admission is that one has to write a Mizar article first. Thus, for this project the author contributed an article regarding the divergence of the sum of prime reciprocals [4].

The third challenge is that Mizar does not have a "small trusted kernel" in any reasonable sense. The main components that perform trusted activities are the "analyzer" and the "checker", and most of the code of the verifier is contained in these modules. Moreover, the code is written in Pascal and Polish (two languages that the author is not very good at), and is 50 years old, meaning that there is a huge amount of technical debt in the code.[2] To get a good sense of what everything is doing and why it is correct (or not), we estimated the best option would simply be to rewrite it and closely follow what the original code is doing. Rust was chosen as the implementation language because it is well suited to writing console applications with a focus on correctness, performance, and refactoring, all of which were important for something that could play the role of `verifier`.

We are not the first to present aspects of Mizar from an outsider's perspective. J. Harrison's Mizar mode in HOL Light [6] is a simulation of large parts of the Mizar frontend, with the MESON prover used in place of the Mizar checker. J. Urban's MPTP [12] project exports the statements of checker goals and sends them to ATPs. And most recently, Kaliszyk and Pąk presented a model of Mizar as encoded in Isabelle [7], but stop short of a fully automatic proof export mechanism.

One may reasonably ask why the MPTP export is not already everything you could wish regarding proof export. There are two problems: (1) Because it is using external ATPs rather than Mizar's own code, it does not and cannot reasonably be expected to achieve 100% success on the MML, and even when it works the Mizar checker often outperforms the external ATP, because the MML is significantly overfit to the Mizar checker. (2) Because it instruments the "checker inference" level, we verify "`by`" steps but not the skeleton steps that glue them together, and some significant type reasoning happens in the analyzer.

In this paper, we will try to give an implementer's perspective of Mizar: how it works under the hood, and some things we discovered while trying to replicate the functionality. The reader should be aware that this paper is focused on internal details and implementation correctness of the Mizar system, not the abstract theory of Mizar (that is, Tarski–Grothendieck set theory and the soft type system). See [5, 7] for information on these aspects of the language.

---

[1] A happy after-effect of this project is that the Mizar source code itself has been made public! It is now available at `https://github.com/MizarProject/system`.

[2] Technically the current incarnation of Mizar ("PC Mizar") only dates to 1986. See [9] for the early history of Mizar.

2 Mizar internals: A (determined) user's perspective

When one runs the `verifier` on a Mizar file, say `tarski.miz`, one will see something like this:

```
Verifier based on More Strict Mizar Processor, Mizar Ver. 8.1.11 (Linux/FPC)
Copyright (c) 1990-2022 Association of Mizar Users
Processing: mizshare/mml/tarski.miz

Parser    [ 138]   0:00
MSM       [ 132]   0:00
Analyzer  [ 137]   0:00
Checker   [ 137]   0:00
Time of mizaring: 0:00
```

This refers to modules named `Parser`, `MSM`, `Analyzer`, `Checker`, so one can reasonably guess that the code is split up into multiple passes and these are the names of the modules.

- `Parser` reads the text and converts it into an abstract syntax tree.
- `MSM` stands for "More Strict Mizar", which is a slightly more elaborated version of the syntax tree. Notably, this pass does (some kinds of) name resolution, as well as resolving and auto-binding reserved variables.
- `Analyzer` is responsible for processing the large scale logical structure of a Mizar document. It resolves symbol overloading and checks types, as well as tracking the "thesis" as it is transformed by Mizar proof steps.
- `Checker` is called by the analyzer whenever there are atomic proof obligations. It is sometimes referred to as the "**by**" proof automation because it is called whenever there is a step proved using the **by** keyword.

Both the analyzer and the checker are soundness-critical components. Clearly the checker is important because it actually checks proofs – a bug here will mean that an incorrect proof is accepted – but the analyzer is also important because it determines the proof obligations that will be sent to the checker. If the analyzer sends the wrong assertion to the checker or simply doesn't call the checker at all, then it might assert a theorem that hasn't been properly checked.

Another visible effect of running `verifier` on a `.miz` file is that a huge number of other files are generated with bespoke extensions, which reveals another implementation quirk of the Mizar system, which is that the four components above are very loosely coupled and communicate only via the file system. The parser reads the `.miz` file and produces a `.wsx` file, which is read by `MSM` to produce `.msx`, which is read by the `transfer2analyzer` module (not mentioned above) to produce `.par`, which is read by `Analyzer` to produce `.xml`, which is read by `Checker` and verified.

The general data flow of a Mizar verification looks like this:

- The first tool to process the article is the "accommodator," `accom`. This is the only tool which reads data from articles other than the current one; it is responsible for aggregating data from imports and collecting them for the current article.
  - reads: `mml.ini`, `mml.vct`, `mizar.dct`, *art*`.miz`, and *dep*.{`dco`, `dre`, `dcl`, `dno`, `did`, `drd`, `dpr`, `def`, `the`, `sch`}
  - writes: *art*.{`vcl`, `sgl`, `cho`, `aco`, `atr`, `ere`, `ecl`, `eno`, `nol`, `ano`, `frm`, `prf`, `eid`, `erd`, `epr`, `dfs`, `dfe`, `dfx`, `eth`, `esh`, `fil`, `err`, `log`}

  Here *art* is the article being processed and *dep* is an article which is imported by this one. For example `xboole_0.miz` has a **`theorems TARSKI`** declaration in its `environ` section, so the accommodator will load `tarski.the` to get the theorems from article `tarski`

and put them in `xboole_0.eth`. Similar things are done for constructors, notations, definitions, etc., each of which is in a separate file, which is why so many files are read and written in this step.

- The next pass is the parser, which parses the body of the *art*`.miz` article and produces a *art*`.wsx` file (Weakly Strict Mizar) containing the AST, along with *art*`.frx` containing the formats declared in the article.
- The MSM pass reads *art*`.wsx` and writes *art*`.msx`, performing name resolution and filling in the types of `reservation` variables in statements.
- The `transfer2analyzer` pass appears to be for backward compatibility reasons, as it reads *art*`.msx` and translates it to *art*`.par`, which is the same thing but in a less extensible format. It also resolves format references.
- The analyzer pass reads *art*.{`eno`, `epr`, `dfs`, `par`} and writes *art*`.xml`, which is the article AST again but using fully elaborated and typechecked terms, and with all the statements of checker subgoals explicitly annotated.
- The checker is the last pass. It reads *art*.{`ref`, `ere`, `atr`, `ecl`, `dfe`, `dfx`, `epr`, `eid`, `erd`, `eth`, `esh`, `xml`} and verifies the theorems.

The underlined files are the ones which we needed to parse while implementing the analyzer and checker passes.

Thanks to the efforts of J. Urban in 2004 [11], most of these internal files between the components are in XML format. Those that are not XML are highlighted in red. The only non-XML files we need to read are *art*`.ref` and *art*`.ere`, and both of these have rather simplistic number-list formats.

If one opens one of these files, one is presented with the next major challenge, which is that terms are pervasively indexed and hence reading the expressions can be quite difficult. Moreover, there are many *distinct* index classes which are differentiated only by the context in which they appear, so without knowing how the program processes the indices or what array is being accessed it is hard as an outsider to follow the references. In the Rust implementation this issue is addressed by using "newtypes" to wrap each integer to help distinguish different indexing sets. There are currently 36 of these newtypes defined: for example there are numbers for functors, selectors, predicates, attributes, formats, notations, functor symbols (not the same!), left bracket symbols, reserved identifiers, etc.

After parsing these, one ends up with an expression such as $\forall \_ : M_1,\ R_4(K_1(B_1, K_2(N_2)), N_1)$, which means something like "for all x of the first type, fourth relation holds of the first function applied to x and the second function applied to 2, and 1". For many purposes, this is sufficient for debugging, but one tends to go cross-eyed staring at these expressions for too long. Ideally we would be able to reverse this indexification to obtain the much more readable expression $\forall x : \mathbf{Nat}, x \cdot (-2) \leq 1$. However, the code to do this does not exist anywhere in Mizar, because the Mizar checker never prints expressions. The only output of the checker is a list of (`line`, `col`, `err_code`) triples, which are conventionally postprocessed by the `errflag` tool to insert markers like the following in the text:

```
for x being Nat holds x = x
proof
  let x be Nat;
end;
::>,70
::>
::> 70: Something remains to be proved
```

This error message is pointing at the **end** keyword, but notably it does not say *what* remains to be proved, here $x = x$. A debug build of Mizar will actually print out expressions to the *art*`.inf` file, but they are similar to the $R_4(K_1(\dots))$ style.

Luckily, this issue has been addressed outside the main Mizar codebase: J. Urban's HTMLizer is an XSLT stylesheet which can transform the XML intermediate files into fully marked up reconstructed Mizar documents, and which we adapted to design the formatter.

With the present version of the formatter, and with appropriate debugging enabled, an input like the following:

```
for x,y being Nat holds x = 1 & y = 2 implies x + y = 3;
```

yields this debugging trace:

```
input: ∃ b0: natural set, b1: natural set st
  (b0 = 1) ∧ (b1 = 2) ∧ ¬((b0 + b1) = 3)
refuting 0 @ TEST:28:33:
  ∃ b0: natural set, b1: natural set st
    (b0 = 1) ∧
      (b0 c= 1) ∧
      (1 c= b0) ∧
      (b1 = 2) ∧
      (b1 c= 2) ∧
      (2 c= b1) ∧
      ¬((b0 + b1) = 3) ∧
      (((b0 + b1) c= 3) → ¬(3 c= (b0 + b1)))
```

In addition to showing some of the formatting and indentation behavior of the reconstructed expression, this also reveals some aspects of the checker, like how the goal theorem has been negated and the definitional theorem $x = y \leftrightarrow x \subseteq y \wedge y \subseteq x$ has been eagerly applied during preprocessing.

One other feature that is demonstrated here is "negation desugaring", which requires some more explanation. Internally, Mizar represents all expressions using only $\neg$, $\forall$ and n-ary $\wedge$. So $P \to Q$ is mere syntax for $\neg(P \wedge \neg Q)$ and $\exists x, P(x)$ is actually $\neg\forall x, \neg P(x)$. (Even $P \leftrightarrow Q$ is desugared, to $\neg(P \wedge \neg Q) \wedge \neg(Q \wedge \neg P)$, so too much recursive use of $\leftrightarrow$ can cause a blowup in formula size.) This normalization ensures that different spellings of the same formula are not distinguished, for example if the goal is $P \vee Q$ then one may prove it by **assume not** `P;` **thus** `Q;`. Double negations are also cancelled eagerly. In the formatter, we try to recover a natural-looking form for the expression by pushing negations to the leaves of an expression, and also writing $A \wedge B \to C \vee D$ if after pushing negations we get a disjunction such that the first few disjuncts have an explicit negation. (Mizar actually carries some annotations on formulas to help reconstruct whether the user wrote **not** `A` **or** `B` or `A` **implies** `B`, but we chose not to use this information as it is often not available for expressions deep in the checker so we wanted a heuristic that works well in the absence of annotation.)

3 The checker

Mizar is broadly based on first order logic, with the non-logical axioms of Tarski–Grothendieck set theory, with a type system layered on top. While types can depend on terms, so one may call it a dependent type theory, this is not a type system in the sense of Martin-Löf type theory: types are essentially just predicates over terms in an untyped base logic, and the language allows typing assertions to be treated as predicates.

3.1 Core syntax

The core syntax of Mizar uses the following grammar:

$a, b, c, d, e, F, G, H, K, U, V, P, R, S, T_M, T_G ::= \text{ident}$

$t ::= a, b, c, d, e$	variable	$\varphi ::= \top$	true
$\mid n$	numeral	$\mid \neg\varphi$	negation
$\mid \{F, G, K, U\}(\overrightarrow{t_i})$	function application	$\mid \bigwedge \overrightarrow{\varphi_i}$	conjunction
$\mid H(\overrightarrow{t_i}) := t'$	local function app.	$\mid \bigwedge_{b=t}^{t'} \varphi$	flex conjunction
$\mid$ **the** τ	choice	$\mid \forall\, b : \tau.\ \varphi$	for all
$\mid \{t \mid \overrightarrow{b_i : \tau_i} \mid \varphi\}$	Fraenkel	$\mid \{P, R\}(\overrightarrow{t_i})$	predicate
$\tau ::= \overrightarrow{\chi_i}\ T_M(\overrightarrow{t_i})$	mode	$\mid S(\overrightarrow{t_i}) := t'$	local pred.
$\mid \overrightarrow{\chi_i}\ T_G(\overrightarrow{t_i})$	struct type	$\mid t$ **is** χ	attribute
$\chi ::= \pm V(\overrightarrow{t_i})$	attribute	$\mid t : \tau$	qualification

Broadly speaking, terms are first order, meaning applications of function symbols to variables. The exception is the Fraenkel operator $\{t \mid \overrightarrow{b_i : \tau_i} \mid \varphi\}$, which should be read as "the set of $t(\overrightarrow{b_i})$ where the $\overrightarrow{b_i : \tau_i}$ are such that $\varphi(\overrightarrow{b_i})$", where the variables $\overrightarrow{b_i : \tau_i}$ are quantified in both t and φ.

All terms t have a type τ, and there is a robust subtyping system – most terms have *many* types simultaneously. All types are required to be nonempty, which is what justifies the **the** τ constructor for indefinite description. Types are composed of a collection of attributes (a.k.a. clusters) χ applied to a base type $\{T_M, T_G\}(\overrightarrow{b_i})$. Regular types are called "modes"; new modes can be defined by carving out a subset of an existing mode, and modes need not define a set (in particular, `object` is a primitive mode which is the supertype of everything, and does not constitute a set per Russell's paradox). Structure types are roughly modeled after partial functions from some set of "tags", but they are introduced axiomatically, similarly to how structure types are treated in a dependent type theory such as Coq or Lean.

▶ Remark. Although one can define a mode for any ZFC set or class, modes and sets are not interchangeable in Mizar because they lie in different syntactic classes. Modes are types, which go in the type argument of quantifiers such as the **Nat** in $\forall x : \mathbf{Nat}.\ x \geq 0$, while sets are objects, which can be passed to functions, like $|\texttt{NAT}| = \aleph_0$. In the MML, `NAT` is the set of natural numbers, while `Nat` and `Element of NAT` are modes which describe the type of natural numbers (and in general `Element of A` can be used to treat a set as a type).

Attributes, also known as "adjectives" or "clusters", are modifiers on types, which may be described as intersection typing in modern terminology, although attributes do not stand alone as types. For example, "`x is non empty finite set`" means $\neg(x$ **is** `empty`$) \wedge (x$ **is** `finite`$) \wedge (x : \texttt{set})$. The Mizar system treats the collection of attributes on a type as an unordered list for equality comparisons.

Formulas are composed mainly from negation, conjunction and for-all, but there are some extra formula constructors that deserve attention:

- Not represented in the grammar is that $\neg\neg\varphi$ is identified with φ, and internally there is an invariant that $\neg\neg$ never appears.
- Similarly, conjunctions are always flattened, so $P \wedge (Q \wedge R) \wedge S$ becomes $P \wedge Q \wedge R \wedge S$.
- Unlike most type theories, typing assertions are reified into an actual formula, so it is possible to say $4/2 : \mathbb{N}$ and $\neg(-1 : \mathbb{N})$.
- There is also a predicate for having an attribute, and $(t : \chi\ \tau) \leftrightarrow (t$ **is** $\chi) \wedge (t : \tau)$ is provable.

- Flex-conjunction is written with syntax such as `P[1] & ... & P[n]` (with literal "`...`"), and Mizar knows that this expression is equivalent both to $\forall x : \mathbb{N}.\ 1 \le x \le n \to P(x)$ as well as to an explicit conjunction (when n is a numeral). For example `P[1] & ... & P[3]` will be expanded to `P[1] & P[2] & P[3]` in the checker.

The different letters for functions, predicates, and variables correspond to the different roles that these can play, although in most situations they are treated similarly.

- "Locus variables" (a) are used in function declarations to represent the parameters of a function, mode, cluster, etc. For example a local function might be declared as $K(\overrightarrow{a_i}) : \tau := t$ where τ and t are allowed to depend on the $\overrightarrow{a_i}$, and then when typing it we would have that $K(\overrightarrow{t_i})$ has type $\tau[\overrightarrow{a_i \mapsto t_i}]$.
- "Bound variables" (b) are de Bruijn *levels*, used to represent variables in all binding syntaxes. So for example, $\forall x : \mathbb{N}. \forall y : \mathbb{N}. x \le x + y$ would be represented as $\forall \_ : \mathbb{N}. \forall \_ : \mathbb{N}. b_0 \le b_0 + b_1$. NB: there are two conventions for locally nameless variables, called "de Bruijn indices" (variables are numbered from the inside out) and "de Bruijn levels" (variables are numbered from the outside in), and Mizar's choice of convention is the less common one.
- "Constants" (c) are variables that have been introduced by a **`consider`**, **`given`**, or **`take`** declaration, as well as Skolem constants introduced in the checker when there are existentially quantified assumptions. Constants may or may not be defined to equal some term (for example **`take x = t`** will introduce a variable x equal to t) and the checker will use this in the equalizer if available.
- "Inference constants" (d) are essentially checker-discovered abbreviations for terms. This is used to allow for subterm sharing, as terms are otherwise completely unshared.
- "Equivalence classes" (e) are used in the equalizer to represent equivalence classes of terms up to provable equality. So for example if the equalizer sees a term $x + y$ it will introduce an equivalence class e_1 for it and keep track of all the things known to be in this class, say $e_1 = x + y = y + x = e_1 + 0$.
- Functors (K) and predicates (R) are the simplest kind of definition, they correspond to function and relation symbols in traditional FOL.
- Scheme functors (F) and scheme predicates (P) are the higher-order analogue of constants. They are only valid inside scheme definitions, which declare these at the start and then use them in the statement and proof.
- Local functors (H) and local predicates (S) are function declarations within a local scope, declared with the **`deffunc`** or **`defpred`** keywords. Internally every predicate carries the result of substituting its arguments, so if we declare **deffunc** $H_1(x, y) := x + y$ then an expression like $2 \cdot H_1(x - 1, y)$ is really stored as $2 \cdot (H_1(x - 1, y) := x - 1 + y)$. Some parts of the system treat such an expression like a function application, while others treat it like an abbreviation for a term.
- An aggregate functor (G) is Mizar terminology for a structure constructor $\{\text{foo} := x, \text{bar} := y\} : \text{MyStruct}$, and the converse operator is a selector (U), which is the projection $t.\text{foo}$ for a field. (The Mizar spelling for these operators is `MyStruct(# x, y #)` for the constructor and **`the foo of t`** for the projection.)

3.2 Structure of the checker

The checker is called whenever there is a proof such as "`2 + 2 = 4;`" or "`x <= y by A1,Thm2;`": basically any time there is a **`by`** in the proof text, as well as when propositions just end in a semicolon, this being the nullary case of **`by`**. It consists of three

major components, although there is another piece that is used even before the checker is properly called on a theorem statement:

0. **Attribute inference** ("rounding-up") is used on every type $\overrightarrow{\chi}^{-}\tau$ to prove that it is equivalent to $\overrightarrow{\chi}^{+}\tau$ where $\overrightarrow{\chi}^{+}$ is a superset of $\overrightarrow{\chi}^{-}$. Internally we actually keep *both* versions of the attributes (the "lower cluster" $\overrightarrow{\chi}^{-}$ being the one provided by the user and the "upper cluster" $\overrightarrow{\chi}^{+}$ being what we can infer by applying all inference rules), because it is useful to be able to prove that two clusters are equal if $\overrightarrow{\chi}^{-}_1 \subseteq \overrightarrow{\chi}^{+}_2$ and $\overrightarrow{\chi}^{-}_2 \subseteq \overrightarrow{\chi}^{+}_1$.
1. The **pre-checker** takes the list of assumptions, together with the negated conjecture, and performs a number of normalizations on them, skolemizing existentials, removing vacuous quantifiers, and expanding some definitions. Then it converts the whole formula into *disjunctive* normal form (DNF) and tries to refute each clause.
2. The **equalizer** does most of the "theory reasoning". It replaces each term in the clause with an equivalence class, adding equalities for inference constants and defined constants, as well as registered equalities and `symmetry` declarations, as well as any equalities in the provided clause. It also evaluates the numeric value (for $2+2=4$ proofs) and polynomial value (for $(x+1)^2 = x^2+2x+1$ proofs) of equality classes and uses them to union classes together. These classes also have many *types* since many terms are going into the classes, and all of the attributes of these types are mixed together into a "supercluster" and rounded-up some more, potentially leading to a contradiction. For example if we know that $x = y$ and x is positive and y is negative then the supercluster for $\{x, y\}$ becomes `positive negative` which is inconsistent.
3. The last step is the **unifier**, which handles instantiation of quantifiers. This is relatively simplistic and non-recursive: for each assumption $\forall \vec{b}.\ P(\vec{b})$ it will instantiate $P(\vec{v})$ (where $\vec{v}$ are metavariables) and then construct the possible assignments (as a DNF) to the variables that would make $P(\vec{v})$ inconsistent with some other assumption. It tries this for each forall individually, and if it fails, it tries taking forall assumptions in pairs and unifying them. If that still fails then it gives up – it does not attempt a complete proof method.

Finally, there is one more component which is largely separate from the "`by`" automation:

4. The **schematizer** is the automation that is called for justifications starting with "`from`". These are scheme instantiations. In this case it is very explicitly given the list of hypotheses in the right order, so the only thing it needs to do is to determine an assignment of the scheme variables (F, P) to regular functors and predicates (K, R) or local functors and predicates (H, S). Users are often required to introduce local predicates in order to apply a scheme. Nullary scheme functors (a.k.a constant symbols) can be unified with arbitrary terms, however.

Although we cannot go into full detail on the algorithms here, in the following sections we will go into some of the highlights, with an emphasis on what it takes to audit the code for logical soundness, through the lens of root-causing some soundness bugs.

3.3 Requirements

One aspect of the Mizar system that is of particular interest is the concept of "requirements", which are definitions that the checker has direct knowledge of. For example, the grammar given above does not make any special reference to equality: it is simply one of the possible relation symbols $R_i(x, y)$, and the relation number for equality can vary from one article to the next depending on how the accommodator decides to order the imported relation symbols. Nevertheless, the checker clearly needs to reason about equality to construct equality equivalence classes.

To resolve this, there is a fixed list of "built-in" notions, and one of the files produced by the accommodator (*art*`.ere`) specifies what the relation/functor/mode/etc. number of each requirement is. Importantly, if a constructor is identified in this way as a requirement, this not only allows the checker to recognize and produce expressions like $x = y$, it is *also an assertion that this relation behaves as expected*. If a requirement is given a weird definition, for example if we were to open `xcmplx_0.miz` and change the definition of $x + y$ to mean subtraction instead, we would be able to prove false in a downstream theorem which enables the requirement for $+$, because we would still be able to prove $2 + 2 = 4$ by evaluation.

At the surface syntax level, requirements are enabled in groups, using the **`requirements`** directive in the import section. The requirements are, in rough dependency order:

- `HIDDEN` (introduced after the `HIDDEN` article) is a requirement that is automatically enabled for every Mizar file. It introduces the modes `object` and **`set`**, as well as $x = y$ and $x \in y$. (The article `HIDDEN` itself is somewhat magical, and cannot be processed normally because every file takes an implicit dependency on `HIDDEN`.)
- `BOOLE` (introduced after `XBOOLE_0`) introduces the adjective **`x is empty`**, as well as set operators $\emptyset$, $A \cup B$, $A \cap B$, $A \setminus B$, $A \oplus B$, and A meets B (i.e. $A \cap B \neq \emptyset$). These operators have a few extra properties such as $A \cup \emptyset = A$ that are used in the equalizer.
- `SUBSET` (introduced after `SUBSET_1`) introduces the mode **`Element of A`**, along with $\mathcal{P}(A)$, $A \subseteq B$, and the mode **`Subset of A`**. The checker knows about how these notions relate to each other, for example if $x \in A$ then x : **`Element`**(A). (Because types have to be nonempty, x : **`Element`**(A) is actually equivalent to $x \in A \vee (A = \emptyset \wedge x = \emptyset)$. So the reverse implication used by the checker is $(x$: **`Element`**$(A)) \wedge \neg(A$ **`is empty`**$) \to x \in A$.)
- `NUMERALS` (introduced after `ORDINAL1`) introduces succ(x), **`x is natural`**, the set $\mathbb{N}$ (spelled `NAT` or `omega`), 0, and **`x is zero`**. Note that 0 is not considered a numeral in the sense of section 3.1, it is a functor symbol $K_i()$. Numbers other than 0 can be written even before the `ORDINAL1` article, but they are uninterpreted sets; after this requirement is added the system will give numerals like 37 the type **`Element of NAT`** instead of **`set`**.
- `REAL` (introduced after `XXREAL_0`) introduces $x \leq y$, **`x is positive`**, **`x is negative`**, along with some basic implications of these notions.
  - `NUMERALS + REAL` enables the use of flex-conjunctions $\bigwedge_{i=a}^{b} \varphi(i)$, since these expand to the expression $\forall i : \mathbb{N}.\ a \leq i \leq b \to \varphi(i)$ which requires $\mathbb{N}$ and $\leq$ to write down.
- `ARITHM` (introduced after `XCMPLX_0`) introduces algebraic operators on the complexes: $x + y$, $x \cdot y$, $-x$, x^{-1}, $x - y$, x/y, i, and **`x is complex`**. This also enables the ability to do complex rational arithmetic on numerals, as well as polynomial normalization.

3.3.1 Soundness considerations of the requirements

There is a slight mismatch between what the user has to provide in order to enable a requirement and what the checker gets to assume when a requirement is enabled which causes a challenge for proof export or other external soundness verification. As the list above might indicate, generally checker modules corresponding to a requirement are enabled as soon as all of the *constructors* involved in stating them are available; for example we can see this with the `NUMERALS + REAL` prerequisite for flex-conjunctions. (More precisely, flex-conjunctions are enabled exactly when $\mathbb{N}$ and $\leq$ become available.) However, the checker needs more than that to justify the manipulations it does with them.

For example the checker exploits the fact that $\bigwedge_{i=1}^{3} \varphi(i)$ is equivalent both to $\forall i : \mathbb{N}.\ 1 \leq i \leq 3 \to \varphi(i)$ and to $\varphi(1) \wedge \varphi(2) \wedge \varphi(3)$, and so this amounts to an assertion that $a \leq i \leq b \leftrightarrow i = a \vee i = a + 1 \vee \cdots \vee i = b$ is provable when a and b are numerals such that $a \leq b$. The reverse implication follows from numerical evaluation, and the forward implication is a metatheorem that can be proven by induction, assuming the existence of lemma L saying $a \leq i \to i = a \vee \text{succ}(a) \leq i$:

▶ **Theorem 1.** *If a and n are numerals and $i : \mathbb{N}$, then*

$$i : \mathbb{N} \vdash a \le i \le \mathrm{succ}^n(a) \to i = a \vee i = \mathrm{succ}(a) \vee \cdots \vee i = \mathrm{succ}^n(a)$$

is provable.

Proof. By induction on n. Applying the induction hypothesis with $\mathrm{succ}(a)$ and $n-1$, we get

$$i : \mathbb{N} \vdash \mathrm{succ}(a) \le i \le \mathrm{succ}^n(a) \to i = \mathrm{succ}(a) \vee \cdots \vee i = \mathrm{succ}^n(a)$$

so it suffices to show $a \le i \to i = a \vee \mathrm{succ}(a) \le i$, and we appeal to lemma L. ◀

So ideally, when introducing $\le$ or the `REAL` requirement, one would be required to supply a proof of lemma L somehow to justify that the checker will be making use of this fact in the following article. Unfortunately, there is no actual place to inject this theorem into the system, because there is no concrete syntax for introducing requirements. That is, even taking all the `.miz` files in the MML together there is nothing that would indicate that `XXREAL_0` is the article which allows the `REAL` requirement to be used.[3]

The way this actually works is that when the accommodator sees a **`requirements REAL;`** directive, it reads the (hand-written) `real.dre` XML file, which explicitly names the constructor number for $\le$ and the fact that it is in article `XXREAL_0`. We would like to propose that this file also contains *justifications* for involved constants so that the theorems aren't smuggled in without proof. (Or even better, requirement declarations become a part of the language proper, so that they can get **`correctness`** proof blocks like any other justified property.)

The case of `NUMERALS + REAL` enabling flex-conjunctions is especially interesting because neither of these requirements depends on the other, so neither `numerals.dre` nor `real.dre` can state the compatibility theorem $a \le i \to i = a \vee \mathrm{succ}(a) \le i$ between them. (This doesn't require a major reorganization to fix, since of course the article that introduces `REAL`, `xxreal_0.miz`, references the article which introduces `NUMERALS`, `ordinal1.miz`.)

4 The analyzer

The analyzer plays an interesting role in Mizar. This is an essentially completely separate inference system from the checker, which has much stricter rules about equality of expressions, and it is what forms the "glue" between different lines of proof. It is a large module only because the language of Mizar is quite expansive, with 109 keywords, including:

- Different kinds of definitions for modes, functors, predicates; redefinitions;
- Definitions by case analysis;
- "Notations" (**`synonym`** and **`antonym`** declarations);
- "Properties" like **`commutativity`**, **`projectivity`** or **`involutiveness`**;
- Cluster registrations (existential, functor, and conditional);
  - Existential clusters assert that $\overrightarrow{\chi}\ \tau$ is nonempty and hence a legal type
  - Functor clusters assert that t **is** $\overrightarrow{\chi}$ for some term t
  - Conditional clusters assert that $\overrightarrow{\chi}$ implies $\overrightarrow{\chi}'$

[3] Note that is also a Mizar article called `real.miz`, which contains some of the lemmas that are auto-proved by the `REAL` requirement, but there is no formal relation between the article and the requirement, and it is only an incomplete approximation to the lemmas required to justify the requirement, not formally checked. This should be easy to fix, and will more or less fall out of any attempt at proof export.

- "Reductions", equalities the checker automatically uses for simplification;
- "Identifications", equalities the checker automatically uses for congruence closure;
- Local declarations;
- Schemes;
- Reservations (variables with types declared in advance);
- Propositions and theorems;

and this is not an exhaustive list.

This also only covers top-level items. Inside a **`proof`** block there is a different (overlapping but largely disjoint) set of legal items, called skeleton steps, which correspond to natural deduction rules. Inside a proof there is a variable that holds the current "thesis", the goal to prove, and the **`thesis`** keyword resolves to it, unless one is in a **`now`** block, where the thesis is not available and is reconstructed from the skeleton steps.

- **`let x be T;`** is the forall introduction rule: it transforms the thesis from $\forall x : T.\ \varphi(x)$ to $\varphi(x)$ and introduces a constant $x : T$.
- **`assume A;`** is the implication introduction rule: it transforms the thesis from $A \to B$ to B and pushes a proposition A, which can be referred to using **`then`**.
- **`thus A;`** is the conjunction introduction rule: it transforms the thesis from $A \wedge B$ to B, and gives A as a goal to the checker.
- **`take t;`** is the existential introduction rule: it transforms the thesis from $\exists x.\ \varphi(x)$ to $\varphi(t)$. There is also **`take x = t;`** which is the same but introduces x as an abbreviation for t; this version can also be used in a **`now`** block.
- **`consider x being T such that A;`** is existential elimination: it introduces $x : T$ and a proposition $A(x)$ that can be labeled, and gives $\exists x : T.\ A(x)$ as a goal to the checker.
- **`given x being T such that A;`** is a combination of implication introduction and existential elimination: it transforms $(\exists x : T.\ A(x)) \to B$ to B and introduces $x : T$ and a proposition $A(x)$ that can be labeled.
- **`reconsider x = t as T;`** introduces $x : T$ as a new local constant known to be equal to t, and gives $(t : T)$ as a proof obligation to the checker. (This is mainly used when $t : T$ is not already obvious to the type system, and allows $t : T$ to be proved by the user.)
- **`per cases;`** is disjunction elimination, and it has two variations:
  - followed by a sequence of **`suppose A_i; ... end;`** blocks, it gives $\bigvee_i A_i$ as a goal to the checker and makes A_i available in each block, leaving the thesis unchanged;
  - followed by a sequence of **`case A_i; ... end;`** blocks, it gives $\bigvee_i A_i$ as a goal to the checker, and if the thesis is $\bigvee_i (A_i \wedge B_i)$ then B_i becomes the thesis in each block.

Additionally, the formulas don't have to exactly match what the skeleton steps say. For example one can start a proof of $\forall x : \mathbb{N}.\ \varphi(x)$ using **`let x be set;`** because **`set`** is a supertype of `Nat`. Definitional unfolding can also be forced by a skeleton step, for example if the thesis is $A \subseteq B$ then **`let x be set;`** is a legal step provided the right **`definitions`** directive is supplied.

The other major role of the analyzer is to elaborate types, terms, and formulas from their input form to the core grammar shown in section 3.1. There are two things that make this challenging:

- Mizar heavily uses overloading, where the same function symbol can have several definitions (and `redefinition`s, which are definitions which use the same base term but can have different input and output types). These are resolved by declaration order (last declaration wins) and typing. Types are propagated exclusively from the inside out, using this type-based overload resolution, although you can use `e qua A` as a type ascription to influence the selection.

- Many declarations have "invisible arguments", also known as "implicit arguments" in the literature. These are filled in by a straightforward first-order unification process.[4]

5 Mizar soundness bugs

One of the fortuitous side effects of going over each line of code and rewriting it to something morally equivalent in a different language is that one can find a lot of bugs. Bugs can happen even when one takes great efforts to avoid them [1, 8], but external review can definitely help. Mizar is a large project with a long history and a small team, whose source code was not publicly accessible, with many soundness-critical parts, which is pretty much a worst case scenario for finding soundness bugs. This `mizar-rs` project has been tremendously successful in ferreting out these bugs, with no less than four proofs of false that will be given below. These errors have been reported to the Mizar developers, and a patched version is available[5].

While it is unfortunate that the software wasn't perfect to start with, this is evidence for the usefulness of external checkers, and it is a way for us to improve the original Mizar. We hope that by telling the story of how these bugs work we can give some sense of some of the issues that can arise when doing proof checking, as well as some more internal details whose importance may not have been obvious.

5.1 Exhibit 1: polynomial arithmetic overflow

This is the largest of the contradiction proofs, and we will show only the main part of it.[6] The only non-MML notion used is the adjective `a is x-ordered` defined as $x < a$.

```
theorem contradiction
proof
 consider x being 1-ordered Nat such that not contradiction;
 1 is 0-ordered; then
 A1: x * x is 1-ordered; then
 consider x1 being 0-ordered Nat such that B1: x1 = x * x;
 A2: 1 < x1 by A1,B1; then
 x1 * x1 is x1-ordered by XREAL_1:155; then :: 0 < a ∧ 1 < b → a < a · b
 consider x2 being x1-ordered Nat such that B2: x2 = x1 * x1;
 ...
 consider x31 being x1-ordered Nat such that B31: x31 = x30 * x30;
 consider x32 being x1-ordered Nat such that B32: x32 = x31 * x31;
 C: x32 * x1 = x1 by
    B1,B2,B3,B4,B5,B6,B7,B8,B9,B10,B11,B12,B13,B14,B15,B16,
    B17,B18,B19,B20,B21,B22,B23,B24,B25,B26,B27,B28,B29,B30,B31,B32;
 x32 is x1-ordered implies x1 < x32;
 then 0 < x1 & 1 < x32 by A2,XXREAL_0:2; :: a ≤ b ∧ b ≤ c → a ≤ c
 hence contradiction by C,XREAL_1:155;    :: 0 < a ∧ 1 < b → a < a · b
end;
```

[4] Unfortunately, the unification process does not have completely unique solutions, because attributes are unordered, and this can lead to overfitting in the MML to the Mizar attribute ordering. For example, `rng`$(f) \subseteq B$ returns the range of a function $f : A \to B$ as an element of `Subset of B` (and `rng` has hidden arguments $\texttt{rng}_{A,B}(f)$ inferred from the types). But the function type is split into separate attributes as `A-defined B-valued Function`, so if `f is B-valued C-valued Function` then it is up to a variety of implementation details whether you get the type of `rng f` as `Subset of B` or `Subset of C`. This happens in practice for the empty function, which is registered as `NAT-valued`, `RAT-valued`, and a few others.

[5] `https://github.com/digama0/mizar-system`

[6] The full proof can be found at `https://github.com/digama0/mizar-rs/blob/itp2023/itp2023/false1/false1.miz`.

As mentioned earlier, there is a module in the equalizer for polynomial evaluation. This means that each expression which is a complex number will also be expressed as a polynomial in terms of basic variables, and two expressions which compute to equal polynomials will be equated. This is how things like $-(a+b) = -a + -b$ are proved automatically by the checker.

These polynomials are of the form $\sum_i c_i \prod_j x_{i,j}^{n_{i,j}}$, and the starting point for this proof was the discovery that the $n_{i,j}$ are represented as signed 32-bit integers and overflow is not checked. Turning this into an exploit is surprisingly difficult however, because one cannot simply write down $x^{2^{32}}$ because the power function is not one of the requirements (see section 3.3). The best thing we have for creating larger polynomials is multiplication, but we can get to $x^{2^{32}}$ with 32 steps of repeated squaring, which is what the large block of `consider` statements is doing.

So the strategy is to construct $x_1 = x^2$, $x_2 = x^4$, all the way up to $x_{30} = x^{2^{30}}$. After this things start to get weird: $x_{31} = x^{-2^{31}}$ because of signed overflow, and $x_{32} = x^0$. The system does not recognize $x_{32} = 1$ however, because it maintains an invariant of monomial powers being nonzero, and single powers are also handled specially, so we have to go up to $x_{32} \cdot x_1 = x^2$. Since x_1 is also x^2, the system will equate $x_{32} \cdot x_1 = x_1$ which is the key step **C**. After this, we simply need to separately prove that since $x_{32} \cdot x_1$ is really $x^{2^{32}+2}$, it is strictly larger than $x_1 = x^2$ as long as we choose $x > 1$.

To prove this last fact we use the attribute inference mechanism to prove that all of the intermediates are strictly greater than x_1, since $x_1 < a$ implies $x_1 < a \cdot a$ as long as $x_1 > 1$. Hence $1 < x_1 < x_{32}$ so $x_1 < x_{32} \cdot x_1 = x_1$, which is a contradiction.

5.2 Exhibit 2: negation in the schematizer

This one requires absolutely no imports; we include the complete proof below including the import section.

```
environ begin

scheme Foo{P[set,set]}: P[1,1] implies P[1,1]
proof thus thesis; end;

theorem contradiction
proof
  1 = 1 implies 1 <> 1 from Foo;
  hence thesis;
end;
```

This defines a very trivial scheme which just says that $P(1,1)$ implies $P(1,1)$. The interesting part is the instantiation of this scheme in the main proof. In the schematizer, we do not negate the thesis and try to prove false, we just directly match the thesis against the goal. It does not attempt any fancy higher-order unification: if it needs to unify $P(\overrightarrow{t_i}) \stackrel{?}{=} \pm R(\overrightarrow{t_i}')$ it will just assign $P := \pm R$ and proceed with $t_i \stackrel{?}{=} t_i'$ for each i. Or at least it should do that, but there is a bug wherein it instead assigns $P := R$ and ignores the $\pm$ part, so we can trick it into unifying $P(x,y) := (x = y)$ and then think that $P(x,y) \stackrel{?}{=} (x \neq y)$ is still true. So we use $1 = 1$ to prove $1 \neq 1$ and then prove a contradiction.

5.3 Exhibit 3: flex-and unfolding

```
theorem contradiction
proof
  (1 = 1 or 1 = 1) & ... & (2 = 1 or 1 = 1);
  hence thesis;
end;
```

This one is bewilderingly short. We start by asserting a (true) flex-and statement $\bigwedge_{i=1}^{2}(i = 1 \vee 1 = 1)$. We prove this by using the forall expansion of the flex-and, $\forall i : \mathbb{N}.\ 1 \leq i \leq 2 \to i = 1 \vee 1 = 1$, which is of course true because the $1 = 1$ disjunct is provable.

The second part is simply `hence contradiction`, so we are calling the checker to *disprove* the same statement. So we assume $\bigwedge_{i=1}^{2}(i = 1 \vee 1 = 1)$ and one of the things the pre-checker does here is to expand out the conjunction, and we would expect it to produce $(1 = 1 \vee 1 = 1) \wedge (2 = 1 \vee 1 = 1)$ from which no contradiction can be found. However, what it actually produces is $1 = 1 \wedge 2 = 1$, which can be disproved.

To see why this happens, we have to look more specifically at the forall-expansion without the negation sugar employed thus far. What Mizar actually sees for the expansion of the flex-and expression is $\forall i : \mathbb{N}.\ \neg(1 \leq i \wedge i \leq 2 \wedge (i \neq 1 \wedge 1 \neq 1))$, except that as mentioned previously conjunctions are always flattened into their parents. The code for expanding flex-and expects the expansion body to be the third conjunct past the forall and negation, but after flattening the third conjunct is actually $i \neq 1$ and there is an unexpected fourth conjunct $1 \neq 1$ that is forgotten.

5.4 Exhibit 4: flex-and substitution

This is the most worrisome of the bugs that have been presented, because it is not simply a bad line of code but rather an issue with the algorithm itself. As a result, this one survived the translation to Rust and was discovered to affect both versions, and moreover it has still not been fixed in `mizar-rs`, because the fix could not be rolled out without breaking the MML. (There is an "unsound flag" which controls whether to do the thing that is unsound or the thing that is sound but fails to validate the MML.)

```
theorem contradiction
proof
  A: now
    let n be Nat;
    assume n > 3 & (1 + 1 <> 3 & ... & n <> 3);
    hence contradiction;
  end;
  ex n being Nat st n > 3 & (1 + 1 <> 3 & ... & n <> 3)
  proof
    take 2 + 2;
    thus 2 + 2 > 3 & (1 + 1 <> 3 & ... & 2 + 2 <> 3);
  end;
  hence thesis by A;
end;
```

To explain what is happening here, we have to talk about another aspect of flex-and expressions which was not mentioned in section 3.1, which is that a flex-and expression $\bigwedge_{i=a}^{b} \varphi(i)$ is actually stored as five pieces of information: the bounds a and b, the expansion

$\forall i : \mathbb{N}.\ a \le i \le b \to \varphi(i)$, from which $\varphi(i)$ can be reconstructed (if one is careful – see section 5.3), and the bounding expressions $\varphi(a)$ and $\varphi(b)$. As the reader may have noticed, the syntax of Mizar very much prefers to only discuss the bounding expressions, since flex-and expressions in the concrete syntax are written as $\varphi(a) \wedge \cdots \wedge \varphi(b)$ with literal dots, and no place to supply a, b or $\varphi(i)$.

When one writes an expression like $P \wedge \cdots \wedge Q$, Mizar essentially diffs the two expressions to determine what is changing and what the values are on each side. So if one writes $1 + 1 \neq 3 \wedge \cdots \wedge n \neq 3$ then Mizar infers that the desired expression is $\bigwedge_{i=1+1}^{n} i \neq 3$.

The problem is that this operation is not stable under substitution. If we take that expression and substitute $n := 2 + 2$, then we end up with $\bigwedge_{i=1+1}^{2+2} i \neq 3$, but if we were to write out $1 + 1 \neq 3 \wedge \cdots \wedge 2 + 2 \neq 3$ ourselves we would end up with $\bigwedge_{i=1}^{2} i + i \neq 3$ instead. This is somewhat annoying, especially if one is trying to write an exploit example, but it's not obviously a soundness bug yet because internally we are still storing the expansion which has all the details – $\bigwedge_{i=1+1}^{2+2} i \neq 3$ and $\bigwedge_{i=1}^{2} i + i \neq 3$ are different expressions.

However, the equality check between two flex-and expressions is simply $\varphi(a) = \varphi'(a)$ and $\varphi(b) = \varphi'(b)$! The example above is crafted to demonstrate that this is not sound in general, since $\bigwedge_{i=1+1}^{2+2} i \neq 3$ is false but $\bigwedge_{i=1}^{2} i + i \neq 3$ is true.

In the first part of the proof we show that $\forall n.\ \neg(n > 3 \wedge \bigwedge_{i=1+1}^{n} i \neq 3)$, which is true. The interesting part is the second half, where we take $n := 2 + 2$. At this point the thesis is $2 + 2 > 3 \wedge \bigwedge_{i=1+1}^{2+2} i \neq 3$, but we cannot write this expression. What we write instead is $2 + 2 > 3 \wedge (1 + 1 \neq 3 \wedge \cdots \wedge 2 + 2 \neq 3)$, which as mentioned will be interpreted as $2 + 2 > 3 \wedge \bigwedge_{i=1}^{2} i + i \neq 3$. This should be rejected for not matching the thesis, but it has the same endpoints so it is accepted, and the checker is able to prove it by case analysis. If we were to write **`thus thesis`** to let Mizar insert the unmentionable proposition, then the analyzer will accept it as being the right thesis but the checker will not be able to prove it.

The reason this is a hard bug to squash is because the MML actually relies on it; there are several proofs in `aofa_l00.miz` that break when strict flex-and checking is enabled. All of the examples found seem to be variations on $(\bigwedge_{i=a+c}^{b+c} \varphi(i)) \leftrightarrow (\bigwedge_{i=a}^{b} \varphi(i + c))$, which is true, so there is some hope that a more subtle check can be used to fix the soundness issue without making the prover much worse.

5.5 Honorable mention: attributes that don't exist

This is not an exploitable bug to my knowledge, but it is notable for being widespread, and it is difficult for `mizar-rs` to support without resulting in very weird behavior. Consider the following Mizar article:

```
environ
  vocabularies ZFMISC_1, SUBSET_1;
  notations ZFMISC_1, SUBSET_1;
  constructors TARSKI, SUBSET_1;
  requirements SUBSET; :: , BOOLE;
begin
for x,B being set , A being Element of bool B st x in A holds x in B;
```

This checks as one would expect, and it is in fact a true statement (and it is not at all contrived). However, the reasoning that gets the checker to accept the proof is... somewhat suspicious. It goes as follows:

1. Suppose $A : \text{Element}(\mathcal{P}(B))$, $x \in A$ and $x \notin B$.
2. Because $A : \text{Element}(\mathcal{P}(B))$ and $x \in A$, it follows that B is not empty and $x : \text{Element}(B)$.
3. Because $x \notin B$, B is not empty, and $x : \text{Element}(B)$, contradiction.

Table 1 Comparison of the original (Pascal) implementation of Mizar with `mizar-rs` (Rust). Tests were performed on an 8 core 11th Gen Intel Core i7-1165G7 @ 2.80GHz, with 8 threads.

	PC Mizar	`mizar-rs`	ratio
lines, checker	30 458	14 474	2.10
lines, checker + analyzer	56 926	20 096	2.83
compiling MML, checker only, real time	57.37 min	11.33 min	5.06
compiling MML, checker only, CPU time	417.57 min	73.70 min	5.67
compiling MML, checker + analyzer, real time	71.38 min	11.71 min	6.10
compiling MML, checker + analyzer, CPU time	490.55 min	81.93 min	5.99

Because the example has been minimized, there is really not much going on in the proof because these are all essentially primitive inferences. The problem here is "B is not empty", because the `BOOLE` requirement which supplies the `B is not empty` predicate is not available (note the environment section). The constructor for empty is actually available in the environment because it has been brought in indirectly via the `constructors SUBSET_1` directive, but without the requirement the checker just sees it as a normal attribute.

So what, then, is the checker doing? This seems to be a case of multiple bugs cancelling each others' effects. Requirements are internally represented by an integer index, where zero means that the requirement is not available. Normally any handling of a requirement involves a check that the requirement is nonzero first, but we forget that in step 2, and as a result we end up adding attribute 0 to B, which is meaningless. The second bug is in step 3, where we again forget to ask whether the requirement index is nonzero, and so we find attribute 0 and interpret it (correctly) as meaning that B is not empty.

This would just be a curious bug, but for the fact that it is exploited all over the place because when creating articles it is standard to minimize the environment section by removing anything that keeps the proof valid, and this bug allows one to remove the `BOOLE` requirement without breaking the proof. We had to patch 46 articles that all had this same issue. In all cases we only need to add the `BOOLE` requirement to fix the issue.

6 Results

The whole project is 20 096 lines of code (see table 2), or 14 474 if we restrict attention to those files used in the checker, which is 1/2 to 1/3 of the equivalent code in Pascal (see table 1). We credit this mainly to the language itself – Rust is able to express many complex patterns that would be significantly more verbose to write in Pascal, and PC Mizar is also written in a fairly OOP-heavy style that necessitates a lot of boilerplate. Furthermore, Pascal is manually memory-managed while Rust uses "smart-pointer" style automatic memory management, so all the destructors simply don't need to be written which decreases verbosity and eliminates many memory bugs.

The performance improvements are most striking: we measured a 5-6× reduction in processing time to check the MML. This is most likely a combination of characteristics of the LLVM compiler pipeline, along with many small algorithmic improvements and removing redundant work. (The reduction is even larger when adding the other phases of the Mizar system – accom, parser, MSM – so that we can skip the costly I/O and serialization steps of section 2.)

Table 2 Line counts for the files in `mizar-rs`. Files in red are only required for the analyzer, not the checker.

file	lines
`analyze.rs`	3471
`main.rs`	2747
`equate.rs`	2617
`types.rs`	2114
`parser/msm.rs`	1489
`unify.rs`	1346
`parser/mod.rs`	1280
`checker.rs`	1083
`reader.rs`	890
`parser/article.rs`	759
`ast.rs`	662
`equate/polynomial.rs`	586
`format.rs`	581
`bignum.rs`	372
`util.rs`	99
total	20096

7 Conclusion & Future work

We have implemented a drop-in replacement for the `verifier -c` checker of the Mizar system that is able to check the entire MML, which gets a significant performance improvement. Furthermore, we have improved Mizar by uncovering and reporting some soundness bugs.

While this implementation is explicitly trying not to diverge from the Mizar language as defined by the PC Mizar implementation and the MML, there are many possible areas where improvements are possible to remove unintentional or undesirable restrictions. For example, anyone who has played with Mizar will have undoubtedly noticed that all the article names are 8 letters or less, for reasons baked deeply into PC Mizar. Should `mizar-rs` become the official Mizar implementation, it would be possible to lift this restriction without much difficulty.

This project was also originally started to get proof export from Mizar, and to that end replacing one large trusted tool with another one does not seem like much of an improvement. But (bugs notwithstanding) we saw nothing while auditing the checker that is not proof-checkable or unjustified, and remain confident that proof export is possible.

References

1 Mark Miles Adams. Proof auditing formalised mathematics. *Journal of Formalized Reasoning*, 9(1):3–32, 2016.

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*, 61:9–32, 2018.

3 Grzegorz Bancerek, Czesław Byliński, Adam Grabowski, Artur Korniłowicz, Roman Matuszewski, Adam Naumowicz, Karol Pak, and Josef Urban. Mizar: State-of-the-art and beyond. In *Intelligent Computer Mathematics: International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings.*, pages 261–279. Springer, 2015.

4 Mario Carneiro. The Divergence of the Sum of Prime Reciprocals. *Formalized Mathematics*, 30(3):209–210, 2022. doi:doi:10.2478/forma-2022-0015.

5 Adam Grabowski, Artur Korniłowicz, and Adam Naumowicz. Mizar in a nutshell. *Journal of Formalized Reasoning*, 3(2):153–245, 2010.

6 John Harrison. A Mizar Mode for HOL. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of *Lecture Notes in Computer Science*, pages 203–220, Turku, Finland, 1996. Springer-Verlag.

7 Cezary Kaliszyk and Karol Pąk. Semantics of Mizar as an Isabelle object logic. *Journal of Automated Reasoning*, 63:557–595, 2019.

8 Ondřej Kunčar and Andrei Popescu. A consistent foundation for Isabelle/HOL. In *Interactive Theorem Proving: 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings 6*, pages 234–252. Springer, 2015.

9 Roman Matuszewski and Piotr Rudnicki. Mizar: the first 30 years. *Mechanized mathematics and its applications*, 4(1):3–24, 2005.

10 Michal Muzalewski. *An outline of PC Mizar*. Fondation Philippe le Hodey, 1993.

11 Josef Urban. XML-izing Mizar: Making Semantic Processing and Presentation of MML Easy. In Michael Kohlhase, editor, *Mathematical Knowledge Management, 4th International Conference, MKM 2005, Bremen, Germany, July 15-17, 2005, Revised Selected Papers*, volume 3863 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 2005. doi:10.1007/11618027_23.

12 Josef Urban. MPTP 0.2: Design, implementation, and initial experiments. *Journal of Automated Reasoning*, 37:21–43, 2006.

Now It Compiles!

Certified Automatic Repair of Uncompilable Protocols

Luís Cruz-Filipe
Department of Mathematics and Computer Science, University of Southern Denmark, Odense, Denmark

Fabrizio Montesi
Department of Mathematics and Computer Science, University of Southern Denmark, Odense, Denmark

Abstract

Choreographic programming is a paradigm where developers write the global specification (called choreography) of a communicating system, and then a correct-by-construction distributed implementation is compiled automatically. Unfortunately, it is possible to write choreographies that cannot be compiled, because of issues related to an agreement property known as knowledge of choice. This forces programmers to reason manually about implementation details that may be orthogonal to the protocol that they are writing.

Amendment is an automatic procedure for repairing uncompilable choreographies. We present a formalisation of amendment from the literature, built upon an existing formalisation of choreographic programming. However, in the process of formalising the expected properties of this procedure, we discovered a subtle counterexample that invalidates the original published and peer-reviewed pen-and-paper theory. We discuss how using a theorem prover led us to both finding the issue, and stating and proving a correct formulation of the properties of amendment.

2012 ACM Subject Classification Theory of computation → Concurrency; Theory of computation → Automated reasoning; Software and its engineering → Concurrent programming languages

Keywords and phrases choreographic programming, theorem proving, compilation, program repair

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.11

Related Version *Preprint*: `https://arxiv.org/abs/2302.14622`

Funding This work was partially supported by Villum Fonden, grants no. 29518 and 50079, and the Independent Research Fund Denmark, grant no. 0135-00219.

Acknowledgements We thank the anonymous reviewers for their useful comments, which helped us improve the quality of this article.

1 Introduction

Programming correct implementations of protocols for communicating systems is challenging, because it requires writing a correct program for each participant that performs the right send and receive actions at the right times [23]. *Choreographic programming* [27] is an emerging paradigm that offers a direct solution: protocols are written in a "choreographic" programming language, and then automatically compiled to correct implementations by means of an operation known as *Endpoint Projection* (EPP or projection for short) [5, 14, 16, 17, 20, 24, 25].

Choreographic languages are inspired by the Alice and Bob notation of security protocol [29], in the sense that they offer primitives for expressing communications between different processes. Implementations are usually modelled in terms of a process calculus. Besides being simple, choreographic programming is interesting because it typically includes strong theoretical guarantees, most notably deadlock-freedom and an operational correspondence between choreographies and the (models of the) generated distributed implementations.


© Luís Cruz-Filipe and Fabrizio Montesi;
licensed under Creative Commons License CC-BY 4.0

Not all choreographies can be compiled (or *projected*) to a distributed implementation due to a problem known as "knowledge of choice" [6]. Consider the following choreography for a simple purchase scenario (this example also anticipates some of our syntax).

```
buyer.offer ⟶ seller.x;
If seller.acceptable(x) Then seller.product ⟶ buyer.y; End
                        Else End
```

Listing 1 An unprojectable choreography.

This choreography reads: a `buyer` communicates their offer for the purchase of a product to a `seller`, who stores the offer in their local variable `x`; the `seller` then checks whether the offer is acceptable, and in the affirmative case sends the `product` to `buyer`. This choreography cannot be projected to a behaviourally-equivalent implementation, because `buyer` has to behave differently in the two branches of the conditional. However, this conditional is evaluated by `seller`, and `buyer` has no way of discerning which branch gets chosen.

Choreographies are typically made projectable by adding selections, i.e., communications of constants called *selection labels*.[1] A projectable version of Listing 1 looks as follows.

```
buyer.offer ⟶ seller.x;
If seller.acceptable(x) Then seller ⟶ buyer[left]; seller.product ⟶ buyer.y; End
                        Else seller ⟶ buyer[right]; End
```

Listing 2 A projectable choreography.

This choreography differs from the previous one by the presence of a selection in each branch of the conditional. Specifically, if `seller` chooses the `Then` branch, they now communicate the label `left` to `buyer`. Otherwise, if the `Else` branch is chosen, the label `right` is communicated instead. The key idea is that now the implementation generated for `buyer` can read the label received by `seller` and know which branch of the conditional should be executed. Since labels are constants, compilation can statically verify that `buyer` receives different labels for the different branches, and therefore has "knowledge of choice".

Projection can be smart about knowledge of choice, allowing selections to be kept to a minimum [3]. A process only needs to know which branch of a conditional has been chosen if its behaviour depends on that choice; if the process has to perform the same actions in both branches of a conditional, then this knowledge is irrelevant to it. Knowledge of choice can also be propagated: if a process `q` knows of a choice performed by another process `p`, then either process can forward this information to any other process that needs it.

Amendment. Previous work investigated how unprojectable choreographies can be automatically transformed into projectable ones. Such a transformation is called *amendment* [10, 22] or repair [1, 15]. For example, applying the amendment procedure from [10] to the choreography in Listing 1 returns the choreography in Listing 2 (up to minor differences in notation).

Amendment is interesting for (at least) two reasons. On a practical level, it can suggest valid selection strategies to developers to make their choreographies executable – or even do it automatically, so that they do not have to worry about knowledge of choice. On a theoretical level, it allows porting completeness properties of the set of all choreographies to the set of projectable choreographies.

[1] Selections are essentially the choreographic version of branch selections in session types, or the additive connectives in linear logic [4, 18].

An example of the latter occurs in the study of *Core Choreographies* (CC), a minimalistic theory of choreographic programming [10], where we showed that the set of projectable choreographies in CC is Turing-complete in two steps. First, we showed that CC is Turing-complete, ignoring the question of projectability (the choreographies constructed in the proof are clearly not projectable); then, we defined an amendment procedure and prove an operational correspondence between choreographies and their amendments. As a consequence, the subset of projectable choreographies is also Turing-complete. A similar argument, using the operational correspondence result between projectable choreographies and their implementations, shows that the process calculus used (*Stateful Processes*, or SP) is Turing-complete.

The problem. Our original objective was to formalise amendment and its properties from [10] in the Coq theorem prover, building upon our previous formalisation of CC [12] and its accompanying notion of projection [11]. That formalisation uses a variation of CC based on the theory from [28], which we found more amenable to formalisation. Unfortunately, after formalising the definition of amendment, our attempt to prove its operational correspondence result failed. An inspection of the state of the failed proof quickly led us to a counterexample.

The incorrectness of the original statement jeopardises the subsequent developments that rely on it, in particular Turing completeness of the set of projectable choreographies and of SP. These results were instrumental in substantiating the claim that CC is a "good" minimalistic model for choreographic programming. This finding pointed us towards a more ambitious goal: reformulate the operational correspondence for amendment such that it is correct, and still powerful enough to obtain the aforementioned consequences.

Our motivation to formalise our results in Coq is in line with the increasing awareness in the community that these types of bugs are not uncommon in concurrency theory [26]. In particular, the authors of [15] identified a number of incorrect results in previous work on choreographies, which affect part of the work on choreography repair from [1].

Contribution. To the best of our knowledge, this is the first time that choreography amendment has been formalised. We state and prove a relaxed version of the operational correspondence between choreographies and their amendments in the Coq theorem prover, thus increasing confidence in its correctness. We discuss how working with an interactive theorem prover was instrumental to identifying counterexamples that guided us towards this new, correct formulation that considers all corner cases. We then use our result to formalise the proofs of Turing completeness of projectable choreographies and SP from [10], which were not included in [12].

Structure of the paper. We present the relevant background on CC and its formalisation in Section 2. Section 3 presents the definition of amendment, its formalisation, and discusses and corrects the operational correspondence result from [10]. Section 4 shows that the revised semantic property is still strong enough to derive the Turing completeness results in that work. We discuss related work in Section 5 and conclude in Section 6.

Our exposition assumes some familiarity with interactive theorem proving. We include some Coq code in the article, but the work is intended to be accessible to non-Coq experts.

2 Background

We summarise the latest version of the Coq formalisation of CC [13]. For simplicity, we omit two ingredients that are immaterial for our work: the fact that the language is parameterised on a signature, and the fact that communications have annotations (these are meant to

include information relevant for future implementations in actual programming languages). This allows us to omit some subterms that play no role in the development of amendment.

In our presentation, we use Coq notation with some simplifications for enhanced readability: choreography and process terms are written by overloading the dot symbol (this is not allowed by the Coq notation mechanism), and inductive definitions and inference rules are given with the usual mathematical notation.

2.1 Core Choreographies

We start by giving an overview of Core Choreographies (CC) together with its formalisation in Coq [12].

Syntax. The syntax of CC is given by the following grammar.

```
C ::= η; C | If p.b Then C1 Else C2 | Call X | RT_Call X ps C | End
η ::= p.e ⟶ q.x | p ⟶ q[l]
```

A choreography `C` can be either: a communication η followed by a continuation (η; `C`); a conditional `If p.b Then C1 Else C2`, where the process `p` evaluates the boolean expression `b` to choose between the branches `C1` and `C2`; a procedure call `Call X`, where `X` is the name of the procedure being invoked; a runtime term `RT_Call X ps C`;[2] or the terminated choreography `End`. A communication η can be: a value communication `p.e ⟶ q.x`, read "process `p` evaluates expression `e` locally and sends the result to process `q`, which stores it in its local variable `x`"; or a selection `p ⟶ q[l]`, where the label `l` can be either `left` or `right`, read "`p` sends label `l` to `q`".

Choreographies are formalised in Coq as an inductive type called `Choreography`. Table 1 summarises the Coq types used in this paper and our conventions for ranging over their elements.

Executing a choreography requires knowing the definitions of the choreographies associated to the procedures that can be invoked, as well as the processes involved in those procedures. A set of procedure definitions is defined as a mapping from procedure names to pairs of process names and choreographies.

```
Definition DefSet := RecVar → (list Pid)*Choreography.
```

For simplicity, this is defined as a total function – any procedure that is not used in the choreography can simply be mapped to ([`p`], `End`) for some process `p`. (For technical reasons, the set of process names is not allowed to be empty.)

A *choreographic program* is then a pair consisting of a set of procedure definitions and a choreography (which represents the "main" or "running" choreography).

```
Definition Program := DefSet * Choreography.
```

We write `Procedures P` and `Main P` for the two components of `P`. The set of all processes used by a program `P` is defined as `CCP_pn P`.

It is standard practice to assume some well-formedness conditions about choreographies, e.g., that no process communicates with itself. Choreographic programs have additional well-formedness conditions that must hold for all procedures that can be reached at runtime.

[2] Runtime terms are needed for technical reasons in the definition of the semantics of choreographies [12]. These aspects are irrelevant for the present development.

Table 1 Summary of types in the original Coq formalisation [12, 11].

Type	Variable	Description
`Choreography`	`C`	Choreographies
`Pid`	`p`, `q`, `r`, `s`	Process names (identifiers)
`list Pid`	`ps`	List of process names
`Var`	`x`, `y`, `z`	Variable names
`Val`	`v`	Values
`Expr`	`e`	Expressions (evaluate to values)
`BExpr`	`b`	Boolean expressions (evaluate to Booleans)
`Label`	`l`	Labels (`left` and `right`)
`RecVar`	`X`	Procedure names (or recursive variables)
`DefSet`	`D`	Sets of procedure definitions in CC
`State`	`s`	Maps from variables to values
`Configuration`	`c`	Choreographic programs equipped with states
`TransitionLabel`	`t`	Transition labels
`list TransitionLabel`	`tl`	Lists of transition labels
`Behaviour`	`B`	Behaviours
`option Behaviour`	`mB`	`option` monad for behaviours
`Network`	`N`	Networks
`DefSetB`	`D`	Sets of procedure definitions in SP
`Program`	`P`	Choreographies/networks with procedure definitions

This notion is not decidable in general, but it becomes so in the practical case of programs that only use a finite number of procedures. We return to this aspect at the end of Section 3.2, where it becomes relevant.

▶ **Example 1.** The choreographies in Listings 1 and 2 are well-formed.

Semantics. The intuitive system assumptions in CC are that: processes run independently of each other (concurrently) and possess local stores (associating their variables to values); communications are synchronous; and the network is reliable (messages are not lost nor duplicated, and they are delivered in the right order between any two processes). These assumptions are imported from process calculi, where they are quite standard.

▶ **Example 2.** Since processes run concurrently, it is possible to express choreographies with concurrent behaviour. Consider the following simplification of the factory example in [28].

```
o.order ⟶ p.x; o'.order' ⟶ p'.y; End
```

Listing 3 Parallel orders.

In Listing 3, two processes `o` and `o'` place their respective orders to two different providers `p` and `p'`. Since all processes are distinct and there is no causal dependency between the two communications, the two communications can in principle be executed in any order. This gives rise to a notion of out-of-order execution for choreographies.

The semantics of choreographies in [12] is given as a labelled transition system on configurations, which consist of a program and a (memory) state. States associate to each process a map from variable names to values, which defines the memory of that process.

```
Definition State := Pid → Var → Value
```

States come with some notation: `s [==] s'` says that `s` and `s'` are extensionally equal, and $\mathtt{s}[\![\mathtt{p},\mathtt{x} \Rightarrow \mathtt{v}]\!]$ is the state obtained from updating `s` with the mapping $\mathtt{p},\mathtt{x} \mapsto \mathtt{v}$.

$$\frac{\texttt{v} := \texttt{eval e s p} \quad \texttt{s'} \;[==]\; \texttt{s}[\![\texttt{q,x} \Rightarrow \texttt{v}]\!]}{(\texttt{D},\texttt{p.e} \longrightarrow \texttt{q.x; C,s}) \;\text{—}[\texttt{TL\_Com p v q}]\!\longrightarrow (\texttt{D,C,s'})}\;\texttt{CC\_Com}$$

$$\frac{\texttt{s} \;[==]\; \texttt{s'}}{(\texttt{D},\texttt{p} \longrightarrow \texttt{q[l]; C,s}) \;\text{—}[\texttt{TL\_Sel p q l}]\!\longrightarrow (\texttt{D,C,s'})}\;\texttt{CC\_Sel}$$

$$\frac{\texttt{beval b s p} = \texttt{true} \quad \texttt{s} \;[==]\; \texttt{s'}}{(\texttt{D},\texttt{If p.b Then C1 Else C2,s}) \;\text{—}[\texttt{TL\_Tau p}]\!\longrightarrow (\texttt{D,C1,s'})}\;\texttt{CC\_Then}$$

$$\frac{\texttt{disjoint\_eta\_rl}\ \eta\ \texttt{t} \quad (\texttt{D},\texttt{C},\texttt{s}) \;\text{—}[\texttt{t}]\!\longrightarrow (\texttt{D,C',s'})}{(\texttt{D},\eta;\ \texttt{C,s}) \;\text{—}[\texttt{t}]\!\longrightarrow (\texttt{D},\eta;\ \texttt{C',s'})}\;\texttt{CC\_Delay\_Eta}$$

$$\frac{\texttt{disjoint\_p\_rl p t} \quad (\texttt{D},\texttt{C1,s}) \;\text{—}[\texttt{t}]\!\longrightarrow (\texttt{D,C1',s'}) \quad (\texttt{D},\texttt{C2,s}) \;\text{—}[\texttt{t}]\!\longrightarrow (\texttt{D,C2',s'})}{(\texttt{D},\texttt{If p.b Then C1 Else C2,s}) \;\text{—}[\texttt{t}]\!\longrightarrow (\texttt{D,If p.b Then C1'Else C2',s'})}\;\texttt{CC\_Delay\_Cond}$$

Figure 1 Semantics of choreographic configurations (selected rules).

With these concepts in place, we can show some representative transition rules for choreographic configurations in Figure 1.[3] Transitions have the form $(\texttt{D},\texttt{C},\texttt{s}) \;\text{—}[\texttt{t}]\!\longrightarrow (\texttt{D,C'},\texttt{s'})$, where `t` is a transition label that allows for observing what happened in the transition.

Rule `CC_Com` deals with the execution of a value communication from a process `p` to a process `q`: if the expression `e` at `p` can be evaluated to a value `v` (first condition, which uses the auxiliary function `eval`), then the communication term is consumed and the state of the receiver is updated such that its receiving variable `x` is now mapped to value `v`. The transition label `TL_Com p v q` denotes that `p` has communicated the value `v` to `q`, modelling what would be visible on a network.

Rule `CC_Sel` is similar but does not alter the state of the receiver (the role of selections will be clearer when we explain the language for modelling implementations of choreographies). The transition label `TL_Sel p q l` registers the communication of label `l` from `p` to `q`.

Rule `CC_Then` deals with the case in which a process `p` can evaluate the guard `b` of a conditional to `true` (using the auxiliary function `beval`), proceeding to the then-branch of the conditional. The transition label `TL_Tau p` denotes that process `p` has executed an internal action (τ is the standard symbol for such actions in process calculi).

Rule `CC_Delay_Eta` deals with out-of-order execution of communications, formalising the reasoning anticipated in Example 2. Specifically, the continuation of an interaction η is allowed to perform a transition (without affecting η) as long as the transition does not involve any of the processes in η. The latter condition is checked by the first premise of the rule, `disjoint_eta_rl` η `t`, which checks that the processes mentioned in η are distinct from those mentioned by the transition label `t`. Rule `CC_Delay_Cond` applies the same reasoning to the out-of-order execution of conditionals.

The reflexive and transitive closure of the transition relation is written $\text{—}[\texttt{tl}]\!\longrightarrow^*$, where `tl` is a list of transition labels.

▶ **Example 3.** For any `D` and `s` such that `order` evaluates to `v` at `o` and `order'` evaluates to `v'` at `o'`, according to `eval`,

$$(\texttt{D},\texttt{o.order} \longrightarrow \texttt{p.x; o'.order'}\longrightarrow \texttt{p'.y; End},\texttt{s}) \;\text{—}[\texttt{TL\_Com o v p;TL\_Com o' v' p'}]\!\longrightarrow^* (\texttt{D,End,s'})$$

[3] In the actual formalisation, the transition relation was defined in two layers for technical reasons. This technicality is immaterial for our development, since our results follow from the rules shown here.

and

$$(\texttt{D},\texttt{o.order} \longrightarrow \texttt{p.x; o'.order'} \longrightarrow \texttt{p'.y; End},\texttt{s}) \xrightarrow{\texttt{[TL\_Com o' v' p';TL\_Com o v s]}}{}^{*} (\texttt{D},\texttt{End},\texttt{s'})$$

where $\texttt{s'} \ [==] \ \texttt{s}[\![\texttt{s1},\texttt{x} \Rightarrow \texttt{v}]\!][\![\texttt{s2},\texttt{x} \Rightarrow \texttt{v'}]\!]$.

2.2 Processes

Implementations of choreographies are modelled in Stateful Processes (SP) [11], a formalised process calculus following [28]. SP follows the standard way of representing systems of communicating processes, where the code of each process is given separately and communication is achieved when processes perform compatible I/O actions.

Syntax. The code of a process is written as a behaviour (`B`), following the grammar below.

```
B ::= p!e; B | p?x; B | p+l; B | p & mB1 // mB2 | If b Then B1 Else B2 | Call X | End
mB ::= None | Some B
```

These terms are the local counterparts to the choreographic terms of CC. The first two productions deal with value communication. Specifically, a send action `p!e; B` sends the result of evaluating `e` to the process `p` and then continues as `B`. Dually, a receive action `p?x; B` receives a value from `p`, stores it in `x`, and then continues as `B`.

Selections are implemented by the primitives `p+l; B` and `p & mB1 // mB2`. The former sends the label `l` to the process `p` and continues as `B`. The latter is a branching term, where `mB1` and `mB2` are the behaviours that the process will execute upon receiving `left` or `right`, respectively. To cover the case where a process does not offer a behaviour for a specific label, `mB1` and `mB2` have type `option Behaviour`.

Conditionals (`If b Then B1 Else B2`), procedure calls (`Call X`), and the terminated behaviour (`End`) are standard.

Processes are intended to run together in networks. These are formalised as maps from processes to behaviours.

```
Definition Network := Pid → Behaviour.
```

Networks come with some convenient notation for their construction: `p[B]` is the network that maps `p` to `B` and all other processes to `End`; and `N | N'` is the composition of `N` and `N'`. In particular, `(N | N') p` returns `N p` if this is different from `End`, and `N' p` otherwise.[4]

▶ **Example 4.** The following network implements the choreography in Listing 2.

```
buyer[ seller!offer; seller & Some (seller?y; End) // Some End ] |
seller[ buyer?x; If acceptable(x) Then buyer+left; buyer!product; End
                                  Else buyer+right; End ]
```

For the semantics of networks, we need two additional ingredients. The network $\texttt{N} \setminus \texttt{p}$ is obtained from `N` by redefining `p`'s behaviour as `End` (`p` is "removed" from `N`). The relation `N (==) N'` holds if the networks `N` and `N'` are extensionally equal.

As in CC, processes in a network can invoke procedures defined in a separate set.

```
Definition DefSetB := RecVar → Behaviour.
```

[4] This asymmetry does not matter for our results, since we never compose networks that define nonterminated behaviours for the same processes.

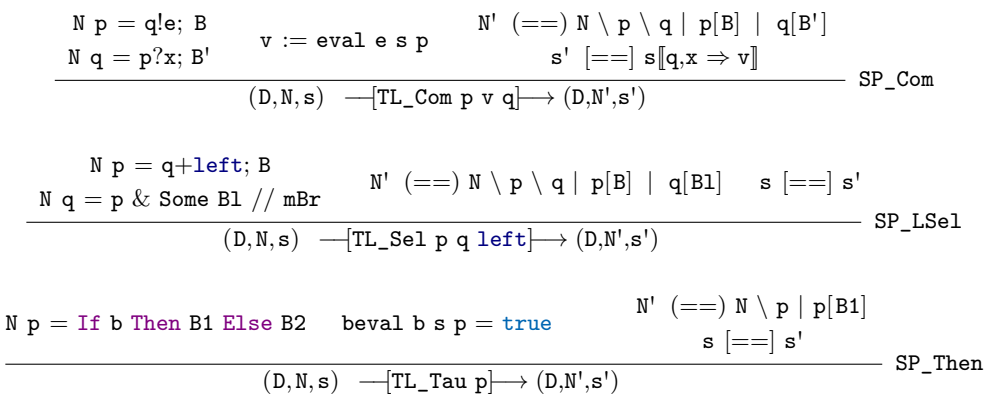


Figure 2 Semantics of network configurations (selected rules).

A `Program` in SP consists of a set of procedure definitions and a network. Assuming that this set is globally accessible simplifies the definition of the semantics of SP; in implementations, each process would have a local copy of this set, or of the subset of procedures it needs to invoke.

```
Definition Program := DefSetB * Network.
```

We use `D` to range over elements of `DefSetB` and `P` to range over elements of `Program`, as for choreographies (the difference will be clear from the context).

Semantics. The semantics of SP is also given as a labelled transition system on configurations that consist of a program and a memory state, as in CC. A selection of the transition rules defining this semantics is displayed in Figure 2.

Rule `SP_Com` matches a send action at a process `p` with a compatible receive action at another process `q` (conditions `N p = q!e; B` and `N q = p?x; B'`). The resulting network `N'` is obtained from `N` by replacing the behaviours of these processes with their continuations (`N \ p \ q | p[B] | q[B']`). The update to the state is handled as in CC.

Rules `SP_LSel` and its dual `SP_RSel` model, respectively, the selection of the left and right branches offered by a branching term, by inspecting the label sent by the sender. Rule `SP_Then` captures the case in which a conditional enters its then-branch.

2.3 Endpoint Projection (EPP)

Choreographies are compiled to networks by a procedure defined in two layers. We begin by defining a *behaviour projection*, which compiles the desired behaviour from a choreography for a given process. This procedure is a partial function, and since all functions in Coq are total it was formalised as the following inductive relation.

```
bproj : DefSet → Choreography → Pid → Behaviour → Prop
```

Term `bproj D C p B`, written `⟦D,C | p⟧ == B`, reads "the projection of `C` on `p` in the context of the set of procedure definitions `D` is `B`".[5]

[5] The parameter `D` is used for projecting procedure calls, which is immaterial to the current work.

$$\frac{\llbracket \mathtt{D}, \mathtt{C} \mid \mathtt{p} \rrbracket == \mathtt{B}}{\llbracket \mathtt{D}, \mathtt{p} \longrightarrow \mathtt{q}[\mathtt{l}]; \mathtt{C} \mid \mathtt{p} \rrbracket == \mathtt{q}{+}\mathtt{l}; \mathtt{B}}\ \mathtt{bproj\_Pick} \qquad \frac{\mathtt{p} \neq \mathtt{r} \quad \mathtt{q} \neq \mathtt{r} \quad \llbracket \mathtt{D}, \mathtt{C} \mid \mathtt{r} \rrbracket == \mathtt{B}}{\llbracket \mathtt{D}, \mathtt{p} \longrightarrow \mathtt{q}[\mathtt{l}]; \mathtt{C} \mid \mathtt{r} \rrbracket == \mathtt{B}}\ \mathtt{bproj\_Sel}$$

$$\frac{\mathtt{p} \neq \mathtt{q} \quad \llbracket \mathtt{D}, \mathtt{C} \mid \mathtt{q} \rrbracket == \mathtt{B}}{\llbracket \mathtt{D}, \mathtt{p} \longrightarrow \mathtt{q}[\mathtt{left}]; \mathtt{C} \mid \mathtt{q} \rrbracket == \mathtt{p}\ \&\ \mathtt{Some}\ \mathtt{B}\ //\ \mathtt{None}}\ \mathtt{bproj\_Left}$$

$$\frac{\llbracket \mathtt{D}, \mathtt{C1} \mid \mathtt{p} \rrbracket == \mathtt{B1} \quad \llbracket \mathtt{D}, \mathtt{C2} \mid \mathtt{p} \rrbracket == \mathtt{B2}}{\llbracket \mathtt{D}, \mathtt{If}\ \mathtt{p.b}\ \mathtt{Then}\ \mathtt{C1}\ \mathtt{Else}\ \mathtt{C2} \mid \mathtt{p} \rrbracket == \mathtt{If}\ \mathtt{b}\ \mathtt{Then}\ \mathtt{B1}\ \mathtt{Else}\ \mathtt{B2}}\ \mathtt{bproj\_Cond}$$

$$\frac{\mathtt{p} \neq \mathtt{r} \quad \llbracket \mathtt{D}, \mathtt{C1} \mid \mathtt{p} \rrbracket == \mathtt{B1} \quad \llbracket \mathtt{D}, \mathtt{C2} \mid \mathtt{p} \rrbracket == \mathtt{B2} \quad \mathtt{B1}\ [\mathtt{V}]\ \mathtt{B2} == \mathtt{B}}{\llbracket \mathtt{D}, \mathtt{If}\ \mathtt{p.b}\ \mathtt{Then}\ \mathtt{C1}\ \mathtt{Else}\ \mathtt{C2} \mid \mathtt{p} \rrbracket == \mathtt{B}}\ \mathtt{bproj\_Cond'}$$

Figure 3 Selected rules for behaviour projection.

Intuitively, behaviour projection is computed by going through the choreography; for each choreographic term, projection constructs the local action that the input process should perform to implement it. The rules defining `bproj` that are relevant for this work are those that deal with selections and conditionals. These are shown in Figure 3.

A label selection $\mathtt{p} \longrightarrow \mathtt{q}[\mathtt{l}]$ is projected as either: (i) the sending of label `l` to `q` for process `p` (rule `bproj_Pick`); (ii) the appropriate branching term that receives `l` from `p` for process `q`, where only the branch for `l` offers a behaviour (rules `bproj_Left` and the dual rule `bproj_Right`); or (iii) no action for any other process (rule `bproj_Sel`).

Similarly, a conditional in a choreography is projected to a conditional for the process that evaluates the guard (rule `bproj_Cond`). However, projecting conditionals becomes complex when considering the other processes, because this requires dealing with the problem of knowledge of choice discussed in Section 1. This case is handled by rule `bproj_Cond'`, which sets the result of projection to be the "merging" of the projections of the two branches, written `B1 [V] B2 == B`, if this is defined.

Intuitively, merging attempts to build a behaviour `B` from two behaviours `B1` and `B2` that have similar structures, but may differ in the labels that they accept in branching terms. For all terms but branchings, merging requires term equality and then proceeds homomorphically in subterms. This is exemplified by the rules `merge_End`, `merge_Sel`, and `merge_Cond` in Figure 4.

The interesting part regards the merging of branching terms, which has a rule for every possible combination. Figure 4 shows two representative cases. If two branching terms have branches for different labels, then we obtain a branching term where the two branches are combined as exemplified by rule `merge_Branching_SNNS`. If two branching terms have overlapping branches, then we try to merge them as exemplified by rule `merge_Branching_SSSS`.[6]

As we remarked, merging (seen as a partial function) can be undefined, for example `End` and `p+l; End` cannot be merged. This gives rise to the notion of *projectability* anticipated in Section 1: a choreography `C` is projectable on a process `p` in the context of a set of procedure definitions `D` if `bproj` is defined for those parameters.

```
Definition projectable_B D C p := ∃ B, ⟦D,C | p⟧ == B.
```

[6] Due to space constraints, the names of these rules have been abbreviated in Figure 4.

$$\frac{}{\texttt{End}\ [\texttt{V}]\ \ \texttt{End} == \texttt{End}}\ \texttt{merge\_End} \qquad \frac{\texttt{B1}\ [\texttt{V}]\ \ \texttt{B2} == \texttt{B}}{\texttt{p+1; B1}\ [\texttt{V}]\ \ \texttt{p+1; B2} == \texttt{p+1; B}}\ \texttt{merge\_Sel}$$

$$\frac{\texttt{Bt1}\ [\texttt{V}]\ \ \texttt{Bt2} == \texttt{Bt} \quad \texttt{Be1}\ [\texttt{V}]\ \ \texttt{Be2} == \texttt{Be}}{\texttt{If p.e Then Bt1 Else Bt2}\ [\texttt{V}]\ \texttt{If p.e Then Be1 Else Be2} == \texttt{If p Then Bt Else Be}}\ \texttt{merge\_Cond}$$

$$\frac{}{\texttt{p \& Some bL // None}\ [\texttt{V}]\ \texttt{p \& None // Some bR} == \texttt{p \& Some bL // Some bR}}\ \texttt{SNNS}$$

$$\frac{\texttt{bL1}\ [\texttt{V}]\ \ \texttt{bL2} == \texttt{bL} \quad \texttt{bR1}\ [\texttt{V}]\ \ \texttt{bR2} == \texttt{bR}}{\texttt{p \& Some bL1 // Some bR1}\ [\texttt{V}]\ \texttt{p \& Some bL2 // Some bR2} == \texttt{p \& Some bL // Some bR}}\ \texttt{SSSS}$$

Figure 4 Definition of the merge relation (selected rules).

This is generalised by `projectable_C D C ps`, which states that `C` is projectable for all processes in the list `ps`. For a choreographic program `P` to be projectable, written `projectable_P P`, we require that `Main P` be projectable for all processes in `CCP_pn P` and that all procedures be projectable for the processes that they use.

With projectability in place, Endpoint Projection (EPP) is defined as a function that maps a projectable choreographic program to a process program in SP.

```
Definition epp P : projectable_P P → Program.
```

The second argument of `epp` is a proof of `projectable_P P`, but the formalisation includes a lemma showing that the result does not depend on this term.

▶ **Example 5.** The behaviours of `buyer` and `seller` in Example 4 are the respective projections for these two processes of the choreography in Listing 2.

The definition of `epp` allows us to apply program extraction and obtain a certified compiler from choreographies to networks – see [8] for a discussion and examples.

2.4 Turing completeness

The authors of [12] formalise that CC is Turing-complete, in the sense that all of Kleene's partial recursive functions [21] can be implemented as a choreography for a suitable notion of implementation. The proof is interesting because it considers CC instantiated with very restricted computational capabilities at processes: values are natural numbers; expressions can only be the constant zero, a variable, or the successor of a variable; and Boolean expressions can only check if the two variables at a process contain the same value. Kleene's partial recursive functions are then implemented concurrently, by making processes communicate according to appropriate patterns.

According to [12], a choreographic program `P` implements `f:PRFunction m` (representing a partial recursive function $f : \mathbb{N}^m \to \mathbb{N}$) with input processes $\texttt{ps}_1,\ldots,\texttt{ps}_m$ and output process `q` iff: for any state `s` where $\texttt{ps}_1,\ldots,\texttt{ps}_m$ contain the values $\texttt{n}_1,\ldots,\texttt{n}_m$ in their variable `x`, (i) if $f(\texttt{n}_1,\ldots,\texttt{n}_m) = \texttt{n}$, then all executions of `P` from `s` terminate, and do so in a state where `q` stores `n` in its variable `x`; and (ii) if $f(\texttt{n}_1,\ldots,\texttt{n}_m)$ is undefined, then execution of `P` from `s` never terminates.[7] This is captured by the Coq term `implements P m f ps q`, where `ps` is the vector $\texttt{ps}_1,\ldots,\texttt{ps}_m$.

[7] This is a straightforward adaption of the definition of function implementation by a Turing machine [31].

The proof of Turing completeness encodes partial recursive functions to choreographies that are not always projectable, since they contain no selections but some processes behave differently in conditionals.

3 Amendment

Several works have studied how unprojectable choreographies can be automatically amended to obtain projectable versions [1, 10, 22]. In particular, [10] developed an amendment procedure based on merging. The informal idea that we explore below is that, whenever a choreography contains a conditional, amendment adds selections, in both branches, from the process evaluating the guard to any processes whose behaviour projection is undefined. Intuitively, this makes the output choreography projectable.

▶ **Example 6.** Let `C` be the choreography:

```
p.e ⟶ q.x; If r.b Then (r.e' ⟶ p.y; End) Else End
```

Amending `C` as described yields the following choreography, `A`:

```
p.e ⟶ q.x; If r.b Then (r ⟶ p[l]; r.e' ⟶ p.y; End)
                   Else (r ⟶ p[r]; End)
```

Amendment is claimed to have the following properties.

▶ **Lemma 7** (Amendment Lemma [10], rephrased)**.** *For every choreography `C`:*

1. *The amendment of `C` is well-formed.*
2. *The amendment of `C` is projectable.*
3. *If `DA`, `A`, and `A'` are obtained by amending all procedures in `D` as well as `C` and `C'`, then `(D,C,s)` —[tl]⟶* `(D,C',s')` iff `(DA,A,s)` —[tl']⟶* `(DA,A',s')` for some `tl'`.*

In point one, well-formedness refers to a set of syntactic conditions that exclude ill-written choreographies, e.g., self-communications (interactions where a process communicates with itself) [10]. Points one and two are simple to prove by induction on the structure of the choreography. Point three, unfortunately, is wrong. When attempting to formalise this result, we failed, and the state of the proof led us to the following counterexample.

▶ **Example 8.** Given a suitable state, the choreography `C` from Example 6 can make a transition to `C'` defined as

```
p.e ⟶ q.x; r.e' ⟶ p.y; End
```

by rules `CC_Delay_Eta` and `CC_Then`. However, `C`'s amendment `A` can move to

```
p.e ⟶ q.x; r ⟶ p[l]; r.e' ⟶ p.y; End
```

by the same rules, but this is neither the amendment of `C'`, nor can it reach it since the offending selection term is blocked by the initial communication.

In hindsight, this is not so surprising: amendment introduces causal dependencies that were not present in the source choreography. However, this intuition was completely missed by both authors and reviewers of the original publications discussing amendment [9, 10]. Therefore, amending a choreography can remove some execution paths.

In the rest of this section, we show how to define amendment formally in Coq, and formulate a correct variation of Lemma 7.

3.1 Definition

We decompose the definition of amendment in three functions: one for identifying the processes that need to be informed of the outcome of a specific conditional; one for prepending a list of selections to a choreography; and one that recursively amends a whole choreography by using the former two. This division simplifies not only the definition, but also the structure of proofs about amendment since they can be modularised.

To identify the processes that require knowledge of choice, we define a function `up_list` (`up` is short for "unprojectable processes"). This function recursively goes through a list `ps` of processes and checks for each process in the list whether the choreography `If p.b Then C1 Else C2` can be projected on that process (function `projectable_B_dec` does precisely this test). If this is not the case, then the process is added to the result. (Since projectability is relative to a set of procedure definitions, this also needs to be given as an argument, `D`.)

```
Fixpoint up_list D p b ps C1 C2 : list Pid := match ps with
  | nil ⇒ nil
  | r :: ps' ⇒ let ps'' := up_list D p b ps' C1 C2 in
    if (r =? p) then ps''
    else if projectable_B_dec D (If p.b Then C1 Else C2) r
         then ps''
         else (r :: ps'') end.
```

Note that `p`, as the evaluator of the conditional, does not need to be informed of the outcome. This justifies the check `r =? p`, whose inclusion also avoids introducing self-communications and simplifies subsequent proofs. (Function `up_list` is essentially a filter, but since it tests two predicates we found the current definition to be easier to work with than either a composition of two filters or a filter with a predicate defined as a conjunction.)

The second ingredient is straightforward: given a process `p`, a selection label `l`, and a choreography `C`, it recursively adds selections of `l` from `p` to each element of a list `ps`.

```
Fixpoint add_sels p l ps C : Choreography := match ps with
  | nil ⇒ C
  | r :: ps' ⇒ p ⟶ r[l]; add_sels p l ps' C end.
```

We can now define amendment following the informal procedure described in [10]. Given a list of processes `ps`, we go through a choreography `C`; whenever we meet a conditional on a process `p`, we compute the list of processes from `ps` with an undefined projection and prepend the branches of the conditional with appropriate selections. (We show only the most interesting cases.)

```
Fixpoint amend D ps C := match C with
  | eta; C' ⇒ eta; (amend D ps C')
  | If p.b Then C1 Else C2 ⇒
     let l := up_list D p b ps (amend D ps C1) (amend D ps C2) in
     If p.b Then (add_sels p left l (amend D ps C1))
            Else (add_sels p right l (amend D ps C2))
  | ... end.
```

Amendment is generalised to sets of procedure definitions in the obvious way.

```
Definition amend_D D ps : DefSet := fun X ⇒ (fst (D X), amend D ps (snd (D X))).
```

To amend a program `P`, the parameter `ps` of the previous functions is instantiated with the set of processes used in `P`.

```
Definition amend_P P :=
  (amend_D (Procedures P) (CCP_pn P), amend (Procedures P) (CCP_pn P) (Main P)).
```

This formal definition corresponds to the informal one given in [10]. In particular, all our examples are formalised in Coq.

▶ **Example 9.** Consider the following choreography.

```
If p.b Then (p.e ⟶ q.x; q.e' ⟶ r.y; End)
       Else (q.e'' ⟶ r.y; End)
```

Here, p decides if (i) it will communicate a value to q that can be used in the computation of a later message from q to r (so q acts as a sort of proxy) or (ii) q should just compute the value that it will communicate to r by itself. Amendment is smart enough to notice that while q requires a selection from p, r does not since it behaves in the same way (receive from q on x). Therefore, amending the choreography returns the following.

```
If p.b Then (p ⟶ q[left]; p.e ⟶ q.x; q.e' ⟶ r.y; End)
       Else (p ⟶ q[right]; q.e'' ⟶ r.y; End)
```

3.2 Syntactic Properties

We now discuss the key properties of amendment.

Amendment preserves well-formedness of choreographies (`Choreography_WF`) and choreographic programs (`Program_WF`). This follows from the fact that `add_sels` preserves all syntactic properties of well-formedness, using induction.

```
Lemma amend_Choreography_WF : Choreography_WF C → Choreography_WF (amend D ps C).

Lemma amend_Program_WF : Program_WF (D,C) → Program_WF (amend_D D ps,amend D ps C).
```

(For simplicity, we omit universal quantifiers at the beginning of lemmas.)

Likewise, it is straightforward to prove that amending for some processes guarantees that the output choreography is projectable on all those processes.

```
Lemma amend_projectable_C : projectable_C (amend_D D ps) (amend D ps C) ps.
```

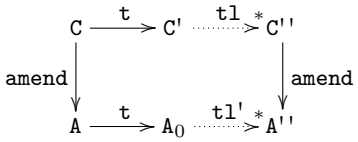
We do not generalise this result to choreographic programs: it is not straightforward to do and our later development does not need it. The issue we encounter is related to a problem discussed in [12, 11]: computing the set of processes and procedures that are used by a choreography can require an infinite number of steps, and is therefore not definable as a function in Coq. (A simple example is a program with an infinite set of procedure definitions where each procedure $\mathtt{X}_i$ invokes the next procedure $\mathtt{X}_{i+1}$.)

The function `CCP_pn` used in the definition of `amend_P` does return the set of processes involved in a program `P`, but it does not check that `P` does not define unused procedures. If this is the case, these procedures may use processes not in `CCP_pn P`, and therefore they may be unprojectable for these processes. Rather than stating a result with complex side-conditions as hypotheses, we prove projectability of particular programs applying `amend_projectable_C` to `Main P` and to the bodies of all procedure definitions. The development in the next section uses this strategy.

3.3 Semantic Properties

We now discuss how the formulation of the semantic relation between a choreography and its amendment needs to be changed.

The counterexample shown earlier suggests allowing both choreographies to perform additional transitions in order to unblock and remove lingering selections introduced by amendment. (In our example, this would be the communication from `p` to `q`.) The correspondence would then look as follows, where the dotted lines correspond to existentially quantified terms:



and the list of transition labels `tl` can be obtained from `tl'` by removing some selections.

Our attempt to prove this result showed that it holds for all cases but one: when the transition `t` is obtained by applying rule `CC_Delay_Cond`.

▶ **Example 10.** We show a minimal counterexample. Consider the choreography

```
If p.b Then (q.e ⟶ r.x; q.e' ⟶ p.x; End)
       Else (q.e ⟶ r.x; End)
```

and its amendment

```
If p.b Then (p ⟶ q[left]; q.e ⟶ r.x; q.e' ⟶ p.x; End)
       Else (p ⟶ q[right]; q.e ⟶ r.x; End) .
```

The original choreography can execute the communication between `q` and `r`, reaching

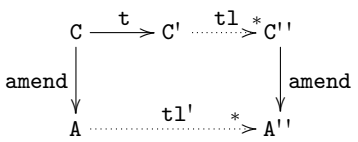
```
If p.b Then (q.e' ⟶ p.x; End) Else End
```

but its amendment needs to run the conditional and a selection before it can execute the same communication.

There are two ways to solve this problem: changing the definition of amendment, or refining the correspondence result further. We opted for the second route, for two reasons: first, we get to keep the original definition given on paper in [10]; second, making amendment clever enough to recognise this kind of situations requires a non-local analysis of the choreography (i.e., looking at the structure of the branches of conditionals instead of simply checking for projectability of the term). In our example, such an analysis could detect that the additional selections from `p` to `q` could be added only after the communication from `q` to `r`, solving the issue.

Therefore, our final correspondence result requires that the amendment of a choreography be allowed to perform additional transitions *before* it matches the transition performed by the original choreography. Since a transition may invoke rule `c_delay_Cond` more than once, this means that the orders of the transitions performed by the original choreography and its amendment can be arbitrary permutations of each other that respect causal dependencies between transitions (ignoring the extra selections).

The correspondence result we prove looks as follows:



where `t`:: `tl` can be obtained from `tl'` by removing some selections and permuting labels.

To formalise this in Coq, we introduce a relation `sel_exp` ("selection expansion") between lists of transition labels.

```
Inductive sel_exp :=
| se_base tl tl' : Permutation tl tl' → sel_exp tl tl'
| se_extra p q l tl tl' tl'' : sel_exp tl tl' →
    Permutation (TL_Sel p q l::tl') tl'' → sel_exp tl tl''.
```

We can now prove a correct version of the correspondence between choreographies and their amendments. There are four results in total: the one depicted above and its generalisation to the case where `t` is replaced with a list of transition labels; and the two dual results where the amendment of a choreography moves first. We show the two more general statements.

```
Lemma amend_complete_many : Program_WF (D,C) → (D,C,s) —[tl]⟶* (D,C',s') →
  ∃ tl' tl'' C'' s'', sel_exp (tl++ tl') tl'' ∧ (D,C',s') —[tl']⟶* (D,C'',s'')
    ∧ (amend_D D ps,amend D ps C,s) —[tl'']⟶* (amend_D D ps,amend D ps C'',s'').

Lemma amend_sound_many : Program_WF (D,C) → let (D' := amend_D D ps) in
  (D', amend D ps C,s) —[tl]⟶* (D',C',s') →
  ∃ tl' tl'' C'' s'', (D',C',s') —[tl']⟶* (D', amend D ps C'',s'')
    ∧ (D,C,s) —[tl'']⟶* (D,C'',s'') ∧ sel_exp tl'' (tl++ tl').
```

The challenging part of the work in this section was understanding what the correct formulation of these results should be. Once we reached this formulation, proofs were relatively straightforward inductions on the given transitions (10–15 lines of Coq code per case).

The formalisation of the amendment lemma consists of 6 definitions, 50 lemmas, and 4 examples, with a total of roughly 1050 lines of Coq code.

4 Implications of Amendment

In the previous section, we had to weaken the original statement for the semantic correspondence guaranteed by amendment that was given in [10]. Since the original statement was used in the proofs of Turing completeness for projectable core choreographies and SP, it is natural to investigate whether our new formulation still yields these results.

For uniformity, we start by reformulating the Turing completeness result for core choreographies from [12], where process names are identified with natural numbers.

```
Theorem CC_Turing_Complete : ∀ n (f:PRFunction n),
  ∃ P, Program_WF P ∧ implements P f (vec_1_to_n n) 0.
```

The theorem states that, for any partial recursive function `f`, there exists a well-formed choreographic program `P` that implements `f` with input processes $1, \ldots,$ `n` and output process 0. The proof is a straightforward combination of results already presented in [12].

Combining this result with our lemmas about amendment yields that the fragment of projectable core choreographies is also Turing-complete.

```
Lemma projCC_Turing_Complete : ∀ n (f:PRFunction n),
  ∃ P, Program_WF P ∧ projectable_P P ∧ implements P f (vec_1_to_n n) 0.
```

The proof is split into several steps. The most interesting sublemma is the one establishing that amending a choreography that implements a function yields a choreography that implements the same function. This is formulated as a general result about amendment.

```
Lemma amend_implements : Program_WF P →
  implements P f ps q → implements (amend_P P) f ps q.
```

The proof uses the fact that terminated choreographies cannot execute further to show that the list of additional transitions added to the original choreography by the amendment lemma (`tl` in the last diagram) must be empty.

The remaining lemmas for `projCC_Turing_Complete` deal with projectability of the amended choreography, as discussed in the previous section, and are simple to prove.

Since amended choreographies are projectable, we can further apply the EPP theorem from [11] to show that SP is also Turing-complete.

```
Theorem SP_Turing_Complete : ∀ n (f:PRFunction n),
  ∃ P, Network_WF (Net P) ∧ SP_implements P f (vec_1_to_n n) 0.
```

The definition of `SP_implements` is a straightforward adaptation of the definition of `implements` for choreographies. The proof of `SP_Turing_Complete` follows a similar strategy to the one for `projCC_Turing_Complete`: we prove a sublemma `epp_implements` stating that the EPP of a choreography that implements a function `f` is a process program that implements `f`.

The formalisation of this section consists of 2 definitions and 11 lemmas, totaling about 250 lines of Coq code. The conciseness of this development substantiates our previous comment on not providing a complex lemma for projectability of programs, at the end of Section 3.2.

5 Related Work

To the best of our knowledge, our work is the first formalisation of amendment, its properties, and its intended consequences.

The work nearest to ours is the original presentation of the amendment procedure that inspired us [10]. As we discussed, the behavioural correspondence for amendment that the authors state is wrong. We developed a correct statement and managed to update and formalise the proofs of Turing completeness for CC and SP accordingly. Our formalisation of the behavioural correspondence also clarifies what semantic property amendment actually guarantees, which might be important for future work and practical applications of amendment.

Amendment or similar procedures have been investigated also for other choreographic languages [1, 22]. In all these works, the general idea is to repair choreographies by identifying the specific places where additional communications are required for implementability. However, the differences between the underlying languages and the techniques used make the resulting procedures very different from ours.

The pioneer work of [22] only deals with finite choreographies without out-of-order execution. This allows for an amendment procedure that analyses the syntax of the choreography: it inspects the choreography and checks that the first communications in the two branches of a conditional have the same sender.

Instead, the choreographies in [1] are automata. Their technique also follows the idea of looking at the possible transitions the choreography can perform in order to repair it, and it uses a notion of realisability inherited from previous work [2] to establish its correctness. Unfortunately, later work [15] showed that Theorem 2 in [2] is incorrect, invalidating the proof of the result that is used in [1].

Differently, our definition of amendment uses merging, first introduced in [3], and projection. While the underlying idea remains the same, this formulation is more intuitive, as the connection between unprojectability and amendment becomes direct. This also simplifies our development and yields shorter proofs.

Our work is based on the most recent version of the formalisation of CC, SP, and EPP [13], which was originally introduced in [11, 12]. We did not need to modify this formalisation in order to use it for our development, which shows that it reached a sufficient level of maturity for being used as a library to reason about choreographies.

Other formalisations of choreographies include: Kalas, a choreographic programming language that targets CakeML [30]; the choreographic DSL Zooid, a Coq library for verifying that message passing code respects a given multiparty session type (these are abstract choreographies without computation) [7]; and multiparty GV, a formalised functional language with a similar goal to Zooid [19].

6 Conclusion

We have presented the first formalisation of an amendment procedure for choreographies. Our work is based on a previous formalisation of CC and its accompanying notion of EPP, which we used as a library. We found this formalisation to be modular and complete enough to support the separate development presented here. In the same spirit of generality and reusability, our formalisation does not add any assumptions about CC that were not present in the library.

Our development is an illustration of how theorem provers can assist in research: interacting with Coq guided us to (i) discovering that the semantic property of amendment found in the background literature for this work is wrong, and (ii) a correct formulation that is still powerful enough for its intended use in previous work.

The formalisation of amendment is amenable to extraction, and therefore our work potentially offers a basis for a certified transformer from arbitrary choreographies in CC to projectable ones. In the future, we plan on studying how this transformer can be integrated into existing frameworks for choreographic programming.

Our notion of amendment is intrinsically related to how EPP is defined for CC. In the literature, there are choreographic languages with a more permissive notion of knowledge of choice, e.g., where replicated processes intended to be used as services are allowed to be involved in only one branch of a conditional [3, 5]. It would be interesting to study how amendment can be adapted to these settings.

References

1 Samik Basu and Tevfik Bultan. Automated choreography repair. In Perdita Stevens and Andrzej Wasowski, editors, *Procs. FASE*, volume 9633 of *Lecture Notes in Computer Science*, pages 13–30. Springer, 2016. doi:10.1007/978-3-662-49665-7_2.

2 Samik Basu, Tevfik Bultan, and Meriem Ouederni. Deciding choreography realizability. In John Field and Michael Hicks, editors, *Procs. POPL*, pages 191–202. ACM, 2012. doi:10.1145/2103656.2103680.

3 Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8:1–8:78, 2012. doi:10.1145/2220365.2220367.

4 Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In Josée

Desharnais and Radha Jagadeesan, editors, *Procs. CONCUR*, volume 59 of *LIPIcs*, pages 33:1–33:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

5 Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In Roberto Giacobazzi and Radhia Cousot, editors, *Procs. POPL*, pages 263–274. ACM, 2013. doi:10.1145/2429069.2429101.

6 Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On global types and multi-party sessions. In Roberto Bruni and Jürgen Dingel, editors, *Procs. FORTE*, volume 6722 of *Lecture Notes in Computer Science*, pages 1–28. Springer, 2011. doi:10.1007/978-3-642-21461-5_1.

7 David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. Zooid: a DSL for certified multiparty computation: from mechanised metatheory to certified multiparty processes. In Stephen N. Freund and Eran Yahav, editors, *Procs. PLDI*, pages 237–251. ACM, 2021. doi:10.1145/3453483.3454041.

8 Luís Cruz-Filipe, Lovro Lugović, and Fabrizio Montesi. Certified compilation of choreographies with `hacc`. In *Formal Techniques for Distributed Objects, Components, and Systems - 43rd IFIP WG 6.1 International Conference, FORTE 2023, Held as Part of the 18th International Federated Conference on Distributed Computing Techniques, DisCoTec 2023, Lisbon, Portugal, June 19-23, 2023, Proceedings*, pages 29–36, 2023. doi:10.1007/978-3-031-35355-0_3.

9 Luís Cruz-Filipe and Fabrizio Montesi. A core model for choreographic programming. In Olga Kouchnarenko and Ramtin Khosravi, editors, *Procs. FACS*, volume 10231 of *Lecture Notes in Computer Science*, pages 17–35. Springer, 2017. doi:10.1007/978-3-319-57666-4_3.

10 Luís Cruz-Filipe and Fabrizio Montesi. A core model for choreographic programming. *Theor. Comput. Sci.*, 802:38–66, 2020. doi:10.1016/j.tcs.2019.07.005.

11 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Certifying choreography compilation. In Antonio Cerone and Peter Csaba Ölveczky, editors, *Procs. ICTAC*, volume 12819 of *Lecture Notes in Computer Science*, pages 115–133. Springer, 2021. doi:10.1007/978-3-030-85315-0_8.

12 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Formalising a Turing-complete choreographic language in Coq. In Liron Cohen and Cezary Kaliszyk, editors, *Procs. ITP*, volume 193 of *LIPIcs*, pages 15:1–15:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ITP.2021.15.

13 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. A formal theory of choreographic programming. *J. Autom. Reason.*, 67(2):21, 2023. doi:10.1007/s10817-023-09665-3.

14 Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies: Theory and implementation. *Log. Methods Comput. Sci.*, 13(2), 2017. doi:10.23638/LMCS-13(2:1)2017.

15 Alain Finkel and Étienne Lozes. Synchronizability of communicating finite state machines is not decidable. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *Procs. ICALP*, volume 80 of *LIPIcs*, pages 122:1–122:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ICALP.2017.122.

16 Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. Choreographies as objects. *CoRR*, abs/2005.09520, 2020. URL: `https://arxiv.org/abs/2005.09520`.

17 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.

18 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9, 2016. Also: POPL, pages 273–284, 2008. doi:10.1145/2827695.

19 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.

20 Sung-Shik Jongmans and Petra van den Bos. A predicate transformer for choreographies – computing preconditions in choreographic programming. In Ilya Sergey, editor, *Procs.*

ESOP, volume 13240 of *Lecture Notes in Computer Science*, pages 520–547. Springer, 2022. doi:10.1007/978-3-030-99336-8_19.

21 Stephen Cole Kleene. *Introduction to Metamathematics*, volume 1. North-Holland Publishing Co., 1952.

22 Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. Amending choreographies. In António Ravara and Josep Silva, editors, *Procs. WWW*, volume 123 of *EPTCS*, pages 34–48, 2013. doi:10.4204/EPTCS.123.5.

23 Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In Tom Conte and Yuanyuan Zhou, editors, *Procs. ASPLOS*, pages 517–530. ACM, 2016. doi:10.1145/2872362.2872374.

24 Alberto Lluch-Lafuente, Flemming Nielson, and Hanne Riis Nielson. Discretionary information flow control for interaction-oriented specifications. In Narciso Martí-Oliet, Peter Csaba Ölveczky, and Carolyn L. Talcott, editors, *Logic, Rewriting, and Concurrency*, volume 9200 of *Lecture Notes in Computer Science*, pages 427–450. Springer, 2015. doi:10.1007/978-3-319-23165-5_20.

25 Hugo A. López and Kai Heussen. Choreographing cyber-physical distributed control systems for the energy sector. In Ahmed Seffah, Birgit Penzenstadler, Carina Alves, and Xin Peng, editors, *Procs. SAC*, pages 437–443. ACM, 2017. doi:10.1145/3019612.3019656.

26 Petar Maksimovic and Alan Schmitt. HOCore in Coq. In Christian Urban and Xingyuan Zhang, editors, *Procs. ITP*, volume 9236 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2015. doi:10.1007/978-3-319-22102-1_19.

27 Fabrizio Montesi. *Choreographic Programming*. Ph.D. Thesis, IT University of Copenhagen, 2013. URL: https://www.fabriziomontesi.com/files/choreographic-programming.pdf.

28 Fabrizio Montesi. *Introduction to Choreographies*. Cambridge University Press, 2023.

29 Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978. doi:10.1145/359657.359659.

30 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, *Procs. ITP*, volume 237 of *LIPIcs*, pages 27:1–27:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ITP.2022.27.

31 Alan M. Turing. Computability and λ-definability. *J. Symb. Log.*, 2(4):153–163, 1937. doi:10.2307/2268280.

Lessons for Interactive Theorem Proving Researchers from a Survey of Coq Users

Ana de Almeida Borges
University of Barcelona, Spain

Annalí Casanueva Artís
Ifo Institut, München, Germany

Jean-Rémy Falleri
Univ. Bordeaux, Bordeaux INP, CNRS, UMR 5800 LaBRI, F-33400 Talence, Institut Universitaire de France, France

Emilio Jesús Gallego Arias
Université Paris Cité, CNRS, Inria, IRIF, F-75013, Paris, France

Érik Martin-Dorel
IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3, Toulouse, France

Karl Palmskog
KTH Royal Institute of Technology, Stockholm, Sweden

Alexander Serebrenik
TU Eindhoven, The Netherlands

Théo Zimmermann
LTCI, Télécom Paris, Institut Polytechnique de Paris, France

Abstract

The Coq Community Survey 2022 was an online public survey of users of the Coq proof assistant conducted during February 2022. Broadly, the survey asked about use of Coq features, user interfaces, libraries, plugins, and tools, views on renaming Coq and Coq improvements, and also demographic data such as education and experience with Coq and other proof assistants and programming languages. The survey received 466 submitted responses, making it the largest survey of users of an interactive theorem prover (ITP) so far. We present the design of the survey, a summary of key results, and analysis of answers relevant to ITP technology development and usage. In particular, we analyze user characteristics associated with adoption of tools and libraries and make comparisons to adjacent software communities. Notably, we find that experience has significant impact on Coq user behavior, including on usage of tools, libraries, and integrated development environments.

2012 ACM Subject Classification Software and its engineering → Formal methods; General and reference → Empirical studies

Keywords and phrases Coq, Community, Survey, Statistical Analysis

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.12

Supplementary Material *Software (Source Code and Data)*: `https://doi.org/10.5281/zenodo.7930567` [12]

Acknowledgements The authors would like to thank the survey participants, the Coq Survey Working Group members who are not simultaneously authors of this paper (Yves Bertot, Nathan Cassee, Jim Fehrle, Jerome Hugues, Barry Jay, Matthieu Sozeau and Enrico Tassi), the survey beta testers, and the translator team (Yishuai Li, Oling Cat and Weidu Kuang).

© Ana de Almeida Borges, Annalí Casanueva Artís, Jean-Rémy Falleri, Emilio Jesús Gallego Arias, Érik Martin-Dorel, Karl Palmskog, Alexander Serebrenik, and Théo Zimmermann;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

Online surveys are employed by organizations [16, 28] and researchers [30] to obtain aggregate data about software communities, e.g., with the aim of understanding community demographics, programming practices, and desired improvements to a software ecosystem.

Some interactive theorem provers (ITPs) such as Coq, Isabelle/HOL, and Lean have many active users across academia and industry, and have comparable popularity to programming languages on developer platforms such as GitHub and Stack Overflow [22]. Yet, to our knowledge, the only public summary for an ITP community survey is Braibant's for the Coq Community Survey 2014 [5]. However, due to the development of Coq itself and changes in the Coq community since 2014, many of its results and conclusions are currently obsolete.

In this paper, we present the results of a new survey of Coq users, which we call the Coq Community Survey 2022. The survey was conducted publicly online during February 2022, using Inria's LimeSurvey platform. The main goal of the survey was to obtain an updated picture of the Coq user community and inform future decisions taken by the Coq development team (*Coq Team* for short) and other stakeholders in the Coq ecosystem. In particular, the survey aimed to help the Coq Team to better prioritize issues and features for the Coq system, and enable effective decision-making about matters pertaining to the software ecosystem maintained by Coq users in academia and industry [3]. One specific Coq Team impetus for the survey was a community-wide discussion on Coq's name and logo in April 2021 on the Coq mailing list [31], centering on the name's pronunciation in English.[1]

The survey was designed and deployed by a working group (WG) that was formed following a public call by the Coq Team. The survey had 109 questions, some of which were conditionally visible, and was available in English and Chinese [8, 7]. Broadly, the survey asked about 1) use of Coq features, integrated development environments (IDEs), libraries, plugins, and tools, 2) views on renaming and improving Coq, and 3) demographic data, such as age, gender, and experience with Coq. The survey received 466 submitted responses.

The initial analysis and presentation of the survey [9, 10, 11] was limited to descriptive response summaries and basic inferred data, such as defining *graduate students* by combining responses to highest completed academic degree and student status. Our response analysis here goes beyond the initial presentation by performing statistical analyses – multiple variable regression analyses that unveil characteristics of Coq users associated with different usage habits. We also compare aggregate data of Coq survey respondents with data from similar surveys in related software communities (Haskell [15] and Stack Overflow [28]), which we believe can give some perspective on the contention that the Coq community is oriented towards programming languages [4]. We make the following contributions:

1. We present the first Coq community survey since 2014, and to our knowledge the survey of an ITP community with the largest number of respondents thus far.
2. We have communicated descriptive results to the Coq community early, by publishing a series of articles on the Coq Discourse forum and GitHub issues to make specific stakeholders (mostly IDE maintainers) aware of the results of highest interest to them.
3. We provide novel survey data analyses by performing multiple variable regression analyses to answer one main research question: **How do different categories of respondents use Coq?** In addition, we look at satisfaction and needs for improvements for the same categories of respondents. Our code and results are part of our public dataset [12].

[1] While the renaming issue is outside paper scope, aggregate renaming responses are available [12, 2].

The paper is structured as follows. Section 2 describes the survey design and deployment, the data analysis process, and the methodology for our statistical analyses, including the definition of our respondent categories of interest. Section 3 mentions the previous descriptive analysis of the survey results, including some examples, and performs a comparison between the Coq community and the Stack Overflow and Haskell communities, based on their respective 2022 surveys. Section 4 compares Coq use across different categories of Coq users. Section 5 compares differing satisfaction and user needs across the same categories of Coq users. Section 6 discusses threats to validity of our results, and measures taken to mitigate them. Finally, Section 7 discusses our results and concludes.

2 Methodology

The survey was designed, deployed, managed, and analyzed by members of a working group (WG), which was formed after a public call for participation by the Coq core team. Members of this WG met weekly during a year and a half, first to design the survey, then to advertise it to the Coq community, and finally to analyze and share the results.

2.1 Survey Design and Deployment

Survey questions were collaboratively created by WG members. The WG took inspiration from the previous Coq Community Survey in 2014 [5], and similar surveys for programming languages [20]. The WG made efforts to adhere to survey best practices [19], e.g., on ordering demographic questions late to avoid their answers affecting other questions.

The survey was available in English [8] and Chinese [7], having been first created in English and then translated to Chinese by a team of volunteers. These volunteers also translated the answers to the open questions back to English. Ideally, the survey would have been translated to more languages, particularly the ones that have an existing community of Coq users who are not necessarily proficient in English. For example, there are Coq learning resources in Chinese [26] and Japanese [25], as well as dedicated categories for several languages in the Coq Discourse forum. The WG contacted colleagues with the necessary language knowledge to perform Chinese, Japanese, and Russian translations, but only secured a commitment for the Chinese translation.

The survey had 109 questions, some of which were only visible depending on answers to previous questions. Broadly, the survey asked about 1) use of Coq features, IDEs, libraries, plugins, and tools, 2) views on renaming Coq, Coq improvements, and issues such as inclusion, and 3) demographic data, such as age, gender, and length of experience with Coq and other ITPs or functional programming languages. All the questions were optional.

After a preliminary version of the survey was finalized, it was sent to a small group of beta testers, who were asked to fill out that preliminary version, time themselves, and send feedback on their experience. Some changes were made due to this feedback, and the initial prediction of the time it would take to fully complete the closed-question part of the survey (30 minutes) was estimated based on this beta testing. We made this estimation available to potential survey respondents on the first page of the survey and in our announcements.

Before survey deployment, we contacted the Inria Ethics Review Board (ERB), who informed us that full ERB approval of our survey was not required and referred us to Inria's Data Protection Officer (DPO) for data protection compliance matters. Based on our communication with the DPO, we developed a GDPR conformance statement and followed best practices for data protection. In particular, we removed the raw data and only share aggregated data, including open text answers which we manually sanitized to remove any personally identifiable elements.

The survey was deployed on Inria's LimeSurvey platform and was open during the month of February 2022. The deployed survey was advertised to Coq users primarily via the following Coq community forums: Discourse, Zulip chat, Coq-club mailing list, Coq Twitter, Coq Reddit, and the Coq website. The WG also advertised in more general venues and other related communities, such as the Types-announce mailing list, Agda and Lean Zulip chats, and on language-specific forums in Chinese and in French (mailing lists of GDR IM and GPL). These announcements asked only previous or current Coq users to answer. Still, the survey included the question "How long have you used Coq?" with the option to answer "I have never used Coq." Those who chose that option were still asked a number of questions, but most of the survey was hidden, since most questions assumed previous Coq experience.

2.2 Analysis Process

After the survey closed, WG members collaboratively analyzed the response data. For GDPR compliance, the WG decided to keep the raw survey data within the EU. Therefore, the WG used Inria-hosted tools and restricted raw data access to members located in the EU. The Chinese translators translated the Chinese answers back into English, and then two authors wrote the analysis code in Python within a Jupyter notebook, using the `pandas` and `matplotlib` libraries. Plots were created for each closed-answer survey question (often multiple plots for each question) and automatically deployed to a static Inria-hosted website for later inclusion in summary blog posts (see Section 3). In order to perform the statistical analyses described in Section 4, we exported pre-processed data from the Jupyter notebook for a third author to perform regressions using Stata.[2] We provide the Jupyter notebook and Stata code as supplementary material [12] to allow scrutiny of the source code, and also provide all the generated plots for the survey questions. However, due to privacy concerns (risk of re-identifying specific respondents), we do not provide the raw survey data, which is required to reproduce the analyses using the Jupyter notebook.

2.3 Statistical Analyses

We do a multiple variable regression analysis to unveil the characteristics of Coq users associated with different usage habits and expectations. This statistical method allows us to establish the relationship between multiple regressors (or independent variables) and one dependent variable (or outcome) [32]. Including several variables at the same time is needed to disentangle the distinct effect of possibly correlated variables. If we did not include such correlated variables simultaneously, part of the effect of the excluded variables would be captured by the coefficient of the included correlated ones. One must however keep in mind that: 1) these results cannot be interpreted causally and 2) there can still be variables correlated with the dependent or independent variables that were not included in the regression.

For each category of users, we test several hypotheses. This will lead to over-rejecting null hypotheses (i.e., some associations between variables will be considered true when in reality they are not present). To overcome this problem, it is necessary to explicitly consider the multiplicity of the testing framework. Thus, for all our estimates, we report, in addition to classic standard errors (in parentheses), the Romano-Wolf-corrected p-values for multiple hypothesis testing (in brackets) [27, 6]. This correction has a higher ability to correctly reject

[2] Stata is a software package for statistical analysis, see `https://www.stata.com`.

false null hypotheses than previous techniques such as Bonferroni and Holm [17]. Following common standards in fields with a long tradition of statistical analysis such as economics, and considering that, by controlling the family-wise error rate, the Romano-Wolf correction is more conservative than alternative approaches to multiple hypothesis testing (giving us robust levels of significance), we interpret estimates as significant as soon as the p-values are below the 0.1 threshold (instead of the 0.05 threshold which would be more common in empirical software engineering).

2.3.1 Defining Categories of Interest

We studied four user characteristics that we considered could be relevant in explaining differences in how respondents use Coq: their experience with Coq, their learning status, their application-domain for Coq, and their location. More specifically:

Experience level We define two binary variables that illustrate various stages of experience with Coq. The first variable represents whether the user has more than 2 years of experience, and the second variable whether the user has more than 5 years of experience. About 64% of our sample has more than 2 years of experience and about 40% more than 5. The omitted category represents users with less than 2 years of experience. We use these variables (corresponding to arbitrary thresholds) instead of a numeric variable representing the number of years of experience because experience does not always have a linear association with outcomes, such as the likelihood of using a particular feature. In particular, we can expect that, beyond some level of experience, additional years of experience are very unlikely to be associated with any additional changes in the outcome.

Learners We define a binary variable corresponding to learners. To define learner, we use the answer to the question "For what purpose have you used Coq?". A learner is a respondent who answered any of "Learning Coq" or "Learning something other than Coq" to this (multiple-choice) question, without also responding any of "Teaching Coq", "Teaching something other than Coq", "Academic research", or "Industrial research / application". About 17% of our sample are defined as learners. The way that learners use Coq may be related to their learning context, and thus significantly differ from other Coq users. Understanding these differences could inform researchers and engineers trying to make proof assistants easier to learn.

Application domains for using Coq Coq can be used for various applications: verifying software, hardware, or theoretical systems, formalizing existing or new mathematical results, etc. The way of using Coq may differ depending on the application domain (extraction comes to mind as being specifically relevant for software verification), but also on the users' background. This is why we decided to test whether software verification specifically (the most common objective for Coq users) was related to differing practices. About 67% of our sample use Coq for software verification.

Location Various world regions have different research traditions (such as putting more or less focus on theory) and may also have different technical culture. This is why it seemed important to evaluate whether this relates to different ways of using Coq. Our baseline is Europe, where about half of our respondents live. We define two binary variables to represent the respondents' location: one for North America (which represents about 25% of our respondents' location) and one for the rest of the world (excluding both North America and Europe). This is why Europe does not appear explicitly in our tables.

2.3.2 Controls

In addition to the variables presented above, our regressions include additional variables as controls. In particular, we include: the age of respondents, their gender (more precisely, a binary variable for whether the respondent declared being a woman), their operating system (two binary variables for Windows and macOS), their research area (a binary variable for whether the respondent does research in math or logic), their employment (a binary variable for whether the respondent is employed by a private organization), and whether they hold a PhD. We do not analyze the results of the regressions for these variables; they are only there to avoid capturing something that could be correlated both to the independent variables and to the outcome. For example, if we include years of experience without including age, we might be capturing the effect of being of a certain generation on using a specific feature that was particularly popular during a specific period. Because years of experience is correlated with age, part of the association between age and the outcome is captured by the coefficient of years of experience. Yet, in this example, the experience would not be relevant.

2.3.3 Selecting Population to Analyze

Our survey received 466 submitted responses. However, for the statistical analyses, we restrict ourselves to the 390 respondents who say that they have used Coq in the last year (as what we want to assess is current Coq practices). Furthermore, we can only perform regressions on data where our (dependent and independent) variables, including our controls, are defined. We remove 18 respondents who did not provide their age. This results in looking at a subsample of 372 responses. On one specific outcome (IDE satisfaction), we look at an even smaller subsample of 367 responses, because respondents need to have answered at least one IDE satisfaction question for this outcome to be defined.

3 Descriptive Observations and Comparison to Other Communities

3.1 Descriptive Analysis and Observations

After the survey closed, the WG performed a descriptive analysis of the data. The results were continuously shared with the community through a series of articles posted to the Coq Discourse forum. The topics were as follows: *Who is using Coq and in what context?* [9]; *How are people using Coq?* [10]; *How is Coq used? (features, tools, libraries)* [11]. They include many plots deployed from a GitLab repository to a perennial URL,[3] in SVG and PNG format. Results from the articles and additional results on renaming Coq were also presented at the Coq Workshop 2022 [1, 2]. Specific results were shared as repository issues:

Proof General `https://github.com/ProofGeneral/PG/issues/671`
Company-Coq `https://github.com/cpitclaudel/company-coq/issues/258`
CoqIDE `https://github.com/coq/coq/issues/16580`
VsCoq `https://github.com/coq-community/vscoq/issues/308`
Coqtail `https://github.com/whonore/Coqtail/issues/277`
jsCoq `https://github.com/jscoq/jscoq/issues/261`
coq_jupyter `https://github.com/EugeneLoy/coq_jupyter/issues/46`
Continuous integration `https://github.com/coq-community/manifesto/issues/141`

[3] `https://thzimmer.gitlabpages.inria.fr/coq-survey-2022-assets/asset-listing.html`

In addition, the WG prepared detailed internal reports and presentations to the Coq core team on the results to the renaming questions to help the team make a decision regarding a possible renaming of Coq. At the time of writing, no decision has been made official yet.

Here, we present only a small fraction of these descriptive results: a selection of demographic markers in Figure 1 and IDE and operating system (OS) use in Figure 2. We comment on these while comparing the Coq community to two other relevant communities, and refer the reader to the Discourse forum articles for a more detailed presentation [9, 10, 11].

3.2 Comparison of the Coq Community to Other Communities

In this section, we compare the Coq community (CC) to two distinct baseline populations. We use the 2022 Stack Overflow survey results [28] as a proxy for the "general" programming community, since Stack Overflow is a popular website used by millions of software developers. We will refer to this population as SO. We use the 2022 Haskell survey results [15] as a proxy for the functional programming (FP) community, as Haskell is a mature but not obsolete functional programming language. We believe this population is interesting due to the proximity between the ITP and FP communities. We will refer to this population as HC.

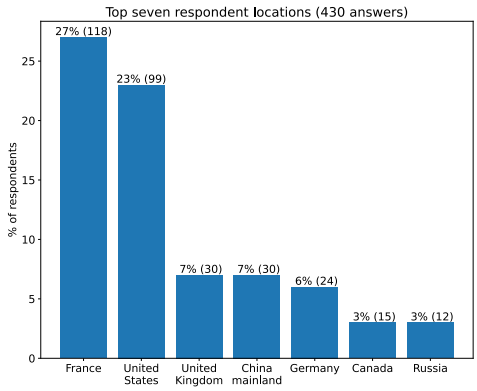
Location. The United States is consistently one of the top-5 locations, as are Germany and the United Kingdom, for all three populations. As can be seen in Figure 1a, CC has a large ratio of French respondents, even outnumbering US respondents, likely for historical reasons (Coq was originally developed in France). A notable difference is that Indian respondents, numerous in the SO population, are rare in the FP and CC populations, indicating a missed opportunity for FP and ITP.

Gender. There is no data for HC, but a comparison can be done between CC and SO. It shows the same trend of an overwhelming ratio of respondents identifying as men (about 90%), while only about 6% identify as women, and the remaining 4% as non-binary or other.

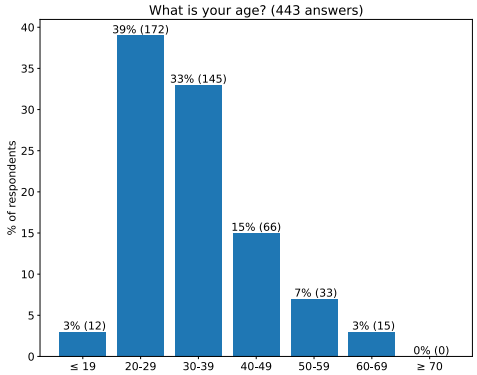
Age. There is no data for the HC population, but SO and CC (Figure 1b) can be compared. Although the bins are slightly different between SO and CC (shifted by five years), the overall trend seems similar between the two populations. Very young programmers (less than 20 years old) are rare (about 5%), with the bulk of programmers being approximately between 20 and 40 years old. Older programmers (more than 50 years old) represent about 10% of both populations.

Education. CC has fewer bins for education than either HC or SO (Figure 1c). We enable direct comparisons between surveys by merging bins where possible and dropping bins and corresponding answers otherwise, since bins are mutually exclusive. This analysis yields responses for no diploma for SO 3%, HC 2%, CC 1%; high school diploma for SO 26%, HC 17%, CC 5%; bachelor's degree for SO 45%, HC 37%, CC 18%; master's degree for SO 23%, HC 30%, CC 31%; PhD degree for SO 3%, HC 14%, CC 46%. These results place HC between SO and CC in terms of educational achievement, but closer to SO given that CC has such a high skew towards doctoral degrees.

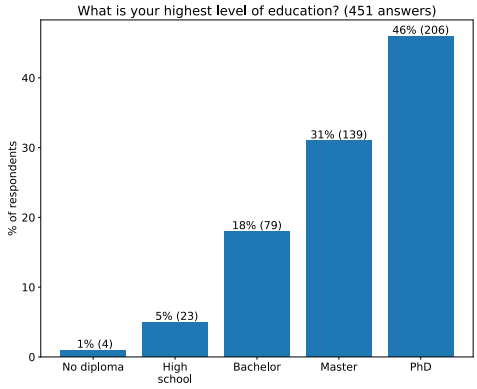
Academic use. We now compare the "academic" bins of the three surveys, even if their definitions are not exactly the same. In both the CC survey and the HC survey, respondents are asked if they operate in an academic context and if they are students or not. In the



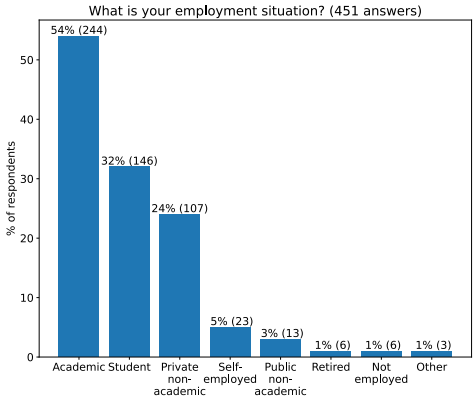
(a) Top locations of residence of the respondents.



(b) Age of the respondents.



(c) Highest education level completed by respondents.

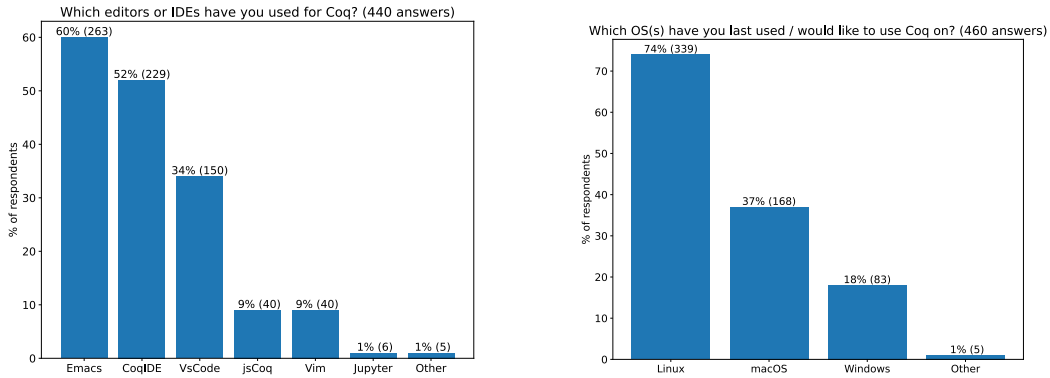


(d) Employment situation of the respondents.

Figure 1 Plots depicting the location, age, education level and employment situation of the respondents to the 2022 Coq Community Survey.

SO survey, academic researcher and student are exclusive categories. In CC, as can be seen in Figure 1d, 54% of respondents are academically employed, and 32% are students. In HC, 18% of the respondents use Haskell in an academic context and 15% are students. In comparison, only 4.42% of Stack Overflow survey respondents are academic researchers and 9.13% are students. In conclusion, CC and HC respondents operate in an academic context much more frequently than SO respondents. Even in comparison with the HC respondents, CC is even more tied to academia, where it represents about half of the respondents.

IDEs. The bins are not the same across the surveys, but we can compare the following three main IDEs: Vim, Emacs, and VS Code. See Figure 2a for IDE use in the CC population. VS Code is used by about 70% of SO, 43% of HC and 34% of CC. Emacs is used by about 5% of SO, 30% of HC and 60% of CC. Vim is used by about 23% of SO, 40% of HC and 9% of CC. When considering only Emacs, Vi(m), and VS Code, the Coq survey answers suggest that Coq users are significantly more likely to use Emacs than either Haskell users or developers on Stack Overflow. This is partly driven by long-time Coq users preferring Emacs (and its ProofGeneral package) and VS Code support for Coq being more recent.



(a) IDEs used by the respondents.

(b) Operating systems used by the respondents.

Figure 2 Plots on IDE and OS use of the survey respondents.

OSes. The bins are not the same across the surveys, but we can compare the following three main OSes: Linux, Windows and macOS. For the SO survey, the best data is professional usage as it reflects the coding environment. See the data for CC in Figure 2b. Windows is used by about 49% of SO, 15% of HC and 18% of CC. macOS is used by about 31% of SO, 33% of HC and 37% of CC. Linux is used by about 40% users of SO, 86% of HC and 74% of CC.

These percentages suggest that Linux-based operating systems are dominant among both Haskell and Coq users, with only a small minority of Windows users and a similarly-sized minority using macOS in both communities. In contrast, among Stack Overflow users, a near-majority use Windows. Windows use is slightly higher among Coq survey respondents than Haskell respondents, which is partly driven by new Coq users.

4 Analysis of Coq Use for Different Population Groups

In this section, we analyze how different population groups we identified in the survey use Coq differently, by looking at various outcomes.

4.1 Installation of Coq, Usage of Packages and Features, and CI

In this section, we analyze the results of Table 1, which contains the following outcomes:

opam A binary variable indicating whether respondents installed Coq with opam (a software package manager for OCaml [23] and for Coq as well [29]), either directly, or relying on wrapper scripts provided by the Coq Platform [24]. Because Coq has been around for so long, there are many possible ways to install it. It is packaged in many Linux distributions and generic package managers, and there are binary installers for the main operating systems. The most flexible installation method is by using opam, because it gives access to the whole ecosystem of Coq packages. This is the most common installation method (used by 76% of respondents), but it is also more complex, so the expectation is that users start relying on it only when they encounter the need.

External general libraries A binary variable for whether respondents are "casual" or "advanced" users of any of the external general libraries Coq-Extlib, Coq-std++, Math Classes, Mathematical Components, or TLC. Coq is distributed with a standard library and some embedded tools, but it also comes with a rich package ecosystem. Power users

Table 1 Usage of opam, ecosystem libraries, tools, automation, SSReflect, Continuous Integration (CI), and extraction.

VARIABLES	(1) opam	(2) External general libraries	(3) Number of external tools	(4) Automation	(5) SSReflect	(6) CI	(7) Extraction
> 2 years of experience	**0.183*** **	0.107	-0.118	0.110	-0.0196	0.0512	**0.231*** **
	(0.0576)	(0.0663)	(0.233)	(0.0620)	(0.0644)	(0.0600)	(0.0653)
	[0.008]	[0.253]	[0.818]	[0.253]	[0.818]	[0.720]	[0.005]
> 5 years of experience	0.00177	**0.169** **	**0.945*** **	**0.186** **	**0.138***	**0.320*** **	**0.169** **
	(0.0594)	(0.0684)	(0.241)	(0.0639)	(0.0664)	(0.0619)	(0.0673)
	[0.967]	[0.045]	[0.001]	[0.016]	[0.059]	[0.001]	[0.045]
Learner	**-0.282*** **	-0.0882	0.193	-0.175	-0.0363	**-0.197** **	-0.0513
	(0.0619)	(0.0712)	(0.250)	(0.0665)	(0.0691)	(0.0645)	(0.0701)
	[0.001]	[0.685]	[0.872]	[0.116]	[0.872]	[0.042]	[0.872]
Software verification	0.0470	**0.165** **	**0.742*** **	**0.192*** **	**0.126***	0.0371	**0.290*** **
	(0.0473)	(0.0544)	(0.191)	(0.0509)	(0.0528)	(0.0493)	(0.0536)
	[0.533]	[0.013]	[0.002]	[0.002]	[0.051]	[0.533]	[0.001]
North America	0.00277	0.00986	0.0795	-0.125	-0.0637	0.00597	0.00654
	(0.0519)	(0.0597)	(0.210)	(0.0558)	(0.0580)	(0.0541)	(0.0588)
	[1.000]	[1.000]	[0.999]	[0.170]	[0.860]	[1.000]	[1.000]
Other locations	-0.00968	-0.124	-0.0873	-0.0269	-0.0673	-0.122	**0.172***
	(0.0617)	(0.0710)	(0.250)	(0.0663)	(0.0689)	(0.0643)	(0.0699)
	[0.960]	[0.358]	[0.956]	[0.956]	[0.802]	[0.290]	[0.089]
Additional controls	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Mean of dependent variable	0.758	0.366	0.995	0.702	0.288	0.315	0.446
Observations	372	372	372	372	372	372	372

Note: Each column corresponds to a regression. Standard errors are in parentheses, Romano-Wolf adjusted p-values in brackets. Stars represent different levels of significance (*** p<0.01, ** p<0.05, * p<0.1) according to Romano-Wolf adjusted p-values. Independent variables are described in §2.3.1 and additional controls in §2.3.2. Mean of dependent variable corresponds to the mean of the outcome of each column.

rely on this package ecosystem to be more productive. With this outcome and the next one, we evaluate how different types of users take advantage (or not) of the package ecosystem. Only 37% of respondents are users of a general external library.

Number of external tools A natural number variable counting how many external tools respondents said they were using ("Casual user" or "Advanced user") among a long list: Mtac2, MetaCoq, and Coq-Elpi (from the "Tactic languages" question), as well as all 16 tools from the "Ecosystem plugins and tools" question. Respondents to the survey only use 0.995 of the 19 listed external tools on average.

Automation A binary variable for whether respondents are "casual" or "advanced" users of any of several core automation tools (Micromega, Nsatz, Ring, and the solvers for logic and equality from the standard library, such as firstorder) or external automation tools (AAC Tactics, CoqHammer, and SMTCoq). 70% of respondents are using such a tool.

SSReflect A binary variable for whether respondents are "casual" or "advanced" users of the SSReflect proof language. SSReflect is a consistent set of tactics providing an alternative to the standard tactics. It has been distributed as part of the main Coq package for more than 5 years. 29% of our respondents use SSReflect.

CI A binary variable for whether respondents use Continuous Integration (CI) on any of their projects. CI is the practice of automatically running tests at every change when developing software. This is a very common practice today, which helps to produce better quality software [13]. CI can also be used on mechanized proof projects, usually not to run tests, but simply to verify that the proofs are still accepted. This is particularly useful when developing Coq libraries, because CI can help ensure that a library remains compatible with several consecutive versions of Coq, even though its maintainers only

test one version on their machines. Because of this main application, it is expected that respondents who do not yet have big enough projects will not use CI. 31% of our respondents declared they do.

Extraction A binary variable for whether respondents are "casual" or "advanced" users of any of the officially supported extraction targets. Extraction is a Coq feature allowing the automatic generation of executable code in OCaml, Haskell or Scheme from a program written in Gallina, the functional programming language embedded inside Coq. Since Coq can be used to prove a Gallina program correct with respect to its specification, extraction is a useful mechanism to produce executable code for this program, and it is expected to be strongly associated with using Coq for verifying software. 45% of our respondents said they use extraction.

Experience level

We observe statistically significant results relating experience to each of our outcomes. For most of our outcomes (external general libraries, number of external tools, automation, SSReflect and CI), the threshold to see a statistically significant difference is to reach more than 5 years of experience. The most important jump being in the use of CI, which is of the same magnitude as the mean of the variable over our population.

Sometimes, 2 years of experience are enough to see significant differences: the use of opam increases (18 percentage points (pp) more respondents using opam) at the 2 years of experience threshold, and the use of extraction increases twice, at each of the two experience thresholds (23 pp more respondents using extraction after 2 years, and 17 pp more respondents using extraction after 5 years).

Learners

Learners use opam significantly less often (28 pp fewer respondents use opam when considering learners compared to non-learners with every other independent variable fixed, including experience). This may be explained by learners having a stronger need for an easy installation method and a lesser need for the access to packages that installing with opam provides. The only other statistically significant difference robust to multiple hypothesis testing is in the use of CI. 20 pp fewer learners use CI for their projects compared to non-learners, which can be explained by learners not having any project that really requires the use of CI. Other coefficients are estimated to be negative as well, but are not statistically significant or not robust to multiple hypothesis testing corrections.

Software verification

Besides experience level, the other category of users associated with many statistically significant differences in their use of Coq are respondents applying Coq to do software verification. These users are more likely to use external general libraries (16 additional pp), they use more external tools (an average of 0.74 additional tools, which is close in magnitude to the average number of external tools used by our respondents), they are more likely to use automation, more likely to use SSReflect, and more likely to use extraction. In short, we can say that these respondents (who represent about 67% of our respondents) are the ones who make the most use of the Coq package ecosystem and features. This may not come as a surprise for, e.g., the use of extraction, which is a feature particularly suited to software verification, but may be more surprising when talking about the use of external general libraries or SSReflect, given that MathComp (one of these general libraries) and SSReflect were not created to verify software, but to formalize pure mathematics.

Beyond the previously listed significant results, we can also note a significant, but difficult to interpret difference on the use of extraction by respondents from other locations (excluding Europe and North America).

4.2 User Interfaces

Table 2 Usage of Integrated Development Environment (IDE).

	(1)	(2)	(3)	(4)
VARIABLES	Emacs	CoqIDE	VS Code	jsCoq
> 2 years of experience	0.0164	**0.220*** **	0.00848	0.0135
	(0.0651)	(0.0679)	(0.0679)	(0.0418)
	[0.991]	[0.009]	[0.991]	[0.991]
> 5 years of experience	0.0778	-0.0318	-0.0134	-0.0146
	(0.0672)	(0.0700)	(0.0700)	(0.0431)
	[0.688]	[0.963]	[0.963]	[0.963]
Learner	-0.109	0.124	-0.0897	0.0963
	(0.0699)	(0.0729)	(0.0729)	(0.0448)
	[0.265]	[0.265]	[0.265]	[0.147]
Software verification	0.120	0.0376	-0.110	0.0431
	(0.0535)	(0.0557)	(0.0557)	(0.0343)
	[0.118]	[0.515]	[0.170]	[0.398]
North America	**0.132***	-0.117	-0.00667	-0.0416
	(0.0587)	(0.0612)	(0.0612)	(0.0376)
	[0.077]	[0.145]	[0.899]	[0.458]
Other location	-0.0743	-0.0147	-0.0140	-0.0759
	(0.0697)	(0.0727)	(0.0726)	(0.0447)
	[0.629]	[0.976]	[0.976]	[0.250]
Additional controls	Yes	Yes	Yes	Yes
Mean of dependent variable	0.608	0.516	0.360	0.091
Observations	372	372	372	372

Note: Each column corresponds to a regression. Standard errors are in parentheses, Romano-Wolf adjusted p-values in brackets. Stars represent different levels of significance (*** p<0.01, ** p<0.05, * p<0.1) according to Romano-Wolf adjusted p-values. Independent variables are described in §2.3.1 and additional controls in §2.3.2. Mean of dependent variable corresponds to the mean of the outcome of each column.

We relate the probability to have used one of four IDEs for Coq (Emacs, CoqIDE, VS Code and jsCoq) to our previously defined categories. The user interface, a.k.a. IDE (Integrated Development Environment), is a fundamental part of proof assistants such as Coq, as it is the interaction medium between users and the system.

Our survey included many IDE questions, meant to get a better idea of their user base, their limitations, and to be able to inform stakeholders. As mentioned in Section 3.1, we already shared these specific results with stakeholders by opening issues in relevant projects.

In Table 2, we look at the responses to the question "Which editors or IDEs have you used for Coq?" and relate the results for four of the most used IDEs to our previously defined categories of respondents. Among the four selected IDEs, the two with the most users according to the survey results are Emacs and CoqIDE (used respectively by 61% and 52% of respondents). This is explained by observing that they were the only two available

options for many years. Recently, more focus has been put on developing more modern user interfaces, e.g., based on the very popular Visual Studio Code (VS Code, used by 36% of respondents), or directly in the browser, with jsCoq (used by 9% of respondents).

Unfortunately, most of the significant results obtained in this table were not robust to Romano-Wolf corrections for multiple hypothesis testing. The only two robust results are:

1. Users with more than 2 years of experience are more likely to have used CoqIDE. This can be interpreted by CoqIDE being the most standard user interface for Coq, since it is distributed along the system, and the question being formulated as "Which editors or IDEs have you used for Coq?" and not "Which editors or IDEs are you *currently using* for Coq?". Thus, users are very likely to have used CoqIDE at some point when they are experienced enough.
2. Users in North America are 13 pp more likely to have used Emacs, compared to the Europe baseline, which may be explained by cultural differences or IDE penetration.

We do not get any other statistically significant results, although some are not very far from being significant. In particular, the learner category looks like it could be more frequently using jsCoq, but it is very difficult to draw reliable conclusions given the small number of jsCoq users among our respondents (40).

5 Satisfaction and Needs of Different Population Groups

In this section, we look at how our different population groups perceive Coq and what needs they express. This should inform researchers willing to improve proof assistants for specific use cases or target audiences.

We base our analysis on the responses to the satisfaction questions for the six IDEs that we included additional questions about, as well as one question on the importance of various improvements to be possibly made to Coq, one question asking what additional extraction targets respondents would like to have, and one question asking what additional languages to support in the documentation. Figure 3 presents the descriptive results to the question about improvements to be possibly made to Coq.

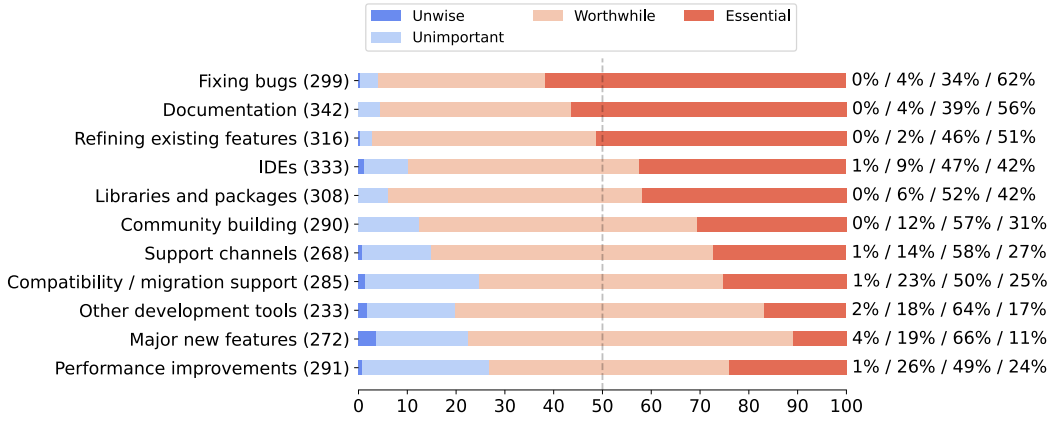


Figure 3 Results to the question "In order to make you more productive in Coq and to encourage others to learn and use Coq, how important are improvements in the following areas (relative to their current state)?". On the right hand-side, the percentages correspond, in order, to the proportion of "Unwise", "Unimportant", "Worthwhile", and "Essential" answers. Improvement categories are ordered according to the majority judgment methodology [14].

Table 3 Expressed satisfaction and needs.

VARIABLES	(1) IDE satisfaction	(2) Essential technical vs community improvements	(3) New extraction targets	(4) Other languages
> 2 years of experience	0.116	0.105	-0.0154	-0.0483
	(0.105)	(0.185)	(0.0510)	(0.0406)
	[0.632]	[0.826]	[0.826]	[0.632]
> 5 years of experience	**-0.236***	**0.484****	-0.00963	-0.00944
	(0.108)	(0.191)	(0.0526)	(0.0419)
	[0.085]	[0.047]	[0.976]	[0.976]
Learner	-0.224	-0.0625	0.0519	-0.00920
	(0.112)	(0.199)	(0.0548)	(0.0436)
	[0.159]	[0.936]	[0.704]	[0.936]
Software verification	0.0429	0.187	**0.163*****	0.0340
	(0.0856)	(0.152)	(0.0419)	(0.0333)
	[0.624]	[0.545]	[0.002]	[0.559]
North America	0.0721	-0.286	-0.00620	0.000884
	(0.0942)	(0.167)	(0.0460)	(0.0366)
	[0.779]	[0.265]	[0.982]	[0.982]
Other locations	0.00952	-0.219	0.0872	**0.122****
	(0.113)	(0.198)	(0.0546)	(0.0434)
	[0.923]	[0.415]	[0.304]	[0.038]
Additional controls	Yes	Yes	Yes	Yes
Mean of dependent variable	3.049	0.325	0.153	0.089
Observations	367	372	372	372

Note: Each column corresponds to a regression. Standard errors are in parentheses, Romano-Wolf adjusted p-values in brackets. Stars represent different levels of significance (*** p<0.01, ** p<0.05, * p<0.1) according to Romano-Wolf adjusted p-values. Independent variables are described in §2.3.1 and additional controls in §2.3.2. Mean of dependent variable corresponds to the mean of the outcome of each column.

Table 3 contains the following outcomes:

IDE satisfaction A natural number variable defined as the maximum IDE satisfaction level (from 0 to 4) for each respondent. We say that the IDEs each respondent assigned a maximum satisfaction level to are their favorites. To be defined, this outcome needs respondents to have answered at least one IDE satisfaction question, which is why we have 5 fewer observations for this outcome. The average satisfaction level given to the favorite IDE is 3, which corresponds to "Satisfied".

Essential technical vs community improvements An integer number variable defined as the difference between the number of essential technical improvements and the number of essential community improvements according to each respondent. We separate Coq improvements (from the question of Figure 3) into two categories: technical improvements (fixing bugs, refining features, IDEs, development tools, new features, performance) and community improvements (documentation, libraries, community building, support channels). We exclude the "Compatibility / migration support" improvement because it could be perceived as belonging to any of the two categories depending on how "support" is interpreted. We count for each of these categories the number of essential improvements, and we take the difference. Thus, a respondent with a positive value for this variable is listing more technical improvements as essential, and a respondent with a negative value for this variable is listing more community improvements as essential. On average,

respondents listed 0.3 additional essential technical improvements compared to community improvements, but note that the list of technical improvements contained two more items than the list of community improvements.

New extraction targets A binary variable for whether respondents provided a non-empty answer to the question on new extraction targets. 15% of the respondents asked for new extraction targets using this open text question.

Other languages A binary variable for whether respondents provided a non-empty answer to the question on languages to support in the documentation. 9% of our respondents asked for documentation in languages other than English using this open text question.

Experience level

Respondents with more than 5 years of experience are significantly more likely to assign a lower satisfaction score (0.24 lower on average) to their favorite IDE. They also list more essential technical improvements than community improvements, compared to other respondents (a 0.48 average difference).

We can explain experienced respondents being less satisfied with their IDE and more adamant about technical improvements by them having learned limitations of their IDEs or other technical limitations during their years of experience.

Other results

The other two statistically significant differences (robust to multiple hypothesis testing) are:

1. Respondents using Coq for software verification are more likely to request new extraction targets (an additional 0.16 pp, which is of the same magnitude as the mean of the variable over our population). Once again, this is not surprising as extraction is particularly suited to software verification, but it is limited by the restricted number of programming languages being currently officially supported as extraction targets.
2. Respondents in other locations (outside Europe and North America) are more likely to request support for new languages in the documentation (an additional 0.12 pp, which is largely superior to the mean of the variable over our population).

6 Threats to Validity

Several characteristics of our analysis represent a possible threat to validity. In terms of external validity (i.e., the validity of applying the conclusions of our study outside the specific context), there are two major concerns. First, it is possible that the Coq community behaves differently from other ITP communities. This can be due both to technical reasons (e.g., different proof assistants may be best suited to different use cases) and to sociological reasons ("birds of a feather flock together"), as could be observed, e.g., with the success of Lean among mathematicians. Second, even within the Coq community, the people who responded to the survey did not form a random sample of the community. It is likely that survey respondents had characteristics different from the average Coq user. In particular, and given the length of the survey, only people with enough time and motivation completed the survey in full. The results then concern this subsample of users and may not be extrapolated to the whole community. We took measures to avoid selection bias due to the use of the English language, but by only making the survey available in a second language (Chinese), and not many more, we are likely to have missed potential respondents in other linguistic communities. We tried to reach Coq users broadly by advertising the survey on many forums and mailing

lists, including language-specific ones, on social networks, and on the Coq website, but each choice of platform necessarily introduced bias in the sample. There is no way to completely avoid that beyond trying to reach even more broadly. For instance, we tried to reach learners as well by asking teachers to advertise the survey to their students, but we do not know how many did so. Our analyses of how different categories of respondents use Coq differently or express different needs reduce the issues of having a biased sample by including many controls, but we very likely missed some important ones.

Beyond external validity, there are other factors that can lead to misleading results. For example, the relatively low number of respondents may reduce the precision of the estimates, increase confidence intervals and reduce the power of the analysis (i.e., the ability to find a statistically significant effect when it exists). The order in which questions were asked may have affected the answers, which is a well-known phenomenon [18]. It was to mitigate this kind of issue that we decided, for example, to place demographics questions last. Respondents may have misunderstood some questions or questions options. We took many measures to make questions as clear as possible (using simple English, collaboratively writing and proofreading questions, using beta testers), but, despite that, we became aware after launching the survey that a few questions were still ambiguous, or did not have the originally intended meaning.

7 Discussion and Conclusions

We presented the design, deployment, and summaries and analyses of the responses to the Coq Community Survey 2022. Besides providing up-to-date descriptive data on the Coq community and decision support for the Coq Team, we hope our work can be useful for constructing future surveys targeted at ITP communities. During analysis of tentative future surveys, changes could be tracked for recurring questions and results of our regression analyses, e.g., on users performing software verification, could be replicated.

As part of the supplementary material to our paper [12], we provide 1) the survey definition in LimeSurvey and HTML format, 2) the Jupyter Notebook and Stata code we used to analyze the response data, 3) plots of answers to all closed questions and plots of some interactions between questions, 4) manual analysis of some open-text questions, and 5) answers to open-text questions. In particular, the code in our supplementary material shows how we generated the plots in this paper. However, for GDPR conformance, we do not include the raw response data that could fully replicate plots and other analyses.

Planning, running, and analyzing the survey was work intensive. The WG decided to announce a retention period of one year for the raw survey response data up front. However, this turned out to be a limiting decision in terms of the WG's time and resources. Keeping raw data is a delicate matter, since respondents and their answers may sometimes be identified by correlating survey response data with data from other sources. Nevertheless, we believe asking respondents for permission to store their data for more than one year is warranted, due to the time required for rigorous response analysis.

An important problem with recurring surveys is responder retention, as highlighted by the significant drop in the number of responses to the OCaml community survey between 2020 and 2022 [21]. Three years or more may be needed between surveys in an ITP community to ensure there will be enough responses to repeat previous analyses in a reliable way. In a small community, if filling in a survey becomes perceived as time-consuming routine work of little value, many members may not volunteer for the task. Making most or all questions optional does not address this issue.

The survey included some open-ended questions, whose responses we do not discuss in this paper. Future surveys could code the answers to these questions and include new multiple-choice questions. Since our analyses indicate that experience has a large impact on Coq user behavior, future research could investigate in more detail how users gain experience, and the effectiveness of their methods to improve. Another avenue is to look in detail at methodological problems that users face when applying Coq in their domain, and how users develop workarounds and solutions to these problems.

References

1 Ana de Almeida Borges, Jean-Rémy Falleri, Jim Fehrle, Emilio Jesús Gallego Arias, Érik Martin-Dorel, Karl Palmskog, Alexander Serebrenik, and Théo Zimmermann. Coq Community Survey 2022: Summary of Results, abstract. The Coq Workshop 2022, August 2022. URL: https://coq-workshop.gitlab.io/2022/abstracts/Coq2022-04-01-community-survey.pdf.

2 Ana de Almeida Borges, Jean-Rémy Falleri, Jim Fehrle, Emilio Jesús Gallego Arias, Érik Martin-Dorel, Karl Palmskog, Alexander Serebrenik, and Théo Zimmermann. Coq Community Survey 2022: Summary of Results, slides. The Coq Workshop 2022, August 2022. URL: https://coq-workshop.gitlab.io/2022/slides/Coq2022-04-01-community-survey.pdf.

3 Andrew W. Appel. Coq's vibrant ecosystem for verification engineering (invited talk). In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, pages 2–11, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3497775.3503951.

4 Jasmin Christian Blanchette. Formalizing the metatheory of logical calculi and automatic provers in Isabelle/HOL (invited talk). In *International Conference on Certified Programs and Proofs*, CPP 2019, pages 1–13, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293880.3294087.

5 Thomas Braibant. Coq survey, 2014. URL: https://github.com/braibant/coq-survey/blob/master/popl-coq.pdf.

6 Damian Clarke, Joseph P. Romano, and Michael Wolf. The Romano–Wolf multiple-hypothesis correction in Stata. *The Stata Journal*, 20(4):812–843, 2020. doi:10.1177/1536867X20976314.

7 The Coq Survey Working Group. Coq Community Survey 2022 in Chinese, 2022. URL: https://thzimmer.gitlabpages.inria.fr/coq-survey-2022-assets/LimeSurvey/questionnaire_chinese.html.

8 The Coq Survey Working Group. Coq Community Survey 2022 in English, 2022. URL: https://thzimmer.gitlabpages.inria.fr/coq-survey-2022-assets/LimeSurvey/questionnaire_356388_en.html.

9 The Coq Survey Working Group. Coq Community Survey 2022 Results: Part I (blog post), 2022. (Who is using Coq and in what context?). URL: https://coq.discourse.group/t/coq-community-survey-2022-results-part-i/1730.

10 The Coq Survey Working Group. Coq Community Survey 2022 Results: Part II (blog post), 2022. (How people are using Coq? — OS, IDEs, CI/CD). URL: https://coq.discourse.group/t/coq-community-survey-2022-results-part-ii/1746.

11 The Coq Survey Working Group. Coq Community Survey 2022 Results: Part III (blog post), 2022. (How is Coq used? — features, tools, libraries). URL: https://coq.discourse.group/t/coq-community-survey-2022-results-part-iii/1777.

12 Ana de Almeida Borges, Annalí Casanueva Artís, Jean-Rémy Falleri, Emilio Jesús Gallego Arias, Érik Martin-Dorel, Karl Palmskog, Alexander Serebrenik, and Théo Zimmermann. Supplementary material for the article "Lessons for Interactive Theorem Proving Researchers from a Survey of Coq Users", May 2023. doi:10.5281/zenodo.7930567.

13 Paul Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, first edition, 2007.

14 Adrien Fabre. Tie-breaking the highest median: alternatives to the majority judgment. *Social Choice and Welfare*, 56(1):101–124, 2021.

15 Taylor Fausak. State of Haskell survey, 2022. URL: https://taylor.fausak.me/2022/11/18/haskell-survey-results/.

16 Sacha Greif and Eric Burel. State of JS, 2022. URL: https://2022.stateofjs.com/en-US/.

17 Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 6(2):65–70, 1979. URL: http://www.jstor.org/stable/4615733.

18 Glenn D Israel and CL Taylor. Can response order bias evaluations? *Evaluation and Program Planning*, 13(4):365–371, 1990.

19 Mark L. Mitchell and Janina M. Jolley. *Resarch Design Explained*. Wadsworth, Cengage Learning, 7th edition, 2010.

20 OCaml Software Foundation. OCaml user survey, 2020. URL: https://discuss.ocaml.org/t/suggestions-from-the-ocaml-survey-result/6791.

21 OCaml Software Foundation. OCaml users survey, 2022. URL: https://ocaml-sf.org/docs/2022/ocaml-user-survey-2022.pdf.

22 Stephen O'Grady. The RedMonk programming language rankings: June 2022, October 2022. URL: https://redmonk.com/sogrady/2022/10/20/language-rankings-6-22/.

23 opam development team. OCaml package manager, 2023. URL: https://opam.ocaml.org.

24 Karl Palmskog, Enrico Tassi, and Théo Zimmermann. Reliably reproducing machine-checked proofs with the Coq Platform. In *Workshop on Reproducibility and Replication of Research Results*, 2022. URL: https://arxiv.org/abs/2203.09835.

25 Benjamin C. Pierce, Chris Casinghino, Michael Greenberg, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2011. Japanese translation. URL: http://proofcafe.org/sf/.

26 Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*, volume 1 of *Software Foundations*. Electronic textbook, 2022. Chinese translation, version 5.7. URL: https://coq-zh.github.io/SF-zh/lf-current/index.html.

27 Joseph P Romano and Michael Wolf. Stepwise multiple testing as formalized data snooping. *Econometrica*, 73(4):1237–1282, 2005.

28 Stack Overflow. Stack Overflow developer survey, 2022. URL: https://survey.stackoverflow.co/2022/.

29 The Coq Development Team. opam archive for Coq, 2023. URL: https://github.com/coq/opam-coq-archive.

30 Mairieli Wessel, Alexander Serebrenik, Igor Wiese, Igor Steinmacher, and Marco A. Gerosa. What to expect from code review bots on GitHub? a survey with OSS maintainers. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, SBES '20, pages 457–462, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3422392.3422459.

31 James R. Wilcox. Why is the Coq logo made to look like a penis?, April 2021. URL: https://sympa.inria.fr/sympa/arc/coq-club/2021-04/msg00006.html.

32 Jeffrey M Wooldridge. *Introductory econometrics: A modern approach*. Cengage learning, 6th edition, 2015.

Formalizing Norm Extensions and Applications to Number Theory

María Inés de Frutos-Fernández
Imperial College London, UK
Universidad Autónoma de Madrid, Spain

Abstract

The field $\mathbb{R}$ of real numbers is obtained from the rational numbers $\mathbb{Q}$ by taking the completion with respect to the usual absolute value. We then define the complex numbers $\mathbb{C}$ as an algebraic closure of $\mathbb{R}$. The p-adic analogue of the real numbers is the field $\mathbb{Q}_p$ of p-adic numbers, obtained by completing $\mathbb{Q}$ with respect to the p-adic norm. In this paper, we formalize in Lean 3 the definition of the p-adic analogue of the complex numbers, which is the field $\mathbb{C}_p$ of p-adic complex numbers, a field extension of $\mathbb{Q}_p$ which is both algebraically closed and complete with respect to the extension of the p-adic norm.

More generally, given a field K complete with respect to a nonarchimedean real-valued norm, and an algebraic field extension L/K, we show that there is a unique norm on L extending the given norm on K, with an explicit description.

Building on the definition of $\mathbb{C}_p$, we formalize the definition of the Fontaine period ring B_{HT} and discuss some applications to the theory of Galois representations and to p-adic Hodge theory.

The results formalized in this paper are a prerequisite to formalize Local Class Field Theory, which is a fundamental ingredient of the proof of Fermat's Last Theorem.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Type theory

Keywords and phrases formal mathematics, Lean, mathlib, algebraic number theory, p-adic analysis, Galois representations, p-adic Hodge theory

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.13

Supplementary Material *Software (Source code)*: `github.com/mariainesdff/norm_extensions_journal_submission`, archived at `swh:1:dir:01f6b345a06ece970e589d4bbc68ee8b9b2cf58a`

Funding
1) EPSRC Grant EP/V048724/1: Digitising the Langlands Program (UK).
2) This work has been supported by the Madrid Government (Comunidad de Madrid – Spain) under the multiannual Agreement with UAM in the line for the Excellence of the University Research Staff in the context of the V PRICIT (Regional Programme of Research and Technological Innovation)

Acknowledgements I would like to thank Kevin Buzzard for many helpful conversations during the completion of this project, Thomas Browning for formalizing normal closures, and Yaël Dillies for the discussions on how best to integrate seminorms in `mathlib`. I also thank the `mathlib` community and maintainers for their support and insightful suggestions during the development of this work.

1 Introduction

Recall that the real numbers $\mathbb{R}$ are defined as the completion of the field $\mathbb{Q}$ of rational numbers with respect to the usual absolute value, and the complex numbers $\mathbb{C}$ as an algebraic closure of $\mathbb{R}$. The field $\mathbb{C}$ is algebraically closed and complete with respect to the extension of the usual absolute value.

However, there are other absolute values that we could consider on the rational numbers. Namely, for any prime number p, there is an associated p-adic absolute value on $\mathbb{Q}$ and, if we complete $\mathbb{Q}$ with respect to this absolute value, we obtain the field $\mathbb{Q}_p$ of p-adic numbers. Based on this definition, we can regard $\mathbb{Q}_p$ as an analogue of the real numbers $\mathbb{R}$.


© María Inés de Frutos-Fernández;
licensed under Creative Commons License CC-BY 4.0

The next natural question is which field would be the p-adic analogue of the complex numbers $\mathbb{C}$. Following the previous reasoning, a first candidate would be an algebraic closure $\mathbb{Q}_p^{\text{alg}}$ of $\mathbb{Q}_p$. This field is algebraically closed, and we will see in Section 3 that it carries a well-defined extension of the p-adic absolute value. However, it turns out that $\mathbb{Q}_p^{\text{alg}}$ is not complete with respect to this absolute value.

Therefore, to get an analogue of $\mathbb{C}$, we need the extra step of taking the completion of $\mathbb{Q}_p^{\text{alg}}$ with respect to its absolute value, obtaining a new field $\mathbb{C}_p$, which is complete by construction and can be shown to be algebraically closed. We call $\mathbb{C}_p$ the field of p-adic complex numbers.

In general, given any field K with an absolute value (or more generally, a seminorm, see Section 2) and a field extension L/K, we can ask whether the absolute value on K can be extended to L and, if so, whether this extension is unique. The main results in this article are the formalization of the proofs of two theorems, the Extension Theorem and the Unique Extension Theorem, giving positive answers to these questions, under hypotheses made precise in Section 3.

The construction of the field $\mathbb{C}_p$ of complex numbers is only one of the motivations for this formalization project. A second one is that this more general theory of norm extensions plays an important role in the proofs of the main theorems of local class field theory [2].

Class field theory is a branch of number theory whose goal is to describe the Galois abelian extensions of a local or global field K, as well as their corresponding Galois groups, in terms of the arithmetic of the field K. Class field theory is a key ingredient in the proof of the Taniyama–Shimura–Weil conjecture, which is in turn required to prove Fermat's Last Theorem.

A third motivation is that having formalized the field $\mathbb{C}_p$ opens the door to formalizing the definitions of *Fontaine's period rings* [19, 12], a work that we initiate in this paper (see Section 5.2). These rings have many applications to Representation Theory and p-adic Hodge Theory. They can be used to detect interesting properties of Galois representations, and to prove comparison theorems between different cohomology theories. They are studied within the Langlands Program [21], one of the biggest research programs in modern mathematics, consisting on a vast family of conjectures that seek to establish deep relations between algebra and analysis.

The results formalized in this paper are necessary prerequisites for the formalization of the proof of Fermat's Last Theorem, as well as for formalizing statements from the Langlands Program. They join a growing family of results from number theory and related areas formalized in Lean, including the p-adic numbers [22], Dedekind domains and class groups [6, 5], idèle class groups [15], Witt vectors [13], Galois theory [8], the Krull topology [25], the Haar measure [31], and perfectoid spaces [9]. See also Section 6.2 for a discussion of related work in other proof assistants.

A repository containing the formalization described in this article is publicly available at the link `https://github.com/mariainesdff/norm_extensions_journal_submission/`. In the article's text, `file.lean` represents the file at `https://github.com/mariainesdff/norm_extensions_journal_submission/blob/master/src/file.lean`. Parts of the formalization have already been integrated in Lean's mathematical library, in which case we will refer the reader to the corresponding files. Some of the code excerpts included in the paper have been edited for clarity.

1.1 Lean and mathlib

This formalization was carried out in the Lean 3 interactive theorem prover [16], based on dependent type theory, with proof irrelevance and non-cumulative universes [11]. An introduction to the language can be found in [3].

Our project is built on top of Lean's mathematical library `mathlib` [24], which currently contains over one million lines of code formalized by almost 300 contributors. The key property of this library is its unified approach to integrate different areas of mathematics, including for example algebra, analysis and topology, all of which we needed for this project. Both in Lean's core library and in `mathlib`, type classes are used to represent mathematical structures on types [4].

We remark that while Lean 4 is already available, the vast majority of the mathematical prerequisites for this project have not been ported to Lean 4's mathematical library yet, so at the current time it is not feasible to use this version for our work. We expect to port the project to Lean 4 once all of its prerequisites have been ported.

1.2 Paper outline

In Section 2 we recall some background on norms and field extensions. Section 3 contains an overview of the proofs of the main results of this paper, concerning the unique extension of a nonarchimedean norm. In Section 4 we discuss some implementation details of our formalization, while in Section 5 we present some applications of our main theorem to the field of Number Theory. Finally, we conclude in Section 6 with a discussion of future work and a reflection on the work presented in this article.

2 Mathematical background

In this section, we define several kinds of seminorms and norms on additive groups and rings, and we recall some definitions from field theory.

Seminorms and norms

Let G be an additive group. A *seminorm* on G is a function $|\cdot| : G \to \mathbb{R}$ such that $|0| = 0$, $|-g| = |g|$ for all g in G, and $|\cdot|$ is subadditive, that is, $|g+h| \leq |g| + |h|$ for all g, h in G. A seminorm $|\cdot|$ such that $|g| = 0$ implies $g = 0$ is called a *norm*. An example of seminorm that is not a norm is the constant zero function.

```
structure add_group_seminorm (G : Type*) [add_group G] :=
(to_fun : G → ℝ)
(map_zero' : self.to_fun 0 = 0)
(add_le' : ∀ (g h : G), self.to_fun (g + h) ≤ self.to_fun g + self.to_fun h)
(neg' : ∀ (g : G), self.to_fun (-g) = self.to_fun g)

structure add_group_norm (G : Type*) [add_group G] extends add_group_seminorm G :=
(eq_zero_of_map_eq_zero' : ∀ g, to_fun g = 0 → g = 0)
```

We say that an additive group seminorm $|\cdot|$ is *nonarchimedean* if it satisfies the *strong triangle inequality*: $|g+h| \leq \max\{|g|, |h|\}$ for all g, h in G. Note that this is stronger than the usual triangle inequality $|g+h| \leq |g| + |h|$. Otherwise, we say that $|\cdot|$ is *archimedean*.

```
def is_nonarchimedean {G : Type*} [add_group G] (f : G → ℝ) : Prop :=
∀ g h, f (g + h) ≤ max (f g) (f h)
```

If R is a ring, then a *seminorm* on R is an additive group seminorm $|\cdot|$ on R that is also submultiplicative, that is, such that $|rs| \leq |r| \cdot |s|$ for all r, s in R. A seminorm $|\cdot|$ is said to be *power multiplicative* if $|r^n| = |r|^n$ $\forall r \in R, n \in \mathbb{N}_{\geq 1}$, and *multiplicative* if $|1| = 1$ and $|rs| = |r| \cdot |s|$ for all r, s in R.

```
structure mul_ring_seminorm (R : Type*) [non_assoc_ring R] extends
add_group_seminorm R, monoid_with_zero_hom R ℝ
```

As in the additive group case, a ring seminorm $|\cdot|$ is a *norm* if $|r| = 0$ implies $r = 0$. A ring norm is said to be *power-multiplicative* or *multiplicative* if it has the corresponding property when regarded as a seminorm.

In this article, all rings will be assumed to be commutative and to have a unit element. We will only consider ring seminorms satisfying the extra hypothesis $|1| \leq 1$. This implies that either $|1| = 1$ or $|1| = 0$, in which case $|\cdot|$ is the zero seminorm.

An example of multiplicative ring norm on the rational numbers $\mathbb{Q}$ is given by the usual absolute value. This norm is archimedean, and if we complete $\mathbb{Q}$ with respect to it, we get the field $\mathbb{R}$ of real numbers.

However, there are other norms that we can consider on the rational numbers, which are widely used in Number Theory. Namely, for every prime number p, we can define a *p-adic norm* as follows. Define a function $v_p : \mathbb{Z} \to \mathbb{Z}$ as $v_p(r) := \max\{n \in \mathbb{Z} \mid p^n \text{ divides } r\}$. The function v_p can be extended to $\mathbb{Q}$ by $v_p(\frac{r}{s}) = v_p(r) - v_p(s)$. We can then define the *p-adic norm* of $x \in \mathbb{Q}$ as $|x|_p := p^{-v_p(x)}$, and it is easy to check from its definition that $|\cdot|_p$ is a nonarchimedean multiplicative norm on $\mathbb{Q}$. When we complete $\mathbb{Q}$ with respect to the p-adic norm, we obtain the field $\mathbb{Q}_p$ of p-adic numbers.

Most of the definitions listed in this section have already been integrated in `mathlib` by the author, and can be found in the `mathlib` files `analysis/normed/group/seminorm.lean` and `analysis/normed/ring/seminorm.lean`.

We will be mainly interested in nonarchimedean (semi)norms, and in particular, the proof of the main theorem uses this property in a significant way. However, some intermediate results are true for arbitrary seminorms, and we have formalized them in that greater generality.

Algebra norms

Let R be a commutative ring with a (submultiplicative) norm $|\cdot|$ and let A be an R-module. An *R-module seminorm* on A is an additive group seminorm $\|\cdot\|$ on A such that $\|r \cdot a\| = |r| \cdot \|a\|$ for all $r \in R, a \in A$. This notion was already defined in mathlib, under the name `seminorm`:

```
structure seminorm (R : Type*) (A : Type*) [semi_normed_ring R] [add_group A]
  [has_smul R A] extends add_group_seminorm A :=
(smul' : ∀ (r : R) (a : A), to_fun (r · a) = ‖r‖ * to_fun a)
```

If moreover A is an R-algebra, we can define an *R-algebra norm* on A as a ring norm $\|\cdot\|$ on A such that $\|r \cdot a\| = |r| \cdot \|a\|$ for all $r \in R, a \in A$. This can be defined in Lean by extending the existing `seminorm` as follows:

```
structure algebra_norm (R : Type*) [semi_normed_comm_ring R] (A : Type*) [ring A]
  [algebra R A] extends seminorm R A, ring_norm A
```

Field extensions

We end this section by recalling that given two fields K and L, we say that L is an *extension* of K, denoted L/K, if there is a homomorphism of rings $K \to L$ (which is necessarily injective). The extension L/K is said to be *algebraic* if every element of L is a root of a

nonzero polynomial with coefficients in K. Given an extension L/K, L is a vector space over the field K, and we say that the extension is *finite-dimensional* if the dimension of L as a K-vector space is finite.

The following Lean code states that L/K is a finite-dimensional algebraic extension of fields (the ring homomorphism from K to L is part of the data of the instance variable `[algebra K L]`).

```
variables {K L : Type*} [field K] [field L] [algebra K L]
  (h_fin : finite_dimensional K L) (h_alg : algebra.is_algebraic K L)
```

An algebraic field extension L/K is *normal* if every irreducible polynomial in $K[X]$ that has a root in L splits into linear factors as a polynomial in $L[X]$. Given any algebraic extension L/K, there is always a minimal field extension N/L inside an algebraic closure of L such that the extension N/K is normal. The field N is unique up to isomorphism, and we call it the *normal closure* of the extension L/K. If the extension L/K is finite, then N/K is finite as well.

3 Extensions of nonarchimedean norms

Let K be a field with a nonarchimedean submultiplicative norm $|\cdot|$, and let L/K be an algebraic extension. We would like to know whether it is possible to extend the norm $|\cdot|$ to a norm $|\cdot|_L$ on the larger field L and, if so, whether this extension is unique. We will show in this section that both questions have a positive answer, under some conditions on the fields K and L and the starting norm $|\cdot|$. Moreover, we will provide an explicit description of the norm $|\cdot|_L$, which we will call the *spectral norm* induced by K. The main results we formalized are [7, Theorem 3.2.1/2 and Theorem 3.2.4/2].

3.1 The spectral norm

Let R be a ring with a nonarchimedean seminorm $|\cdot|$, and let $P := X^m + a_{m-1}X^{m-1} + \cdots + a_1X + a_0 \in R[X]$ be a monic polynomial of degree $m \geq 1$ with coefficients in R. The *spectral value* $\sigma(P)$ of P is defined as $\sigma(P) := \max_{0\leq i<m} |a_i|^{1/(m-i)}$. By convention, we say that the monic polynomial $P = 1$ of degree 0 has spectral value 0.

A first approach to formalize this definition would be to use the function `supr` to take the supremum of the values $|a_i|^{1/(m-i)}$ for i running over the terms of `fin P.nat_degree`, the subtype of natural numbers less than the degree of P. However, this would force us to treat the case $P = 1$ differently in some of the proofs, since the type `fin P.nat_degree` is empty in that case.

Instead, we use the following trick: given any polynomial $P \in R[X]$, we define a map `spectral_value_terms` : $\mathbb{N} \to \mathbb{R}$ sending $i \in \mathbb{N}$ to $|a_i|^{1/(m-i)}$ if i is less than the degree of P, or to 0 otherwise. Since every term $|a_i|^{1/(m-i)}$ is at least zero, taking the supremum of `spectral_value_terms` over all natural numbers returns the spectral value of P. Note that we do not ask that P is monic in our formalized definition, but if P is monic, both definitions agree.

```
variables {R : Type*} [semi_normed_ring R]
def spectral_value_terms (P : R[X]) : ℕ → ℝ := λ (n : ℕ),
  if n < P.nat_degree then ‖ P.coeff n ‖^(1/(P.nat_degree - n : ℝ)) else 0
def spectral_value (P : R[X]) : ℝ := supr (spectral_value_terms P)
```

We prove some of the basic properties of the spectral value, including:

1. The spectral value of a polynomial is always nonnegative.
2. The spectral value of the linear polynomial $X - r$ is equal to the seminorm $|r|$ of r.
3. For any $m \in \mathbb{N}$, the spectral value of X^m is equal to 0. Moreover, if the seminorm $|\cdot|$ is a norm, then these are the only polynomials having spectral value 0.

Now, let K be a field with a nonarchimedean submultiplicative norm $|\cdot|$, and let L/K be an algebraic extension. Then any $y \in L$ is a root of a monic polynomial with coefficients in K, and the *minimal polynomial* of y over K is the monic polynomial of lowest degree in $K[X]$ having y as a root.

The *spectral norm* $|\cdot|_{\text{sp}}$ on L is the function $|\cdot|_{\text{sp}} : L \to \mathbb{R}_{\geq 0}$ sending $y \in L$ to the spectral value of the minimal polynomial of y over K, which we will denote by $|y|_{\text{sp}}$.

```
variables {K : Type*} [normed_field K] {L : Type*} [field L] [algebra K L]
  (h_alg : algebra.is_algebraic K L)
def spectral_norm (y : L) : ℝ := spectral_value (minpoly K y)
```

The terminology "spectral norm" is justified by the fact, shown in the next subsection, that $|\cdot|_{\text{sp}}$ is an algebra norm on L. However, note that this is not at all obvious from the definition, and both proving that the spectral norm satisfies the triangle inequality and that it is a multiplicative function require some serious work.

3.2 Norm extension theorems

In this section, we formalize in Lean 3 the proofs of the main results of the paper: two theorems about existence and uniqueness of extensions of nonarchimedean norms to algebraic field extensions.

First, we have the Extension Theorem, which states that given any field K with a power-multiplicative nonarchimedean norm $|\cdot|$ and any algebraic field extension L/K, the spectral norm on L is a power-multiplicative nonarchimedean K-algebra norm on L extending the norm on K. The theorem also gives us information about how the spectral norm relates to the K-algebra automorphisms of L, and to other extensions of the norm to L:

▶ **Theorem 1** (Extension Theorem, [7, 3.2.1/2]). *Let K be a field with a nonarchimedean power-multiplicative norm $|\cdot|$, L/K an algebraic extension, and $G(L/K)$ the group of K-algebra automorphisms of L.*

- *The spectral norm $|\cdot|_{sp}$ on L is a nonarchimedean power-multiplicative K-algebra norm on L extending the norm $|\cdot|$ on K. All K-algebra isomorphisms of L are isometries with respect to the spectral norm $|\cdot|_{sp}$. Any nonarchimedean power-multiplicative K-algebra norm on L is bounded above by $|\cdot|_{sp}$.*
- *If the field extension L/K is finite and normal, then $|\cdot|_{sp}$ is the only nonarchimedean power-multiplicative K-algebra norm on L extending $|\cdot|$ for which all $g \in G(L/K)$ are isometries. If $|\cdot|'$ is a nonarchimedean power-multiplicative K-algebra norm on L extending $|\cdot|$, then $|x|_{sp} = \max_{g \in G(L/K)} |g(x)|'$ for all $x \in L$.*

If moreover K is complete with respect to a nonarchimedean multiplicative ring norm $|\cdot|$, then the spectral norm on L is the *unique* nonarchimedean multiplicative ring norm on L extending $|\cdot|$. This is called the Unique Extension Theorem.

▶ **Theorem 2** (Unique Extension Theorem, [7, 3.2.4/2]). *Let K be a field that is complete with respect to a nonarchimedean multiplicative norm $|\cdot|$ and let L/K be an algebraic extension. Then the spectral norm on L is the unique multiplicative nonarchimedean norm on L extending the norm $|\cdot|$ on K.*

We will now provide an overview of the proof of these theorems, referencing where to find the full details both in the literature and in our formalization. The general proof strategy is to perform a series of "smoothing steps" in which, starting from a given seminorm (or norm), we construct a new seminorm or norm having better properties.

The proof of Theorem 1 relies on the following lemma:

▶ **Lemma 3.** *Let K be a field with a nonarchimedean power-multiplicative norm $|\cdot|$. Each finite extension L/K has at least one nonarchimedean power-multiplicative K-algebra norm extending the norm $|\cdot|$.*

The statement of this lemma is formalized as follows, and its proof can be found in the file `normed_space.lean`.

```
lemma finite_extension_pow_mul_seminorm (hfd : finite_dimensional K L)
  (hna : is_nonarchimedean (norm : K → ℝ)) :
  ∃ (f : algebra_norm K L), is_pow_mul f ∧ function_extends (norm : K → ℝ) f ∧
  is_nonarchimedean f := ...
```

Proof of Lemma 3. Fix a basis $\{e_1 = 1, \dots, e_n\}$ of L as a K-vector space and define a function $\|\cdot\| : L \to \mathbb{R}$ by setting $\|\sum_{i=1}^n a_i e_i\| := \max_i |a_i|$. We can check that $\|\cdot\|$ is a nonarchimedean K-module norm on L extending the norm on K, and that there exists a positive real number c such that $\|xy\| \leq c\|x\|\|y\|$ for all $x, y \in L$.

In the file `normed_space.lean`, we let `basis.norm` be the norm associated to a basis of a finite-dimensional K-vector space as above, and prove that it has the desired properties; in particular, the existence of the bounding constant c.

```
def basis.norm {ι : Type*} [fintype ι] [nonempty ι] (B : basis ι K L) : L → ℝ :=
λ x, ‖B.equiv_fun x (classical.some (finite.exists_max (λ i : ι, ‖B.equiv_fun x i‖ )))‖
lemma basis.norm_is_bdd {ι : Type*} [fintype ι] [nonempty ι] {B : basis ι K L} {i : ι}
  (hBi : B i = (1 : L)) (hna : is_nonarchimedean (norm : K → ℝ)) :
  ∃ (c : ℝ) (hc : 0 < c), ∀ (x y : L), B.norm (x * y) ≤ c * B.norm x * B.norm y := ...
```

The first smoothing step is to use [7, Proposition 1.2.1/2] to conclude the existence of a K-algebra norm on L extending the norm $|\cdot|$ on K. We prove this proposition in the file `seminorm_from_bounded.lean`. That is, given a function $f : R \to \mathbb{R}$ from a commutative ring R to the real numbers, we define a function `seminorm_from_bounded'`: $R \to \mathbb{R}$ by sending $x \in R$ to $\sup\left\{\frac{f(x\cdot y)}{f(y)} | y \in R, f(y) \neq 0\right\}$, and then prove that this function is a nonarchimedean ring seminorm whenever f satisfies the required hypotheses.

Note that we do not need to make the condition $f(y) \neq 0$ explicit in the definition of `seminorm_from_bounded'`, since by Lean's convention $\frac{f(x\cdot y)}{f(y)}$ will be zero in that case, and we are only interested in this function when f is a function taking nonnegative values.

```
def seminorm_from_bounded' : R → ℝ := λ x, supr (λ (y : R), f(x*y)/f(y))
def seminorm_from_bounded (f_zero : f 0 = 0) (f_nonneg : ∀ (x : R), 0 ≤ f x)
  (f_mul : ∃ (c : ℝ) (hc : 0 < c), ∀ (x y : R), f (x * y) ≤ c * f x * f y)
  (f_add : ∀ a b, f (a + b) ≤ f a + f b) (f_neg : ∀ (x : R), f (-x) = f x) : ring_seminorm R :=
{ to_fun     := seminorm_from_bounded' f,
   map_zero' := seminorm_from_bounded_zero f_zero,
   add_le'   := seminorm_from_bounded_add f_nonneg f_mul f_add,
   mul_le'   := seminorm_from_bounded_mul f_nonneg f_mul,
   neg'      := seminorm_from_bounded_neg f_neg }
lemma seminorm_from_bounded_is_nonarchimedean (f_nonneg : ∀ (x : R), 0 ≤ f x)
  (f_mul : ∃ (c : ℝ) (hc : 0 < c), ∀ (x y : R), f (x * y) ≤ c * f x * f y)
  (hna : is_nonarchimedean f) : is_nonarchimedean (seminorm_from_bounded' f) :=
```

The proof concludes with a second smoothing step, following [7, Proposition 1.3.2/1], which allows us to construct a power multiplicative K-algebra norm on L extending $|\cdot|$. This proposition is formalized in the file `smoothing_seminorm.lean`. Given a real-valued function $f : R \to \mathbb{R}$ from a commutative ring R, we define a function `smoothing_seminorm_def` by sending $x \in R$ to the infimum $\inf_{n\geq 1} |x^n|^{1/n}$, which we show agrees with the limit of this sequence.

```
def smoothing_seminorm_def (x : R) : ℝ := infi (λ (n : pnat), (f(x^(n : ℕ)))^(1/(n : ℝ)))
lemma smoothing_seminorm_def_is_limit (hf1 : f 1 ≤ 1) (x : R) :
  tendsto (smoothing_seminorm_seq f x) at_top (𝓝 (smoothing_seminorm_def f x)) :=
```

We then prove that, whenever f is any ring seminorm on R, the corresponding function `smoothing_seminorm_def` is a power-multiplicative ring seminorm on R. We remark that this smoothing step uses the nonarchimedean nature of the norm $|\cdot|$ in a significant way: proving that `smoothing_seminorm_def` satisfies the strong triangle inequality requires a careful approximation argument relying on f being nonarchimedean. We are not aware of any alternative arguments to show that `smoothing_def` satisfies even the usual triangle inequality. ◀

Having proven Lemma 3, we can present the proofs of the two main theorems, whose formalizations can be found in the files `spectral_norm.lean` and `spectral_norm_unique.lean`.

Proof of the Extension Theorem. We want to show that the function $|\cdot|_{\mathrm{sp}} : L \to \mathbb{R}$ is a power-multiplicative K-algebra norm on L extending the norm on K. We first reduce to the case where the field extension L/K is finite and normal. We can do this because, to check that $|xy|_{\mathrm{sp}} \leq |x|_{\mathrm{sp}}|y|_{\mathrm{sp}}$, we can work on the normal closure of $K(x, y)$, and similarly for the other properties in the definition of power-multiplicative algebra norm. This reduction step just requires us to check that, whenever E is an intermediate field between K and L and x is an element of E, the spectral norm of x is the same whether we regard it as an element of the normal closure of E, or as an element of L:

```
lemma spectral_value.eq_normal (E : intermediate_field K L)
  (h_alg_L : algebra.is_algebraic K L) (x : E) :
  spectral_norm K (normal_closure K E (algebraic_closure E))
    (algebra_map E (normal_closure K E (algebraic_closure E)) x) =
  spectral_norm K L (algebra_map E L x) := ...
```

Since L/K is finite, by Lemma 3 there exists a power-multiplicative K-algebra norm $\|\cdot\|$ on L extending the norm $|\cdot|$ on K.

The next "smoothing step" is to define a function $|\cdot|_G : L \to \mathbb{R}$ that sends $y \in L$ to $|y|_G := |y|_{G(L/K)} := \max_{g\in G(L/K)} \|g(y)\|$. We prove in the file `alg_norm_of_galois.lean` that $|\cdot|_G$ (denoted `alg_norm_of_galois`) is a power-multiplicative K-algebra norm on L extending the norm on K, and that every automorphism $g \in G(L/K)$ is an isometry with respect to $|\cdot|_G$.

```
def alg_norm_of_galois (hna : is_nonarchimedean (norm : K → ℝ)) :
  algebra_norm K L :=
{ to_fun    := λ x, (supr (λ (σ : L ≃ₐ[K] L), alg_norm_of_auto h_fin hna σ x)),
  ... }
lemma alg_norm_of_galois_is_pow_mul (hna : is_nonarchimedean (norm : K → ℝ)) :
  is_pow_mul (alg_norm_of_galois h_fin hna) :=..
```

Since the extension L/K is normal, the minimal polynomial q_y of $y \in L$ is of the form $q_y = \prod (X - g(y))^{p^e}$, where the exponent e is a positive natural number depending on y. We can therefore use [7, Proposition 3.1.2/1(2)] to conclude that $|y|_{\mathrm{sp}} = |y|_G$.

```
lemma spectral_norm_eq_alg_norm_of_galois (h_alg : algebra.is_algebraic K L)
  (h_fin : finite_dimensional K L) (hn : normal K L)
  (hna : is_nonarchimedean (norm : K → ℝ)) :
  spectral_norm K L = alg_norm_of_galois h_fin hna := ...
```

Hence, we have shown that the spectral norm on L is a power-multiplicative K-algebra norm on L extending the norm on K, and that for any other such norm $\|\cdot\|$, we have $|y|_{\mathrm{sp}} = \max_{g \in G(L/K)} \|g(y)\|$ for all $y \in L$. ◀

Proof of the Unique Extension Theorem. We first show that the spectral norm $|\cdot|_{\mathrm{sp}}$ is the only power-multiplicative K-algebra norm on L extending $|\cdot|$. Suppose that $\|\cdot\|$ is another such norm. By [7, Prop. 3.1.5/1] , it suffices to check $\|\cdot\|$ and $|\cdot|_{\mathrm{sp}}$ are equivalent on each field extension of the form $K(y)$, for $y \in L$. This follows from the facts that K is complete and $K(y)$ is finite dimensional over K, and hence any two K-algebra norms on $K(y)$ will be equivalent.

```
theorem spectral_norm_unique' [complete_space K] {f : algebra_norm K L}
  (hf_pm : is_pow_mul f) (hna : is_nonarchimedean (norm : K → ℝ)) :
  f = spectral_alg_norm h_alg hna := ...
```

We point out an implementation detail of the proof of `spectral_norm.unique'`. In order to apply two existing `mathlib` lemmas about linear maps in this proof, respectively called `linear_map.continuous_of_finite_dimensional`, and `continuous_linear_map.is_bounded_linear_map` , we need to consider two different normed space structures on $K(y)$. However, we should not have two different `[normed_space K K(y)]` instances, since this would cause inference problems. To avoid this issue, we work with two copies of $K(y)$, each with their own normed space structure. We do this by defining a copy of $K(y)$ as `E := id K(y)`.

```
set E : Type* := id K(y) with hEdef
```

We use the identity map in this definition so that Lean is not able to infer a normed space structure on $K(y)$ from that on `E`. This allows us to put a different normed space structure on each of the copies.

```
letI N1 : normed_space K E := ...,
letI N2 : normed_space K K(y) :=...
```

To conclude the proof, we need to check that the spectral norm $|\cdot|_{\mathrm{sp}}$ on L is multiplicative, which requires a last "smoothing step". By [7, Proposition 1.3.2/2], for any $y \in L$, there exists a power-multiplicative K-algebra norm $|\cdot|_y$ on L such that y is multiplicative for $|\cdot|_y$, meaning that $|xy|_y = |x|_y|y|_y$ for all $x \in L$. This seminorm is defined by sending $x \in L$ to the limit of $\frac{f(xy^n)}{(f(y))^n}$ as n tends to infinity, and can be found at `seminorm_from_const.lean`

```
def seminorm_from_const_seq (x : L) : ℕ → ℝ := λ n, (f (x * y^n))/((f y)^n)
def seminorm_from_const (x : L) : ℝ := classical.some
  (real.tendsto_of_is_bounded_antitone  (seminorm_from_const_is_bounded c f x)
  (seminorm_from_const_seq_antitone hf1 hc hpm x))
```

Since we have just shown that the spectral norm is the unique power-multiplicative K-algebra norm on L that extends $|\cdot|$, we can conclude that $|\cdot|_{\mathrm{sp}} = |\cdot|_y$. Therefore every y is multiplicative for $|\cdot|_{\mathrm{sp}}$, that is, the spectral norm is multiplicative.

```
lemma spectral_norm_is_mul [complete_space K]
  (hna : is_nonarchimedean (norm : K → ℝ)) (x y : L) :
  spectral_alg_norm h_alg hna (x * y) =
    spectral_alg_norm h_alg hna x * spectral_alg_norm h_alg hna y := ...
```

◀

The main reference we followed in our formalization, [7], is a book on nonarchimedean analysis, in which all results are stated exclusively for nonarchimedean (semi)norms. However, we would like to remark that the second smoothing step in the proof of Lemma 3 is the only part in the proofs of Theorems 1 and 2 in which the nonarchimedean property is necessary. By contrast, all of the remaining smoothing steps remain true for possibly nonarchimedean seminorms (noting that the extension of the norm will only be nonarchimedean if the starting norm has this property), and have been formalized in that generality.

We conclude this section with a concrete example: the extension of the p-adic norm on $\mathbb{Q}_p$ to its algebraic closure $\mathbb{Q}_p^{alg}$.

```
variables (p : ℕ) [fact (nat.prime p)]
@[reducible] def Q_p_alg : Type* := algebraic_closure ℚ_[p]
lemma Q_p_alg.is_algebraic : algebra.is_algebraic ℚ_[p] (Q_p_alg p) :=
algebraic_closure.is_algebraic _
```

By Theorems 1 and 2, the spectral norm is the unique (nonarchimedean) norm on the field $\mathbb{Q}_p^{alg}$ extending the p-adic norm.

```
instance normed_field : normed_field (Q_p_alg p) :=
@spectral_norm_to_normed_field ℚ_[p] _ _ _ _ padic.complete_space
  (Q_p_alg.is_algebraic p) padic_norm_e.nonarchimedean
lemma Q_p_alg.is_nonarchimedean : is_nonarchimedean (norm : (Q_p_alg p) → ℝ) :=
spectral_norm_is_nonarchimedean (Q_p_alg.is_algebraic p)
  padic_norm_e.nonarchimedean
lemma Q_p_alg.norm_extends (x : ℚ_[p]) : ‖ (x : Q_p_alg p) ‖ = ‖ x ‖ :=
spectral_alg_norm_extends (Q_p_alg.is_algebraic p) _ padic_norm_e.nonarchimedean
```

4 Implementation of norms and valuations

4.1 Unbundling seminorms

In `mathlib`, there are several classes to represent algebraic objects with a preferred norm that makes the object into a metric space. For example, a `normed_ring` R is a ring endowed with a submultiplicative norm, which is used to define a metric space structure on R:

```
class normed_ring (R : Type u) : Type u :=
(to_has_norm : has_norm R)
(to_ring : ring R)
(to_metric_space : metric_space R)
(dist_eq : ∀ (x y : R), has_dist.dist x y = ‖x - y‖)
(norm_mul : ∀ (a b : R), ‖a * b‖ ≤ ‖a‖ * ‖b‖)
```

Other related classes are `semi_normed_ring`, `normed_field`, `normed_add_group`, etc. These classes are very useful to prove analytic results, provided that one only needs to consider one fixed norm in the algebraic object (group, ring, etc) being studied.

However, there are situations in which one needs to consider several seminorms on the same object. For example, the proof strategy for Theorems 1 and 2 consisted on, starting from a given seminorm on the field L, constructing a few other seminorms on L having increasingly better properties.

The existing classes are not well-suited for working with several seminorms on the same object. The problem is that they bundle together the algebraic and topological structures of the object. For example, the above definition "`normed_ring`" includes a field "`to_ring`" that encodes the ring structure on R. If we were to put two `normed_ring` instances on R, this would in particular yield two distinct `ring` instances on R, which is not what we want - we want to consider two different norms on R, without varying the ring structure.

One could work around this problem by making multiple copies of the ring, as we did in the proof of Theorem 2. In that particular case, we decided on this approach because it allowed us to reuse some existing topological lemmas in `mathlib`.

However, we consider that a better general approach for simultaneously working with several norms on a ring R, which we follow in the rest of the paper, is to use unbundled versions of seminorms and norms. That is, instead of using a `normed_ring` class that bundles together the ring structure on R, its norm, and the resulting metric space structure, we work over a ring R and we define the `ring_norm` as a function from R to the real numbers satisfying the required hypotheses.

```
structure ring_norm (R : Type u) [ring R] : Type u :=
(to_fun : R → ℝ)
(map_zero' : to_fun 0 = 0)
(add_le' : ∀ (r s : R), to_fun (r + s) ≤ to_fun r + to_fun s)
(neg' : ∀ (r : R), to_fun (-r) = to_fun r)
(mul_le' : ∀ (x y : R), to_fun (x * y) ≤ to_fun x * to_fun y)
(eq_zero_of_map_eq_zero' : ∀ (x : R), to_fun x = 0 → x = 0)
```

4.2 Relating norms and valuations

A *valuation* v on a ring R is a multiplicative map $v : R \to \Gamma_0$ to a linearly ordered commutative monoid with zero Γ_0 that preserves zero and one and satisfies the strong triangle inequality $v(x + y) \leq \max v(x), v(y)$ for all $x, y \in R$.

```
structure valuation (R : Type u) (Γ₀ : Type v)
  [linear_ordered_comm_monoid_with_zero Γ₀] [ring R] : Type (max u v) :=
(to_fun : R → Γ₀)
(map_zero' : to_fun 0 = 0)
(map_one' : to_fun 1 = 1)
(map_mul' : ∀ (x y : R), to_fun (x * y) = to_fun x * to_fun y)
(map_add_le_max' : ∀ (x y : R), to_fun (x + y) ≤ max (to_fun x) (to_fun y))
```

We say that a valuation v has *rank one* if it is nontrivial and there exists an injective morphism of linear ordered groups with zero $\Gamma_0 \to \mathbb{R}_{\geq 0}$.

```
variables {R : Type*} [ring R] {Γ₀ : Type*} [linear_ordered_comm_group_with_zero Γ₀]
class is_rank_one (v : valuation R Γ₀) :=
(hom : Γ₀ →*₀ ℝ≥0)
(strict_mono : strict_mono hom)
(nontrivial : ∃ r : R, v r ≠ 0 ∧ v r ≠ 1)
```

It is easy to see from these definitions that nontrivial nonarchimedean norms correspond to rank one valuations and, in practice, these terms are often used interchangeably in the mathematical literature. However, the formalization of these two notions in the library `mathlib` does not provide a way to relate them.

In the file `normed_valued.lean`, we formalize a dictionary between nonarchimedean norms and rank one valuations on a field L. This is a powerful tool, since it allows us to obtain full access to all of the theorems about these notions available in `mathlib`. Note that there are plenty of formalized results about normed fields and normed spaces, developed by analysts, as well as a very complete theory of valuations, mainly formalized as part of the perfectoid space project [9]. Without a way to convert between norms and valuations, we would be forced to make a choice about which of these results were available to us.

We relate the two definitions as follows. First, given a normed field K for which the norm is nonarchimedean, this norm is automatically a valuation on K (note that we need to use `nnnorm`, the version of the norm taking values on the type $\mathbb{R}_{\geq 0}$ of nonnegative reals).

We then define a function `normed_field.to_valued` that endows K with a valued field structure. To do this, we need to provide a proof that the uniform space structure on K induced by this valuation agrees with the one induced by the normed field structure.

```
variables {K : Type*} [hK : normed_field K]
include hK
def valuation_from_norm (h : is_nonarchimedean (norm : K → ℝ)) : valuation K ℝ≥0 :=
{ to_fun   := nnnorm,
  ... }
def normed_field.to_valued (h : is_nonarchimedean (norm : K → ℝ)) : valued K ℝ≥0 :=
{ v := valuation_from_norm h,
  is_topological_valuation := ...,
  ..hK.to_uniform_space,
  ..non_unital_normed_ring.to_normed_add_comm_group }
```

Conversely, if we start with a field L with a valuation v and a proof `hv` that v is of rank one, then we can show that the function $L \to \mathbb{R}$ sending x to the image of $v(x)$ under the homomorphism `hv.hom` is a nonarchimedean norm on L, and we can endow L with the corresponding normed field structure.

Note that the default constructor of the class `normed_field` does not asks us to provide a uniform space structure on L; instead, it defines this uniform space structure as the one induced by the norm. However, doing this would lead Lean to think that we have two different uniform space structures on L, since we already had the uniform space structure induced by the valuation. We therefore indicate explicitly that the uniform space structure we are considering is the one coming from the valuation, and once again prove that this agrees with the one induced by the norm.

```
variables {L : Type*} [hL : field L] {Γ₀ : Type*} [linear_ordered_comm_group_with_zero Γ₀]
  [val : valued L Γ₀] [hv : is_rank_one val.v]
include hL val hv
def norm_def : L → ℝ := λ x : L, hv.hom (valued.v x)
def valued_field.to_normed_field : normed_field L :=
{ norm              := norm_def,
  dist              := λ x y, norm_def (x - y),
  to_uniform_space := valued.to_uniform_space,
  uniformity_dist  := sorry,
  ..., }
```

5 Applications to number theory

5.1 The p-adic complex numbers

As we recalled in Section 2, the real numbers $\mathbb{R}$ are constructed as the completion of the rational numbers $\mathbb{Q}$ with respect to the usual absolute value. We can then define the complex numbers $\mathbb{C}$ as an algebraic closure of $\mathbb{R}$. The field $\mathbb{C}$ is algebraically closed and complete with respect to the extension of the usual absolute value.

If we take the completion of $\mathbb{Q}$ with respect to the p-adic norm associated to a prime number p, we obtain the field $\mathbb{Q}_p$ of p-adic numbers, which we can regard as an analogue of the real numbers $\mathbb{R}$.

We would also like to find a p-adic analogue of the complex numbers $\mathbb{C}$. Our first guess would be to consider an algebraic closure $\mathbb{Q}_p^{\text{alg}}$ of $\mathbb{Q}_p$. However, although $\mathbb{Q}_p^{\text{alg}}$ is algebraically closed and, as shown in Section 3, the p-adic norm extends uniquely to $\mathbb{Q}_p^{\text{alg}}$, it turns out that $\mathbb{Q}_p^{\text{alg}}$ is not complete with respect to the p-adic norm.

By completing $\mathbb{Q}_p^{\text{alg}}$ with respect to this norm, we obtain a new field $\mathbb{C}_p$, which is by construction complete with respect to the p-adic norm, and can be shown to be algebraically closed. Hence $\mathbb{C}_p$ can be regarded as a p-adic analogue of the complex numbers.

To formalize the definition of $\mathbb{C}_p$, we start from the definition of $\mathbb{Q}_p^{\text{alg}}$. At the end of Section 3.2, we saw that `Q_p_alg p` is a normed field, whose norm is the spectral norm extending the p-adic norm on $\mathbb{Q}_p$. We take advantage of our norm-valuation dictionary from Section 4.2 to show that `Q_p_alg p` is a valued field, and define `C_p p` as the uniform space completion of `Q_p_alg p`. We then introduce the notation $\mathbb{C}$`_[p]` for `C_p p`, which is consistent with the notation $\mathbb{Q}$`_[p]` used in `mathlib` for the p-adic numbers.

```
instance Q_p_alg.valued_field : valued (Q_p_alg p) ℝ≥0 :=
normed_field.to_valued (Q_p_alg.is_nonarchimedean p)
def C_p := uniform_space.completion (Q_p_alg p)
notation `ℂ_[`p`]` := C_p p
```

An alternative, mathematically equivalent approach would have been to define $\mathbb{C}$`_[p]` as the Cauchy completion of `Q_p_alg p`, which would not require to introduce the instance `Q_p_alg.valued_field`. However, then we would have had to prove that $\mathbb{C}$`_[p]` is a normed field whose norm extends that of `Q_p_alg p`.

On the other hand, by defining $\mathbb{C}$`_[p]` as the uniform space completion of the valued field `Q_p_alg p`, we get access to existing results in `mathlib` that allow us to immediately conclude that $\mathbb{C}$`_[p]` is a valued field, whose valuation extends the valuation on `Q_p_alg p`. Therefore, this is a concrete example in which our norm-valuation dictionary has allowed us to gain access to lemmas that would otherwise not have been available.

```
instance : field ℂ_[p] := uniform_space.completion.field
instance C_p.valued_field : valued (ℂ_[p]) ℝ≥0 := valued.valued_completion
instance : has_coe_t (Q_p_alg p) ℂ_[p] := uniform_space.completion.has_coe_t _
lemma C_p.valuation_extends (x : Q_p_alg p) : valued.v (x : ℂ_[p]) = valued.v x :=
valued.extension_extends _
```

Now that we have a `valued_field` instance on $\mathbb{C}$`_[p]`, we just need to show that its valuation has rank one (which is easy, since for example the element p has valuation $1/p$, different from 0 and 1) to gain access to the associated `normed_field` instance on $\mathbb{C}$`_[p]`.

```
instance : is_rank_one (C_p.valued_field p).v := ...
instance : normed_field ℂ_[p] := valued_field.to_normed_field
```

Having the above results, it is now easy to conclude that the norm on `ℂ_[p]` extends the norm on `Q_p_alg p`, and that it is nonarchimedean. All of the results in this section can be found in the file `Cp_def.lean`.

```
lemma C_p.norm_extends (x : Q_p_alg p) : ‖ (x : ℂ_[p]) ‖ = ‖ x ‖ := ...
lemma C_p.is_nonarchimedean : is_nonarchimedean (norm : ℂ_[p] → ℝ) := ...
```

5.2 Fontaine's period rings

Let K be a p-adic field (a finite extension of $\mathbb{Q}_p$) and let $G_K := \mathrm{Gal}(K^{\mathrm{alg}}/K)$ be the *absolute Galois group* of K, that is, the group of K-algebra automorphisms of an algebraic closure K^{alg} of K. A p-adic *Galois representation* is a continuous group homomorphism $\rho : G_K \to \mathrm{GL}(V)$, where V is a finite dimensional $\mathbb{Q}_p$-vector space.

Galois representations are a fundamental object of study in number theory. A precise understanding of how they relate to other mathematical objects (such as elliptic curves and modular forms) was a key ingredient in the proof of Fermat's Last Theory, and remains an active area of research within the Langlands Program, an ambitious collection of conjectures that seek to establish deep relations between seemingly distant areas of mathematics.

Of special interest are those Galois representation that "come from geometry', meaning that the vector space V is a subquotient of the étale cohomology group of an algebraic variety. A famous conjecture by Fontaine and Mazur predicts sufficient conditions for when a Galois representation comes from geometry in this sense.

Fontaine"s strategy was to construct period rings, which are rings that can detect interesting properties of Galois representations. More precisely, a *Fontaine period ring* is a topological $\mathbb{Q}_p$-algebra B with a continuous linear action of G_K, with some compatible additional structures (such as a Frobenius map or a filtration), such that the subring B^{G_K} is a field of points of B invariant under the Galois action is a field, and such that the B^{G_K}-vector space $D_B(V) = (B \otimes_{\mathbb{Q}_p} V)^{G_K}$ is an interesting invariant of the Galois representation V. Given a period ring B, a Galois representation V is called *B-admissible* if $\dim_{B^{G_K}} D_B(V) = \dim_{\mathbb{Q}_p} V$.

For different choices of B, being B-admissible is equivalent to the representation having a certain arithmetic property. We have formalized in `Fontaine_period_rings.lean` the definitions of the following period rings:

1. $B = K^{\mathrm{alg}}$. A Galois representation V is K^{alg}-admissible if the action of G_K on V factors through a finite quotient.

   ```
   def K_alg {K : Type*} [field K] [algebra ℚ_[p] K]
     (h_fin : finite_dimensional ℚ_[p] K) := algebraic_closure K
   ```

2. $B = \mathbb{C}_p$. A Galois representation V is $\mathbb{C}_p$-admissible if the action of the inertia subgroup I_K of G_K on V factors through a finite quotient. See Section 5.1 for the formalization.
3. $B = B_{\mathrm{HT}} := \mathbb{C}_p[X, X^{-1}]$. A Galois representation V is said to be B_{HT}-admissible, or *Hodge-Tate*, if the vector space $\mathbb{C}_p \otimes_{\mathbb{Q}_p} V$ can be decomposed as a product of the form $\mathbb{C}_p(\chi_{\mathrm{cycl}}^{n_1}) \oplus \cdots \oplus \mathbb{C}_p(\chi_{\mathrm{cycl}}^{n_d})$ for some $n_i \in \mathbb{Z}$, where χ_{cycl} denotes the cyclotomic character.

   ```
   def B_HT := laurent_polynomial ℂ_[p]
   ```

   Besides being used to detect interesting properties of Galois representations, Fontaine's period rings are also prominently used in comparison theorems between different cohomology theories.

6 Discussion

6.1 Future Work

The first future goals related to this project would consist on formalizing some well-known properties of the fields $\mathbb{Q}_p^{\text{alg}}$ and $\mathbb{C}_p$. For example, we could show that $\mathbb{C}_p$ is algebraically closed and that $\mathbb{Q}_p^{\text{alg}}$ is not complete with respect to its norm. Certain generalizations of these facts are proven in [7, Proposition 3.4.1/3] and [7, Lemma 3.4.3/1], respectively.

Similarly, it is possible to build on our formalization to prove Hensel's Lemma [7, Proposition 3.3.4/3] and Krasner's Lemma [7, Corollary 3.4.2/2], two fundamental results in p-adic analysis. We remark that there is an existing formalization of Hensel's lemma in `mathlib`, but only for the p-adic numbers; the version we propose would generalize it.

A slightly more ambitious goal is to formalize the definition of the Fontaine period ring B_{dR}. We propose the following strategy to formalize this definition. First, let E be the pre-tilt of $\mathbb{C}_p$, that is, the limit $E := \varprojlim_{x \mapsto x^p} \mathcal{O}_{\mathbb{C}_p}/(p)$, where $\mathcal{O}_{\mathbb{C}_p}$ is the ring of integers of $\mathbb{C}_p$. Let $A_{\text{inf}} := W(E)$ be the ring of Witt vectors of E, and let $B_{\text{inf}}^+ := A_{\text{inf}}[\frac{1}{p}]$ be the localization of A_{inf} away from p.

```
def E := pre_tilt ℂ_[p] (C_p.valued_field p).v 𝒪_ℂ_[p] (valuation.integer.integers _) p
def A_inf := witt_vector p (E p)
def B_inf_plus := localization.away (p : A_inf p)
```

The missing part of the formalization consists on constructing a canonical surjective ring homomorphism $\theta : B_{\text{inf}}^+ \to \mathbb{C}_p$ (the `noncomputable!` tag is required to avoid a timeout, but we expect to be able to remove it when we provide the definition of `theta`):

```
noncomputable! def theta : ring_hom (B_inf_plus p) ℂ_[p] := sorry
lemma theta.surjective : function.surjective (theta p) := sorry
```

By general properties of Witt vectors, to construct this function $\theta : B_{\text{inf}}^+ \to \mathbb{C}_p$, it is enough to define the "sharp" map $\cdot^{\#} : E \to \mathcal{O}_{\mathbb{C}_p}$ sending $\xi := (\xi_0, \xi_1, \cdots) \in E$ to $\xi^{\#} := \lim_{n\to\infty} \hat{\xi}_n^{p^n}$, where each $\hat{\xi}_n$ is an arbitrary lifting of ξ_n to $\mathcal{O}_{\mathbb{C}_p}$, and to study some properties of this map.

Once the definition of `theta` is formalized, we will be able to define the ring B_{dR}^+ as the completion of B_{inf}^+ with respect to the ideal $\ker(\theta)$, and the field B_{dR} as the field of fractions of B_{dR}^+.

```
def B_dR_plus := uniform_space.completion (B_inf_plus p)
def B_dR := fraction_ring (B_dR_plus p)
```

Our final future goal is to formalize some basic properties of the Fontaine period rings B_{HT} and B_{dR}, as well as some of their applications to representation theory.

6.2 Related Work

This project requires to combine results from several mathematical areas, including analysis, field theory, number theory, and topology, and we found Lean's mathematical library `mathlib` to be the most complete library in terms of the required prerequisites. However, some of the background results we needed are also available in other proof assistants.

The p-adic numbers were first formalized by Pelayo, Voevodsky, and Warren [27], in the Coq UniMath library. They were formalized in Isabelle/HOL in 2022 [14].

Coq's Mathematical Components library [23] contains a formalization of Galois theory, developed as part of the odd order theorem project [20]. The proof that every field admits an algebraically closed extension was first formalized in the proof assistant Isabelle/HOL [17]. Field theory constructions such as algebraic extensions [30], algebraic closures [29], and minimal polynomials [28] have recently been added to the Mizar Mathematical Library.

The Isabelle/HOL standard library includes the theory of real normed spaces, much of which has been translated to the complex setting in an Isabelle Archive of Formal Proofs entry by Caballero and Unruh [10]. However, to the author's knowledge, the generalization to normed spaces over arbitrary fields is still missing. Similarly, real and complex normed spaces have been formalized in Mizar (see for instance [26] and [18]). In Coq, the MathComp-Analysis [1] extends the Mathematical Components library with topics in analysis, including in particular results about normed spaces.

6.3 Conclusion

We develop the theory of extensions of nonarchimedean norms to algebraic field extensions, and we build on this work to formalize the field $\mathbb{C}_p$ of p-adic complex numbers and some of Fontaine's period rings. This project fits in within the long-term goal of formalizing a complete proof of Fermat's Last Theorem in the general case. It is also a starting point for formalizing Galois representation theory and p-adic Hodge theory.

The formalization required about 5000 lines of code, of which about 1000 lines have been integrated in the `mathlib` library at the time of writing this article.

References

1 Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, and Kazuhiko Sakaguchi. Competing Inheritance Paths in Dependent Type Theory: A Case Study in Functional Analysis. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 3–20, Cham, 2020. Springer International Publishing.

2 Emil Artin and John Tate. *Class Field Theory*. W. A. Benjamin, New York, 1967.

3 Jeremy Avigad, Leonardo de Moura, and Soonho Kong. *Theorem Proving in Lean*. Carnegie Mellon University, 2021. Release 3.23.0. URL: `https://leanprover.github.io/theorem_proving_in_lean/`.

4 Anne Baanen. Use and Abuse of Instance Parameters in the Lean Mathematical Library. 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 4:1–4:20, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ITP.2022.4`.

5 Anne Baanen, Alex J. Best, Nirvana Coppola, and Sander R. Dahmen. Formalized class group computations and integral points on mordell elliptic curves. In Brigitte Pientka and Steve Zdancewic, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2023, pages 47–62, New York, NY, USA, 2023. Association for Computing Machinery. `doi:10.1145/3573105.3575682`.

6 Anne Baanen, Sander R. Dahmen, Ashvni Narayanan, and Filippo A. E. Nuccio Mortarino Majno di Capriglio. A Formalization of Dedekind Domains and Class Groups of Global Fields. 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 5:1–5:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ITP.2021.5`.

7 Siegfried Bosch, Ulrich Güntzer, and Reinhold Remmert. *Non-archimedean analysis : a systematic approach to rigid analytic geometry*. Springer-Verlag Berlin Heidelberg, 1984.

8 Thomas Browning and Patrick Lutz. Formalizing Galois Theory. *Experimental Mathematics*, 31(2):413–424, 2022. doi:10.1080/10586458.2021.1986176.

9 Kevin Buzzard, Johan Commelin, and Patrick Massot. Formalising Perfectoid Spaces. In Jasmin Blanchette and Cătălin Hriţcu, 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 299–312. ACM, 2020. doi:10.1145/3372885.3373830.

10 José Manuel Rodríguez Caballero and Dominique Unruh. Complex bounded operators. *Archive of Formal Proofs*, September 2021. Formal proof development. https://isa-afp.org/entries/Complex_Bounded_Operators.html.

11 Mario Carneiro. The Type Theory of Lean, 2019. Master thesis. https://github.com/digama0/lean-type-theory/releases/download/v1.0/main.pdf.

12 Xavier Caruso. *An introduction to p-adic period rings*, volume 54 of *Panoramas et Synthèses*, pages 19–92. Société Mathématique de France, 2019.

13 Johan Commelin and Robert Y. Lewis. Formalizing the Ring of Witt Vectors. In Cătălin Hriţcu and Andrei Popescu, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021, pages 264–277, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3437992.3439919.

14 Aaron Crighton. p-adic fields and p-adic semialgebraic sets. *Archive of Formal Proofs*, September 2022. Formal proof development. https://isa-afp.org/entries/Padic_Field.html.

15 María Inés de Frutos-Fernández. Formalizing the Ring of Adèles of a Global Field. 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 14:1–14:18, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ITP.2022.14.

16 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*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-21401-6_26.

17 Paulo Emílio de Vilhena and Lawrence C. Paulson. Algebraically Closed Fields in Isabelle/HOL. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 204–220, Cham, 2020. Springer International Publishing.

18 Noboru Endou. Complex Linear Space and Complex Normed Space. *Formalized Mathematics*, 12(2):93–102, 2004.

19 Jean-Marc Fontaine, editor. *Périodes p-adiques - Séminaire de Bures, 1988*, number 223 in Astérisque. Société mathématique de France, 1994.

20 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.

21 Robert P. Langlands. *Problems in the Theory of Automorphic Forms*, volume 170 of *Lecture Notes in Mathematics*, pages 18–61. Springer, Berlin, Heidelberg, 1970. doi:10.1007/BFb0079065.

22 Robert Y. Lewis. A Formal Proof of Hensel's Lemma over the p-Adic Integers. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, pages 15–26, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293880.3294089.

23 Assia Mahboub and Enrico Tassi. The Mathematical Components Libraries, 2017. URL: https://math-comp.github.io/mcb/.

24 The mathlib Community. The Lean Mathematical Library. In Jasmin Blanchette and Cătălin Hriţcu, editors, *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.

25 Sebastian Monnet. Formalising the Krull Topology in Lean, 2022. URL: https://arxiv.org/abs/2207.09486.

26 Kazuhisa Nakasho, Yuichi Futa, and Yasunari Shidama. Topological Properties of Real Normed Space. *Formalized Mathematics*, 22(3):209–223, 2014. doi:10.2478/forma-2014-0024.

27 Álvaro Pelayo, Vladimir Voevodsky, and Michael A. Warren. A univalent formalization of the p-adic numbers. *Mathematical Structures in Computer Science*, 25(5):1147–1171, 2015. doi:10.1017/S0960129514000541.

28 Christoph Schwarzweller. Ring and Field Adjunctions, Algebraic Elements and Minimal Polynomials. *Formalized Mathematics*, 28(3):251–261, 2020. doi:10.2478/forma-2020-0022.

29 Christoph Schwarzweller. Existence and Uniqueness of Algebraic Closures. *Formalized Mathematics*, 30(4):281–294, 2022. doi:10.2478/forma-2022-0022.

30 Christoph Schwarzweller and Agnieszka Rowińska-Schwarzweller. Algebraic Extensions. *Formalized Mathematics*, 29(1):39–47, 2021. doi:10.2478/forma-2021-0004.

31 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.

Tealeaves: Structured Monads for Generic First-Order Abstract Syntax Infrastructure

Lawrence Dunn
University of Pennsylvania, Philadelphia, PA, USA

Val Tannen
University of Pennsylvania, Philadelphia, PA, USA

Steve Zdancewic
University of Pennsylvania, Philadelphia, PA, USA

Abstract

Verifying the metatheory of a formal system in Coq involves a lot of tedious "infrastructural" reasoning about variable binders. We present Tealeaves, a generic framework for first-order representations of variable binding that can be used to develop this sort of infrastructure once and for all. Given a particular strategy for representing binders concretely, such as locally nameless or de Bruijn indices, Tealeaves allows developers to implement modules of generic infrastructure called *backends* that end users can simply instantiate to their own syntax. Our framework rests on a novel abstraction of first-order abstract syntax called a *decorated traversable monad* (DTM) whose equational theory provides reasoning principles that replace tedious induction on terms. To evaluate Tealeaves, we have implemented a multisorted locally nameless backend providing generic versions of the lemmas generated by LNgen. We discuss case studies where we instantiate this generic infrastructure to simply-typed and polymorphic lambda calculi, comparing our approach to other utilities.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Type theory

Keywords and phrases Coq, category theory, formal metatheory, raw syntax, locally nameless

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.14

Supplementary Material *Software (Source Code)*: `https://github.com/dunnl/tealeaves` archived at `swh:1:dir:213050814e9a7233b944f92c191df8eb360bb84b`

Acknowledgements We wish to thank the anonymous reviewers for their helpful comments and suggestions, and the authors of LNgen for describing their work and experience using LNgen.

1 Introduction

Computer-verified metatheory is increasingly critical for establishing trust in the design and implementation of formal systems [6], which we take to include formal logics, programming languages, query languages, lambda calculi, specification languages, and basically any system with a precise syntax. Formalizing metatheory in a general-purpose proof assistant like Coq requires a lot of tedious reasoning about variable binding. When performed manually, this typically involves the user proving a suite of "infrastructure" lemmas concerned with the properties of capture-avoiding substitution. In practice, if not in principle, this infrastructure is tightly coupled to the exact signature used to generate the syntax, owing to the prolific use of structural recursion and induction on terms. This dependency makes metatheory brittle, prevents reusability, hampers collaboration by users working on different systems, and can make syntax infrastructure more challenging to automate. This paper presents Tealeaves, a Coq framework for building extensible libraries of generic syntax infrastructure that users can *instantiate* to their own syntax, thus facilitating collaboration and reuse. Our framework rests on top of a principled category-theoretic abstraction of raw first-order abstract syntax, that of a *decorated traversable monad*, which we introduce in Section 3.


© Lawrence Dunn, Val Tannen, and Steve Zdancewic;
licensed under Creative Commons License CC-BY 4.0

A wide variety of syntax formalization strategies have been proposed in the literature, and a commensurate number of utilities have been designed to automate syntax infrastructure. Why, then, should we introduce yet another syntax metatheory framework? Generally, frameworks will differ in what sorts of syntax can be handled, what support is required from the proof assistant, and especially the cost of entry to the user. The main novelty of Tealeaves lies in the intersection of three features:

1. **Raw syntax** Tealeaves considers "raw" syntax that is extrinsically typed and scoped, in contrast to work on intrinsically well-scoped, well-typed syntax.
2. **Modular representations** Tealeaves' extensible design is agnostic about how binding is represented, admitting multiple *backend* modules that formalize syntax infrastructure for a particular first-order strategy, such as de Bruijn indices or locally nameless.
3. **Signature-generic** Tealeaves is based on the theory of decorated traversable monads, a set of equations independent of the signature of a particular language. Substitution lemmas proved in the form of reusable Tealeaves backends are proved once and for all, and do not rely on external code generators, which can be slow and fallible.

We briefly summarize some of the more salient points of these features. Sections 4 and 5 offer more detailed comparison to related work.

Extrinsically typed first-order abstract syntax is defined inductively. Notions such as well-scopedness and well-typedness, as well as lemmas about substitution and other operations, are defined *post factum* by structural recursion [10] on terms. This contrasts with intrinsically well-scoped, well-typed syntax [4], which uses the type system of the metatheory (in our case, Coq) to enforce constraints on the embedded syntax, blurring the line between operations' types and their correctness properties. The practical difficulties encountered by the two approaches differ, especially because the intrinsically typed workflow makes heavier use of dependently-typed programming that can be inconvenient to formalize in Coq. The raw approach contrasts with recent work [17] which is formalized in Agda and considers a different category-theoretic abstraction of intrinsically-typed syntax. Both of these styles also contrast with higher-order abstract syntax [26] and representations based on nominal sets [28], which are well-known to require adaptation for use in a general-purpose proof assistant like Coq (see Section 5).

Within the family of first-order approaches, a variety of binding representation strategies are available, with typical examples being de Bruijn indices [13] and locally nameless [11]. At its core, Tealeaves aims to be agnostic about how binding is represented concretely, accommodating multiple representations. Existing utilities for formalizing syntax in Coq, such as Autosubst [32, 36] and LNgen [5], target a specific representation (de Bruijn indices and locally nameless, respectively), and they prove signature-specific lemmas using synthesis or external code generators whereas our lemmas are signature-generic and formalized statically in Coq. Unlike those tools, our design supports variadic binders (i.e. those introducing a variable number of new entities) without modifying the core abstraction.

To achieve the modularity of Tealeaves, it is necessary to have an abstraction of (i.e. interface to) abstract syntax. For us this comes in the form of *decorated traversable monads* (DTMs). Definition 3.1 presents DTMs in terms of a highly expressive combinator `binddt` that we use to define a wide range of syntax-related operations. A previous proof-of-concept Coq framework, GMeta [23], also offers multiple representation strategies formalized generically over syntax, but it lacks a principled abstraction of syntax like DTMs, resorting to proofs by induction on a universe of representable types. One benefit of using DTMs is that the DTM composition law (Equation (3)) yields a fusion law for the composition of any two operations defined with Tealeaves.

To evaluate our framework, we have implemented a locally nameless backend module providing essentially the same infrastructure that a user would generate with LNgen, but whose lemmas are statically verified, generic, and proved using a principled equational theory rather than unverifiable, dynamically-generated proof scripts. We have used this backend to prove type soundness for the simply-typed and polymorphic lambda calculi. The latter especially demonstrates that our framework neatly handles heterogeneous substitution (e.g. substitution of types in terms). We discuss evaluation of Tealeaves in Section 4.

In sum, the contributions of this paper are threefold. (1) We introduce a principled abstraction of raw, first-order abstract syntax, decorated traversable monads, which provides an expressive equational framework for generic reasoning about substitution and related operations. (2) We implement the Coq library Tealeaves, an extensible and modular framework for generic reasoning about syntax, including syntax with many different kinds of variables. (3) We implement a locally nameless Tealeaves backend and use it to formalize progress and preservation lemmas for the two lambda calculi above, evaluating the practicality of our approach.

The rest of this document is laid out as follows. Section 2 explains how a Coq user incorporates Tealeaves into their formal metatheory workflow. Section 3 introduces DTMs and describes how they facilitate generic reasoning. Section 4 evaluates Tealeaves, including a description of our case studies and a feature-wise comparison to the utilities LNgen and Autosubst 1 and 2. Section 5 discusses other related work. Section 6 concludes with our future plans for Tealeaves, especially investigating extensions to the DTM concept.

2 Using Tealeaves

In this section, we examine how Tealeaves fits into the workflow of a formal semanticist working in Coq. As a running example, we consider a formalization of the untyped lambda calculus where variables are represented in the locally nameless style, though Tealeaves accommodates more sophisticated kinds of syntax (see Section 3.4) and can be extended to other representations of variables. It is up to the user what sorts of metatheory they want to develop about the calculus – Tealeaves only provides the syntax infrastructure. The details are unchanged if the user is interested in a typed system because substitution is defined on raw (untyped) terms.

The first-order[1] (or *initial algebraic*) representation of abstract syntax defines terms inductively in the form of *term algebras*. In the simplest case of a single sort of variables, one starts from a base set V of variables and constructs a set $T(V)$ of terms by closing the set under well-sorted applications of constructors. This construction justifies definitions by structural recursion on terms. Figure 1 shows a first-order definition of the syntax of the untyped lambda calculus, called `Lam`, as it would be defined by a user of Tealeaves. To keep the example simple, since we will consider locally nameless variables, the `Abs` case only needs to take the abstraction body as an argument and not a variable name.

Locally nameless is a hybrid strategy mixing Bruijn indices with named variables. A bound variable $n \in \mathbb{N}$ is a natural number that always refers to the n^{th} most recently introduced abstraction, indexing innermost to outermost from 0. A free variable is represented as an `atom`, an abstract type about which we assume only a decidable equality. Figure 2 shows a type `LN` of locally nameless variables as the sum of `nat` and `atom`; this definition is provided by our locally nameless backend. The type of raw lambda terms with locally nameless variables is then `Lam LN`.

[1] "First-order" here refers to the fact that the term constructors do not take Coq-level functions as

```
Inductive Lam (V : Type) : Type :=
  | Var : V -> Lam V
  | Ap  : Lam V -> Lam V -> Lam V
  | Abs : Lam V -> Lam V.
```

Figure 1 Syntax of the untyped lambda calculus.

```
Inductive LN : Type :=
  | Fr : atom -> LN (* free variables *)
  | Bd : nat -> LN. (* bound variables *)
```

Figure 2 The type of locally nameless variables.

As argued by Pollack [29, 30], the main advantage of locally nameless is that there is no possibility of variable capture during substitution and that α-equivalence of expressions coincides with syntactical equality, making this representation more practical in Coq formalizations than a fully named approach as with pen-and-paper. This convenience comes at a mild cost: some terms in `Lam LN` do not correspond to ordinary lambda terms modulo α-equivalence, owing to the possibility of a de Bruijn index n appearing under fewer than $n+1$ abstractions. Such an occurrence is neither free (because it is not an atom) nor bound, so locally nameless substitution lemmas tend to mention a *local closure* predicate ruling out these ill-formed occurrences. Only locally closed terms represent (α-equivalence classes of) ordinary lambda terms.

Without using Tealeaves, most users would not benefit from separating `Lam` and `LN` as shown; they would likely inline `LN` into the definition of `Lam`. We take this approach in Tealeaves mainly so we can exploit the fact that `Lam` is a decorated traversable monad later. Incidentally, this modularity could prove useful to the user who desires to consider more than one representation of variables in the same development, say because one is amenable to formalization in Coq and another is more convenient to program with. As future work, we hope to use Tealeaves to formalize a translation between named and locally nameless variables (see Section 6).

Workflow without Tealeaves

The inductive nature of `Lam` admits a notion of structural recursion on terms, which is used to define operations like capture-avoiding substitution. Our formalization of locally nameless employs five operations: opening one term by another, closing a term by an atom, substitution of a term for a free variable, a free-variable enumeration operation `FV`, and the local closure predicate.[2] The types of these operations are shown in Figure 3. Figure 4 shows an example of how one conventionally defines `FV`.

Users working without tool support must prove a suite of lemmas about these operations, some prototypical examples of which are included in Figure 3. For instance, `subst_fresh` posits that replacing occurrences of an atom `x` with expression `u` in a term `t` leaves `t` unchanged if `x` does not occur in `t`. Such lemmas are needed while developing metatheory about the lambda calculus. They are almost invariably proved by induction on terms.

---

arguments, even if the formal system is higher-order in some other sense.

[2] Local closure can also be given its own `Inductive` definition. In our unified treatment, we prefer to think of the predicate as another operation on terms which happens to return a proposition.

$$\begin{array}{lll} \texttt{open} & : & \texttt{Lam LN} \to \texttt{Lam LN} \to \texttt{Lam LN} \\ \texttt{subst} & : & \texttt{atom} \to \texttt{Lam LN} \to \texttt{Lam LN} \to \texttt{Lam LN} \\ \texttt{close} & : & \texttt{atom} \to \texttt{Lam LN} \to \texttt{Lam LN} \end{array} \qquad \begin{array}{lll} \texttt{FV} & : & \texttt{Lam LN} \to \texttt{list atom} \\ \texttt{LC} & : & \texttt{Lam LN} \to \texttt{Prop} \end{array}$$

$$\texttt{subst\_fresh} : \forall (\texttt{x} : \texttt{atom})(\texttt{u}\,\texttt{t} : \texttt{Lam LN}),\ \texttt{x} \notin \texttt{FV t} \implies \texttt{subst x u t} = \texttt{t}$$

$$\texttt{subst\_spec} : \forall (\texttt{x} : \texttt{atom})(\texttt{u}\,\texttt{t} : \texttt{Lam LN}),\ \texttt{subst x u t} = \texttt{open u (close x t)}$$

$$\texttt{fv\_subst\_upper} : \forall (\texttt{x} : \texttt{atom})(\texttt{u}\,\texttt{t} : \texttt{Lam LN}),\ \texttt{FV (subst x u t)} \subseteq (\texttt{FV t} \setminus \{\texttt{x}\}) \cup \texttt{FV u}$$

$$\texttt{open\_inj} : \forall (\texttt{x} : \texttt{atom})(\texttt{u}\,\texttt{t} : \texttt{Lam LN}),\ \texttt{x} \notin (\texttt{FV t} \cup \texttt{FV u}) \implies \texttt{open (Var (Fr x)) t} = \texttt{open (Var (Fr x)) u} \implies \texttt{t} = \texttt{u}$$

Figure 3 Locally nameless operations and some typical infrastructure lemmas.

```
Fixpoint FV (t : Lam LN) : list atom := match t with
  | Var (Fr x) => [x]
  | Var (Bd _) => []
  | Ap t1 t2 => FV t1 ++ FV t2
  | Abs body => FV body
  end.
```

Figure 4 Example definition in Coq of a recursively-defined function `FV`.

This workflow is inherently linearly-ordered as shown in Figure 5: syntax is defined, syntax infrastructure is implemented, then system-specific metatheory is formalized. "System-specific metatheory" can include properties like confluence of the lambda calculus, which is undoubtedly more interesting to the metatheorist than proving dozens of substitution lemmas.

One way a user could save labor is to use a tool like LNgen [5] to generate the infrastructure. LNgen accepts a grammar from the user in the form expected by Ott [33] and generates Coq modules containing lemmas and operations like those in Figure 3. The high-level workflow is unchanged, however: the infrastructure comes after the syntax is defined.

Whether it be implemented by hand or generated automatically, the syntax infrastructure for `Lam` represents a bottleneck in the user's workflow. It is a prerequisite for developing interesting metatheory, but it depends on the definition of `Lam`, so it cannot be formalized as a general-purpose library. The reason for the dependency is that functions defined by recursion (as well as proofs by induction) essentially follow the shape of `Lam`. For instance, adding a new constructor to `Lam` will break `FV` and `subst_fresh` until the user updates them to account for the new constructor or re-executes LNgen.

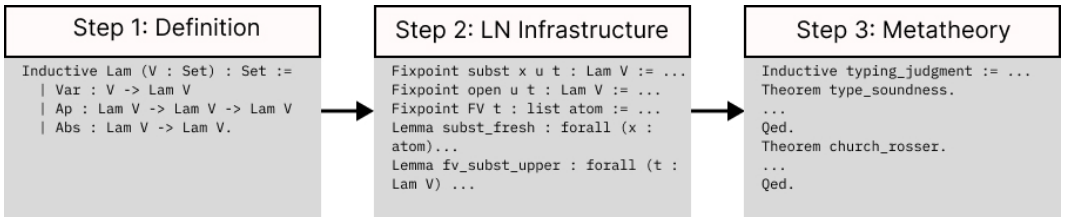


Figure 5 Basic workflow without Tealeaves.

Besides making the infrastructure brittle, this phenomenon implies a user who is formalizing a different syntax cannot reuse the infrastructure for `Lam`. This situation is all the more unfortunate when one realizes that most of the interesting reasoning of locally nameless does not really depend on `Lam` at all. The only interesting case in the proof of `subst_fresh`, for example, is `Var` – the `Ap` and `Abs` cases hold just by induction. Can we do better than this linear workflow?

2.1 The Tealeaves workflow

In a workflow incorporating Tealeaves, the user does not develop the locally nameless syntax infrastructure – we the Tealeaves developers have already implemented it in the form of a reusable Tealeaves *backend* module. Operations and lemmas like those in Figure 3 are provided by this backend, with a caveat: the formalization is generic in the sense that all references to `Lam` are replaced with references to a parameter $\texttt{T} : \texttt{Type} \to \texttt{Type}$. The user's obligation is to *instantiate* the backend to the choice $\texttt{T} = \texttt{Lam}$, which achieves essentially the same effect for the user as if they had constructed the infrastructure themselves. The user benefits as long as it is easier to perform this instantiation than to implement the infrastructure from scratch.

The cost of instantiating the backend is modest: the user must prove that `Lam` forms a *decorated traversable monad* (DTM), a principled category-theoretic concept which Tealeaves defines in the form of a typeclass [34]. All constructs implemented by the backend module are polymorphic over an instance of this typeclass; therefore they can automatically be specialized to any choice of `T`, such as `Lam`, for which a corresponding DTM instance has been registered with Coq's typeclass instance database.

The DTM instantiation process we describe below is for the simplest case when there is one grammatical category (`Lam`) and one sort of variable (arguments to `Var`). Section 3.4 indicates how we generalize this to more complex situations, such as a set of mutually-inductive grammatical categories involving multiple sorts of variables.

Supplying the DTM typeclass instance for `Lam` requires the user to define two operations. The first, which following standard Haskell terminology we call `return` (abbreviated `ret`), represents a coercion from variables to (atomic) terms of the user's syntax. For `Lam`, `ret` is exactly the `Var` constructor.

The more interesting operation is a higher-order function we call `binddt` (`bind` for a `d`ecorated `t`raversable monad). Conceptually, `binddt` acts like a template for defining (some) structurally recursive functions on `Lam`, including context-sensitive substitutions like `open` and "aggregation" operations like `FV`. The type of `binddt`, written in pseudo-Coq notation and specialized to `Lam`, is as follows:

$$\texttt{binddt}\ \texttt{`(Applicative F)}\ \ \texttt{(A B : Type)}\ \ :\ \ \texttt{(nat} \times \texttt{A} \to \texttt{F (Lam B))} \to \texttt{Lam A} \to \texttt{F (Lam B)}$$

We show the definition of `binddt` for `Lam` in Section 3.

Programmers with experience using monads may recognize the previous type as that of the usual `bind` operation extended with two features. First, just like the `traverse` operation of McBride and Paterson [25], the first argument to `binddt` is a choice of applicative functor $\texttt{F} : \texttt{Type} \to \texttt{Type}$. Second, observe that `nat` appears as an input of the function supplied to `binddt` – strictly speaking, one says that `Lam` is a traversable monad decorated *by* the natural numbers under addition. We discuss both of these features in Section 3. To recover the usual `bind` operation, one can instantiate `binddt` at the identity (applicative) functor and apply the projection $\texttt{nat} \times \texttt{A} \to \texttt{A}$.

Once the user defines `binddt`, they must supply a proof that it satisfies the axioms of decorated traversable monads, a set of four equations. These too shall be shown in Section 3.

Altogether, instantiating Tealeaves to `Lam` looks as follows. Note that `ret` and `binddt` are registered as instances of two operational typeclasses [35] called `Return` and `Binddt`. This is just a convenience allowing us to use the notation `ret` and `binddt` throughout Tealeaves and let Coq deduce which DTM instance is being referred to.

```
From Tealeaves Require Import Classes.Kleisli.DTM.
Fixpoint binddt_Lam `{Applicative F} (A B : Type)
    (f : nat * A -> F (Lam B)) (t : Lam A) : F (Lam B) := ...
Instance: Return Lam := Var.
Instance: Binddt nat Lam := binddt_Lam.
Instance: DecoratedTraversableMonad nat Lam.
(* Proofs of the equational axioms of DTMs... *)
Qed.
```

Having bundled all this up into a DTM typeclass instance, the user imports our locally nameless backend, `Tealeaves.Backends.LN`. This module defines all of the operations of locally nameless polymorphically over a choice `T` of DTM (specifically, `T` must be decorated by `nat`). It also supplies polymorphic lemmas. Using Coq's typeclass mechanism, the user can specialize these constructs to their own syntax. We show examples of this specialization below by explicitly passing `(T := Lam)` to each function, but in practice Coq can usually infer the choice of `T` implicitly. These commands will fail with an error message if Coq cannot locate an instance of the `DTM` typeclass for `Lam`.

```
From Tealeaves Require Import Backends.LN.
Check LN.open (T := Lam) : Lam LN -> Lam LN -> Lam LN.
Check LN.subst (T := Lam) : atom -> Lam LN -> Lam LN -> Lam LN.
Check LN.locally_closed (T := Lam) : Lam LN -> Prop.
Check LN.subst_fresh (T := Lam) : forall (t u : Lam LN) (x : atom),
    not (List.In x (FV t)) -> subst x u t = t.
Check LN.subst_spec (T := Lam) : forall (x : atom) (t u : Lam LN),
      subst x u t = open u (close x t).
```

Now that syntax infrastructure for `Lam` is available, the user can proceed with their ordinary workflow, which might consist of defining a type system and developing more interesting metatheory that is specific to the lambda calculus. When the properties of operations like substitution and opening are needed during a proof, the user invokes the corresponding lemma from the backend. The is akin to how one uses the modules that would be generated by LNgen, except no code generation has taken place.

Figure 6 shows a simplified architectural diagram of Tealeaves, which is broadly divided into three parts: the core formalization of DTMs and their properties, the locally nameless backend, and the effort required of the end user. We see again that the user's work logically divides into 3 steps as in Figure 5, but now the second step consists of proving the DTM instance for the user's syntax, while the locally nameless infrastructure is supplied by Tealeaves. We shall explain the rest of the architecture in Section 4.

An inherent limitation of implementing syntax infrastructure as a Coq library is that the backend can only prove a finite number of lemmas about substitution. For locally nameless, it is not entirely clear what constitutes a "complete" set of properties.[3] Suppose a user comes

[3] See Section 4 of [5] for a discussion of this issue in the context of LNgen, along with an informal argument for completeness of their generated lemmas, which our backend also proves.

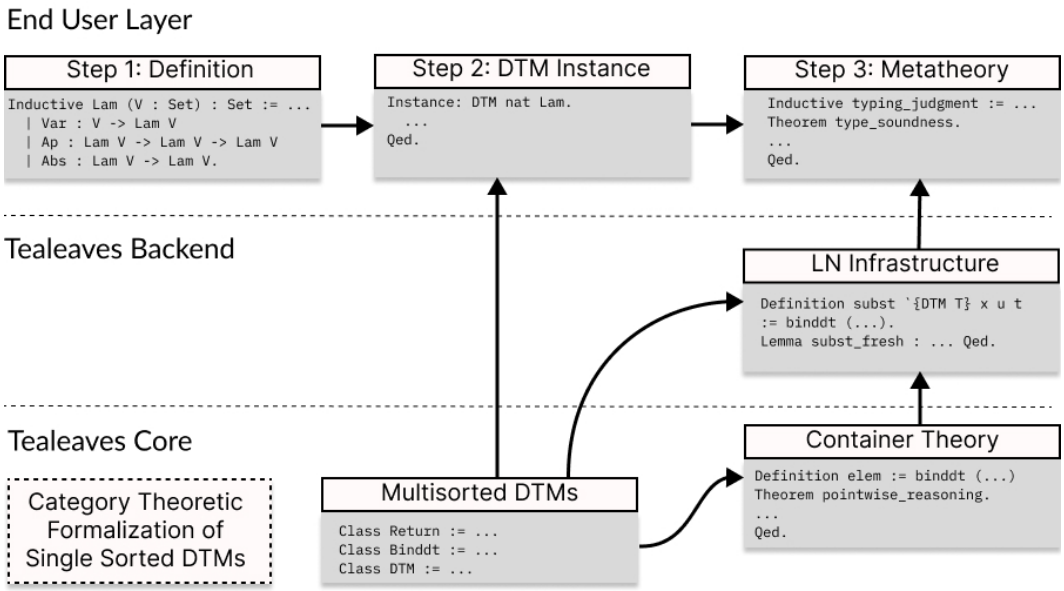


Figure 6 Simplified Tealeaves architecture and user workflow.

across a property of substitution that the backend does not prove. In this case, the user could state their lemma and prove it by induction on `Lam` like usual – Tealeaves does not impede the user's ordinary workflow.[4]

Alternatively, a Tealeaves power user could extend the locally nameless backend with their lemma. Of course, the backend module does not know about `Lam`, so a generic version of their lemma could not be proved by induction on lambda terms. Instead, the proof would have to be developed so as to apply to *any* DTM. Examples of such generic proofs are shown in Section 3.3. The advantage of extending the backend is that the user's effort would be reusable, even to users formalizing other kinds of syntax. This is how Tealeaves facilitates collaboration by users implementing different formal systems. Note that extending the backend with new lemmas requires no programming with Coq's tactic language Ltac [14] or any language outside of Coq itself, unlike the other utilities discussed in Section 4.

3 Decorated Traversable Monads

We now give a high-level intuition for DTMs and their equational theory, starting with the definition of `binddt` for `Lam`. While DTMs can be understood in terms of principled abstractions from monoidal category theory, users of Tealeaves are mostly shielded from this. In this section, we only assume some familiarity with functors and the use of monads to structural functional programs [38]. Like Haskell, we indicate the action of a functor `F` on morphisms as $\texttt{fmap}\,(\texttt{A}\;\texttt{B} : \texttt{Type}) : (\texttt{A} \to \texttt{B}) \to \texttt{F}\;\texttt{A} \to \texttt{F}\;\texttt{B}$. When it improves clarity, we use subscripts to indicate the implicit values of parametric arguments, and superscripts to disambiguate methods of typeclass instances, e.g. `fmap f` vs. $\texttt{fmap}^{\texttt{F}}_{\texttt{A},\texttt{B}}\;\texttt{f}$.

[4] The only possible issue is that unfolding the operations exported by the backend will reveal generic constructs that may be challenging to understand, a problem inherent to any generic framework. Future work on Tealeaves could supply custom simplification tactics to hide some of this complexity.

3.1 Proving the DTM instance for `Lam`

The first step required to instantiate Tealeaves to `Lam` is to define `binddt`, which can be thought of as a template for structurally recursive functions. We have seen that the first argument to `binddt` is a choice of applicative functor, an abstraction introduced by McBride and Paterson [25] and used often by functional programmers. For present purposes it suffices to know that an applicative functor `F : Type → Type` is characterized by two operations, `pure` and `ap`, whose types are as follows:

$$\begin{aligned} \texttt{pure} \quad & (\texttt{A} : \texttt{Type}) \quad : \quad \texttt{A} \to \texttt{F A} \\ \texttt{ap} \quad & (\texttt{A B} : \texttt{Type}) \quad : \quad \texttt{F} (\texttt{A} \to \texttt{B}) \to \texttt{F A} \to \texttt{F B} \end{aligned}$$

Like monads, applicative functors provide a notion of computational effect, but they are a more general abstraction. Intuitively, `pure` lifts a value into the functor by wrapping it in a trivial effect. `ap` applys effectful functions to effectful values, yielding an effectful value. These operations are subject to unsurprising laws given in [25], but they are not important here. The identity functor, written $\mathbf{I}$, is applicative; applicatives are also closed under composition. An *applicative homomorphism* $\phi : \texttt{F} \implies \texttt{G}$ is a natural transformation between applicative functors that commutes with `ap` and `pure` in the obvious way.

A prototypical applicative functor is the datatype `list` of finite lists, interpreted as the effect of non-determinism. `pure a` is the deterministic singleton `[a]`. `ap` applys lists of functions to lists of arguments to get a list of outputs by applying each function to each argument, representing a non-deterministic choice of both. A typical applicative homomorphism would be the transformation that maps a list to the set of its elements. Another important class of examples is given by a constant functor over any monoid, with `pure` and `ap` identified with the unit and multiplication, respectively.

The definition the user should give for `binddt` for `Lam` is as follows. Here, `<*>` is infix notation for `ap`. The helper function `preincr` will be explained below.

```
Fixpoint binddt `{Applicative F} {A B : Type}
  (f : nat * A -> F (Lam B)) : Lam A -> F (Lam B) := match t with
  | Var v => f (0, v)
  | Ap t1 t2 => pure Ap <*> binddt f t1 <*> binddt f t2
  | Abs body => pure Abs <*> binddt (preincr f 1) body
  end.
```

The first non-implicit argument `f` is a *substitution rule* that specifies what should happen at each variable. The role of `binddt` is to apply this substitution rule to each variable in a term. `f` itself takes two arguments. The first, here of type `nat`, represents the number of binders in scope at some variable occurrence, while the second represents the occurrence itself. When `binddt` is specialized to locally nameless case where $\texttt{A} = \texttt{B} = \texttt{LN}$ (recall Figure 2), the second argument to `f` will be either a de Bruijn index or an atom.

The output of `f` has type $\texttt{F}\,(\texttt{Lam B})$, representing an expression to replace the occurrence with, with the added flexibility that it may be wrapped in an applicative effect `F`. To account for this effect, the `Ap` and `Abs` cases of `binddt` lift the constructor into `F` with `pure` and replace ordinary function application with effectful application `<*>`. This pattern for working with applicative effects is common enough that there is an established notation of "idiom brackets" [25] (not shown here) to reduce the syntactic clutter.

We draw the reader's attention to the `Var` case: $\texttt{binddt f (Var v)} = \texttt{f}\,(0, \texttt{v})$. This definition is in fact an axiom of DTMs (Equation (1)) and corresponds to the fact that there are no binders in scope in an atomic expression. This may appear to suggest that `f` will only

ever see 0 binders in scope. Actually, `f` is informed about binders using a helper function `preincr` ("precompose increment"), whose definition is $(\texttt{preincr f n})\,(\texttt{n'}, \texttt{v}) = \texttt{f}(\texttt{n} + \texttt{n'}, \texttt{v})$. That `Abs` is a binder is reflected in the recursive call to `binddt`, which modifies `f` with `preincr`. The idea is that when `preincr f 1` is eventually applied to a binding context and a variable, it will increment its binding context before calling `f`. The reader should convince themselves that when the recursion of `binddt f t` bottoms out on a `Var`, the invocation of `f` will be of the form

$$\underbrace{\texttt{preincr}\big(\texttt{preincr}\big(\ldots\big(\texttt{preincr f 1}\big)\ldots\big)\ \texttt{1}\big)\ \texttt{1}}_{n \text{ times}}\,(0, \texttt{v}) = \texttt{f}\,(\texttt{n}, \texttt{v})$$

where $n \in \mathbb{N}$ is the number of `Lam` constructors gone under during recursion.

This scheme is quite general. For example, to extend the lambda calculus with a variadic `Let` construct accepting a list `l` of bound definitions, we can use `preincr f (length l)` to introduce several new entities at once. We can also use monoids other than `nat`. For example, a fully-named representation could use the monoid of finite lists of names under concatenation. The principle limiting what kinds of information one can pass to `f` using `preincr` is that `binddt` must satisfy Equation (3), below, a constraint we discuss further in Section 4.

The final step of instantiating the backend is to prove that `binddt` satisfies a set of four equations. The axioms are given by the following

▶ **Definition 3.1** (DTM, Kleisli-style presentation)**.** *A* traversable monad decorated by a monoid $\langle \texttt{W}, \cdot, 1_{\texttt{W}} \rangle$ *is a type constructor* $\texttt{T} : \texttt{Type} \to \texttt{Type}$ *equipped with operations:*

$$\begin{array}{rlll} \texttt{ret} & (\texttt{A} : \texttt{Type}) & : & \texttt{A} \to \texttt{T A} \\ \texttt{binddt} & (\texttt{Applicative F}) \quad (\texttt{A B} : \texttt{Type}) & : & (\texttt{W} \times \texttt{A} \to \texttt{F}\,(\texttt{T B})) \to \texttt{T A} \to \texttt{F}\ (\texttt{T B}) \end{array}$$

subject to the following four equations:

$$\texttt{binddt}_{\texttt{F}}\ \texttt{f} \circ \texttt{ret} = \texttt{f} \circ \texttt{ret}^{\texttt{W}\times} \tag{1}$$

$$\texttt{binddt}_{\texttt{I}}\ \big(\texttt{ret} \circ \texttt{extract}^{\texttt{W}\times}\big) = \texttt{id}_{\texttt{TA}} \tag{2}$$

$$\begin{aligned} &\texttt{fmap}^{\texttt{F}}\ (\texttt{binddt}_{\texttt{G}}\ \texttt{g}) \circ (\texttt{binddt}_{\texttt{F}}\ \texttt{f}) = \\ &\qquad \texttt{binddt}_{(\texttt{F} \circ \texttt{G})}\ \big(\lambda(\texttt{w}, \texttt{a}).\texttt{fmap}^{\texttt{F}}\,(\texttt{binddt}_{\texttt{G}}\ (\texttt{preincr g w}))\,(\texttt{f}\ (\texttt{w}, \texttt{a}))\big) \end{aligned} \tag{3}$$

$$\phi \circ \texttt{binddt}_{\texttt{F}}\ \texttt{f} = \texttt{binddt}_{\texttt{G}}\ (\phi \circ \texttt{f}) \quad (\textit{for all } \phi : \texttt{F} \implies \texttt{G}\ \textit{applicative hom.}) \tag{4}$$

In Equation (1), $\texttt{ret}^{\texttt{W}\times}$ is defined $\texttt{ret}^{\texttt{W}\times}\texttt{a} = (1_{\texttt{W}}, \texttt{a})$ where $1_{\texttt{W}}$ is the monoid unit. In (2), $\texttt{extract}^{\texttt{W}\times}$ is the projection $\texttt{extract}^{\texttt{W}\times}\,(\texttt{w}, \texttt{a}) = \texttt{a}$. These functions come from the Cartesian-product-with-monoid class of monads (such as used in 2.6 in [38]), known often as the "logging" or "writer" monad. Note that (2) instantiates `binddt` to the identity applicative, while (3) mentions the composition of two applicatives and (4) mentions homomorphisms between two applicatives.

The operations of the locally nameless backend are defined in terms of `binddt` and `ret`, while its lemmas are proved from these four equations only. Notably, this includes properties like `subst_fresh` (recall Figure 3), which is a *conditional* equality, unlike the axioms above. The next two sections show how the axioms of DTMs give rise to high-level properties like `subst_fresh`.

Category theory

Category theorists may wonder if we can give a more "theoretical" definition of DTMs. Law-abiding traversable functors were defined in [21]. Our library extends these to what we call *decorated*-traversable functors and proves the following characterization.

▶ **Theorem 3.2.** *Definition 3.1 is equivalent to a monoid in the category of decorated-traversable functors.*

This theorem is formalized for single-sorted DTMs (Definition 3.1) and a body of general-purpose category theory as shown in Figure 6. This part of Tealeaves is largely separate from the multisorted formalization described in the rest of this paper. A more thorough explanation of this useful perspective shall be forthcoming.

3.2 DTMs as containers

One often has occasion to consider the notion of variable *occurrence*, especially *occurrence in a binding context*. For example, `FV` lists occurrences of free variables, while local closure stipulates that no de Bruijn index $n \in \mathbb{N}$ *occurs under fewer than* $n+1$ *abstractions*. Both operations are more like "aggregations" than "substitutions." `FV` aggregates free variables into a list, while `LC` quantifies over all occurrences, aggregating a set of propositions (one for each occurrence) into a conjunction. It is not so obvious how an equational theory like that of DTMs can incorporate these collection-themed concepts. Tealeaves achieves this by building on a body of work on traversable [25, 18, 21, 8] and shapely [22] functors.

The way to define aggregations is to instantiate the applicative functor `F` to a (constant functor over some) monoid. The most general such choice is the free monoid, i.e. `list`. In particular, we can enumerate occurrences, including their context, as such:[5]

$$\texttt{tolistd} : \texttt{T LN} \to \texttt{list}\ (\mathbb{N} \times \texttt{LN}) \qquad \texttt{tolistd} \overset{def}{\equiv} \texttt{binddt}_{\texttt{list}\,(\mathbb{N}\times\texttt{LN})}\ (\lambda(\texttt{n}, \texttt{v}).\,[(\texttt{n}, \texttt{v})])$$

We also define a context-sensitive notion of variable occurrence ($\in_d$) as a special case of `binddt`. For a term `t` of `Lam LN`, $(\texttt{n}, \texttt{v}) \in_d \texttt{t}$ means a variable `v:LN` occurs somewhere in `t` underneath `n:nat` abstractions.

$$(\texttt{n}, \texttt{v}) \in_d \texttt{t} \overset{def}{\equiv} \texttt{binddt}_{\vee}\ (\lambda(\texttt{n}', \texttt{v}').(\texttt{n}, \texttt{v}) = (\texttt{n}', \texttt{v}'))\ \texttt{t}$$

Here, $\vee$ indicates we instantiate to the monoid of propositions under disjunction. We also provide a version $\texttt{v} \in \texttt{t}$ that checks for occurrences of `v` in any binding context.

Because variable occurrence is defined a special case of `binddt`, we immediately obtain a characterization of how `ret` and `binddt` interact with the occurrence relation.

▶ **Lemma 3.3.** *Equations* (1) *and* (3) *imply the following, respectively.*

$$(\texttt{n}, \texttt{v}_2) \in_d \texttt{ret}\ \texttt{v}_1 \iff \texttt{v}_1 = \texttt{v}_2 \wedge \texttt{n} = 0 \tag{5}$$

$$(\texttt{n}, \texttt{v}_2) \in_d \texttt{binddt}_{\texttt{I}}\ \texttt{f}\ \texttt{t} \iff$$
$$\exists\ \texttt{n}_1\ \texttt{n}_2\ \texttt{v}_1, (\texttt{n}_1, \texttt{v}_1) \in_d \texttt{t} \wedge (\texttt{n}_2, \texttt{v}_2) \in_d \texttt{f}\ (\texttt{n}_1, \texttt{v}_1) \wedge \texttt{n} = \texttt{n}_1 \texttt{+}\ \texttt{n}_2 \tag{6}$$

(5) states that the only variable in an atomic expression occurs with 0 binders in scope. (6) characterizes the set of occurrences in `t` after performing a substitution codified by `f`. If $\texttt{v}_1$ occurs in `t` under $\texttt{n}_1$ binders and $\texttt{v}_1$ is replaced by $\texttt{f}\ (\texttt{n}_1, \texttt{v}_1)$, the occurrences introduced by the subterm have $\texttt{n}_1$ added to their context, in addition to their context as occurrences in $\texttt{f}\ (\texttt{n}_1, \texttt{v}_1)$ – binding context accumulates with tree depth.

[5] Tealeaves generally names context-aware versions of operations with a trailing `d` for *decoration*.

$$\texttt{subst x u} = \texttt{bind}\ (\texttt{subst}_{\text{loc}}\ \texttt{x u}) \qquad \texttt{subst}_{\text{loc}}\ \texttt{x u v} = \begin{cases} \texttt{u} & \text{if } \texttt{v} = \texttt{Fr x} \\ \texttt{ret v} & \text{else} \end{cases}$$

$$\texttt{open u} = \texttt{bindd}\ (\texttt{open}_{\text{loc}}\ \texttt{u}) \qquad \texttt{open}_{\text{loc}}\ \texttt{u}\ (\texttt{n}, \texttt{v}) = \begin{cases} \texttt{Bd (m-1)} & \text{if } \texttt{v} = \texttt{Bd m}, \texttt{m} > \texttt{n} \\ \texttt{u} & \text{if } \texttt{v} = \texttt{Bd n} \\ \texttt{ret v} & \text{else} \end{cases}$$

$$\texttt{close x} = \texttt{fmapd}\,(\texttt{close}_{\text{loc}}\ \texttt{x}) \qquad \texttt{close}_{\text{loc}}\ \texttt{x}\ (\texttt{n}, \texttt{v}) = \begin{cases} \texttt{Bd (m+1)} & \text{if } \texttt{v} = \texttt{Bd m}, \texttt{m} \geq \texttt{n} \\ \texttt{Bd n} & \text{if } \texttt{v} = \texttt{Fr x} \\ \texttt{v} & \text{else} \end{cases}$$

$$\texttt{FV} = \texttt{foldMap}_{\texttt{list}}\ \texttt{FV}_{\text{loc}} \qquad \texttt{FV}_{\text{loc}}\ \texttt{v} = \begin{cases} \texttt{[x]} & \text{if } \texttt{v} = \texttt{Fr x} \\ \texttt{[]} & \text{else} \end{cases}$$

$$\texttt{LC} = \texttt{foldMapd}_{\wedge}\ \texttt{lc}_{\text{loc}} \qquad \texttt{LC}_{\text{loc}}\ (\texttt{n}, \texttt{v}) = \begin{cases} \texttt{n > m} & \text{if } \texttt{v} = \texttt{Bd m} \\ \texttt{True} & \text{else} \end{cases}$$

Figure 7 Locally nameless operations defined as special cases of `binddt`.

Reasoning about syntax often involves conditions on the variable occurrences – for example, `subst_fresh` requires knowledge about the freshness of a given variable. The next theorem gives a pointwise reasoning principle that is used to exploit information about occurrences. This theorem is proved using the coalgebraic presentation of traversability developed in [20].

▶ **Theorem 3.4** (Pointwise reasoning). *Let* $\texttt{T}$ *be a DTM. For all* $t : \texttt{T A}$ *and* $f, g : \texttt{W} \times \texttt{A} \to \texttt{F}\,(\texttt{T B})$ *where* $\texttt{F}$ *is any applicative functor, the following reasoning principle holds.*

$$(\forall\,(\texttt{w} : \texttt{W})\,(\texttt{a} : \texttt{A})\,, (\texttt{w}, \texttt{a}) \in_d \texttt{t} \implies \texttt{f}\ (\texttt{w}, \texttt{a}) = \texttt{g}\ (\texttt{w}, \texttt{a})) \implies \texttt{binddt}_{\texttt{F}}\ \texttt{f t} = \texttt{binddt}_{\texttt{F}}\ \texttt{g t}.$$

3.3 Locally nameless backend

Let $\texttt{T}$ be a DTM decorated by `nat`. We now define the operations of Figure 3 and prove some exemplary lemmas. Our locally backend actually uses multisorted DTMs (defined in the next section), but the basic principles are the same.

The five main operations are defined in Figure 7. On the right, each operation is defined "locally" in terms of its action on individual variable occurrences; three such operations require the number `n` of binders in scope. On the left, each operation is extended to operate on terms using a combinator, all of which are special cases of `binddt`. E.g. `bindd` is `binddt` specialized to the identity applicative, while `fmapd` is like `bindd` for maps rather than substitutions. The `foldMap*` operations instantiate the applicative functor argument of `binddt` to a monoid: `list` uses concentation, while $\wedge$ is shorthand for the monoid of propositions under conjunction. The following properties are proven for all atoms `x:atom` and abstract terms `t, u:T LN`.

▶ **Lemma 3.5** (`subst_fresh`). $\texttt{x} \notin \texttt{FV t} \implies \texttt{subst x u t} = \texttt{t}$.

Proof. Combining Theorem 3.4 with (2) reduces the problem to

$$\forall(\texttt{v} : \texttt{LN}),\ \texttt{v} \in \texttt{t} \implies \texttt{subst}_{\text{loc}}\ \texttt{x u v} = \texttt{ret v},$$

which follows by case analysis on `v` and a lemma $\texttt{y} \in \texttt{FV t} \iff \texttt{Fr y} \in \texttt{t}$ proved by (4). ◀

▶ **Lemma 3.6** (subst_spec). $\texttt{subst x u t} = \texttt{open u (close x t)}$.

Proof. By (3) and (1), open u (close x t) can be fused together to obtain

$$\texttt{bindd}\ (\lambda(\texttt{n}, \texttt{v}).\texttt{open}_{\texttt{loc}}\ \texttt{u}\ (\texttt{n}, \texttt{close}_{\texttt{loc}}\ \texttt{x}\ (\texttt{n}, \texttt{v})))\ \texttt{t}.$$

One then shows the middle expression is equal to $\lambda(\texttt{n}, \texttt{v}).\texttt{subst}_{\texttt{loc}}\ \texttt{x u v}$ by case analysis. ◀

▶ **Lemma 3.7** (fv_subst_upper). $\texttt{FV}\,(\texttt{subst x u t}) \subseteq (\texttt{FV t} \setminus \{\texttt{x}\}) \cup \texttt{FV u}$.

Proof. By $\texttt{x} \in \texttt{FV t} \iff \texttt{Fr x} \in \texttt{t}$, the problem reduces to

$$\texttt{Fr y} \in \texttt{subst x u t} \implies (\texttt{Fr y} \in \texttt{t} \wedge \texttt{y} \neq \texttt{x}) \vee (\texttt{Fr y} \in \texttt{u})\,.$$

This follows by rewriting the left hand side with (6), and case analysis. ◀

The vast majority of proofs in our locally nameless backend proceed along similar lines: fusing sequential operations with (1) and (3), using (5) and (6) to reason about how operations affect the set of occurrences, applying pointwise reasoning to prove equalities, and case analysis on concrete variables.

3.4 Multisorted DTMs

In practice, few formal systems involve just one sort of variable. For example, polymorphic lambda calculi like System F include both type and term variables. Fixed-point extensions of first-order logic [19], which provide a theoretical foundation for languages like Datalog [2], involve both term and relation variables, as do second-order logics. For such systems it is necessary to consider a generalization of Definition 3.1 that supports parallel substitution of *more than one sort of variable at the same time*. This is because our approach rests on the assumption that all substitution and related operations are special cases of a single operation, so they can always be fused together with an appropriate generalization of (3). This is our motivation for introducing Definition 3.8.

▶ **Definition 3.8** (Multisorted DTM). *Let K be a set of sorts. Let* $\texttt{T} : \texttt{K} \to \texttt{Type} \to \texttt{Type}$ *be a* K*-indexed set of type constructors and let* U *be a type constructor. A* traversable T-module decorated by a monoid $\langle \texttt{W}, \cdot, 1_{\texttt{W}} \rangle$ *is defined from the following data:*

$$\begin{array}{rlcl} \texttt{ret} & (\texttt{A} : \texttt{Type}) & : & \forall\,(\texttt{k} : \texttt{K})\,,\ \texttt{A} \to \texttt{T k A} \\ \texttt{binddt} & (\texttt{Applicative F})\ (\texttt{A B} : \texttt{Type}) & : & (\forall(\texttt{k} : \texttt{K}),\ \texttt{W} \times \texttt{A} \to \texttt{F}\,(\texttt{T k B})) \to \texttt{U A} \to \texttt{F}\ (\texttt{U B}) \end{array}$$

subject to conditions generalizing those of Definition 3.1.

For short, we call an instance of Definition 3.8 a K-sorted DTM (where "M" technically stands for "right Module.") When K is the unit type and $\texttt{T tt} = \texttt{U}$ (tt being Coq's name for the constructor of unit type), this definition reduces to Definition 3.1. In general, such structures represent a grammatical category U inside which one can substitute any of |K|-many different kinds of variables in parallel. Reasoning with multisorted DTMs works as before, but now incorporating case analysis on K as well.

4 Evaluating Tealeaves

Next we discuss how we evaluate Tealeaves. First we describe the size and scope of Tealeaves' core and backend before discussing case studies instantiating Tealeaves with different kinds of syntax. Then we offer a feature-wise comparison to two popular syntax frameworks for Coq, Autosubst versions 1 and 2 [32, 36] and LNgen [5].

Implementation

We recall Figure 6, which shows the division of Tealeaves into two major components:

- Core Tealeaves, which formalizes DTMs and the properties discussed in Section 3.2.
- The locally nameless backend, which develops generic locally nameless syntax infrastructure like that shown in Section 3.3.

The core is a formalization of multisorted DTMs (Definition 3.8). On top of the axioms we implement an additional layer of high-level derived theory like that presented in Section 3.2, the chief export of which is a proof of Theorem 3.4. Additionally, the core includes a general formalization of numerous category-theoretic concepts used to prove Theorem 3.2 for single-sorted DTMs, but this is primarily of theoretical interest; it does not affect end-users and would not be required to port Tealeaves to another proof assistant like Agda. Altogether, as measured by `coqwc`, the core includes about 10,000 lines of specification (including imports, notations, etc.) and 9,000 lines of proof. Of these, the essential parts formalizing multisorted DTMs account for about 2,000 lines of specification of 1,000 of proof.

The locally nameless backend includes a core part independent of DTMs that formalizes basic notions like atoms, sets, and environments. This part consists of about 2000 lines derived from the Metalib library, a component of LNgen, lightly adapted to fit into our more category-theoretic framework. The locally nameless infrastructure, which is parameterized by a DTM instance, consists of about 1000 lines of specification and 650 of proof. The backend export several dozen high-level infrastructural lemmas like the ones generated by LNgen, as well as many other lower-level lemmas. Examples of lemmas include the ones proved in Section 3.3, as well as generalizations that describe the interaction between operations that act on different sorts of variables.

The locally nameless backend supports the claim that the DTM abstraction is adequate for reasoning about raw syntax generically. Next we ask whether this concept is actually useful, i.e. does it save labor, and for which kinds of syntax does it work?

Case studies

So far we have implemented a few different case studies with Tealeaves.

STLC Our first study is a proof of type soundness for the simply-typed lambda calculus (STLC). We use Alectryon [27] to present this file in the form of browser-based tutorial on Equations (1)–(4), demonstrating the general strategy for proving each one. We also provide an alternate version of this tutorial that uses the category-theoretic description of DTMs indicated in Theorem 3.2.

System F In the second study, we instantiate Tealeaves with the syntax of System F before proving type soundness for this system. This makes essential use of multisorted DTMs and the ability of our backend to reason about non-trivial interactions between substitution operations that act on different sorts of variables.

Variadic binding We are developing tutorials demonstrating how to instantiate Tealeaves with languages featuring mutually-inductively defined grammatical categories and variadic binders, such as a `letrec` construct.

The cost to instantiate Tealeaves is to define `binddt` and prove multisorted versions of Equations (1)–(4). These proofs proceed by induction on terms, where each case proceeds by rewriting with laws like those of applicative functors. In the future, we expect to provide automated support for the instantiation process. Happily, three of the DTM axioms are straightforward to prove in most cases, regardless of the user's syntax. Equation (1) defines the behavior of `binddt` on variables and is proved with the `reflexivity` tactic, while (2)

and (4) are straightforward inductive proofs. Equation (3), however, presents a challenge when binding information (i.e. data passed with `preincr`) is computed from an argument which itself is subject to substitution. A key example of this phenomenon is a variadic `let` (or `letrec`) construct that accepts a `list` of definitions, in which case `binddt f` is defined to pass the length of the list to `f` in the `let` body. The bound definitions are themselves subject to substitution with `binddt`, and it is not immediately clear how to prove that this does not change the length of the list, a key requirement of Equation (3) manifest in the two occurrences of `w`. This requires applying the representation theorem for traversals [18], which states that shape is invariant under traversals.

Comparison to other utilities

Three commonly used utilities for automating syntax infrastructure in Coq are Autosubst [32], Autosubst 2 [36], and LNgen [5], all of which involve dynamically generating infrastructure after being provided with a user's syntax. The Autosubst family represent variables as de Bruijn indices, while LNgen generates locally nameless infrastructure. In some ways Tealeaves is more general than these utilities, as the operations they reason about are special cases of `binddt`. Table 1 summarizes the features offered by the utilities.

Autosubst provides tool support for working with de Bruijn indices based on the σ-calculus [1], a version of untyped λ-calculus extended with explicit substitution. Given an `Inductive` definition of a user's syntax, the user calls upon Ltac to synthesize a parallel substitution operation and a small number of equational axioms for this operation. Users invoke a complete decision procedure, `autosubst`, which proves all true equalities between a delineated class of *substitution expressions* from these axioms. Semanticists generate these goals while developing metatheory and call on Autosubst to solve them.

Autosubst provides only ad-hoc support for substitution involving multiple sorts of variables. The limitations of Ltac also prevent their automation from working with mutually-inductively defined grammatical categories. The authors note that the fragile semantics of Ltac mean it is sometimes necessary to manually inspect generated definitions for errors.

Autosubst 2 is an external code generator written in Haskell which accepts a second-order specification of a syntax and generates Coq modules containing proofs of the equations to instantiate an extended calculus that handles multisorted substitution much the same way we do. Compared to the first version, Autosubst 2 handles potentially mutually inductive grammatical categories with multiple kinds of variables. The authors conjecture, but do not prove, that their modified calculus is confluent. Users who modify their syntax must re-execute the external program to reinstantiate the Autosubst library.

The Autosubst family does not provide support for conditional equalities or operations that compute the set of free variables, perhaps because these are not as essential when using de Bruijn indices as when using a locally nameless representation.

LNgen is a code generator that, given an annotated grammar in an Ott-compatible [33] format, generates Coq files containing the operations of locally nameless and proof scripts than synthesize infrastructural lemmas. The scripts proceed by induction and are based on the authors' "knowledge of how such proofs usually go." As with Autosubst 2, modifying the syntax involves re-executing the utility. In private correspondence, the authors of LNgen have reported to us cases of long compile times (about 30 minutes in some cases) and the potential for some proofs to fail, requiring manual intervention from the user. As with Autosubst, this problem is exacerbated by the opaque semantics of Ltac.

Table 1 Features supported by Coq syntax frameworks.

Utility	Representation	Underlying theory	Multisorted	Variadic Binders
Autosubst	de Bruijn	σ-calculus	Ad-hoc	No
Autosubst 2	de Bruijn	σ-calculus	Yes	No
LNgen	Locally nameless	Structural recursion	Yes	No
Tealeaves	Generic	DTMs	Yes	Yes

5 Related work

Besides Autosubst and LNgen, there are syntax metatheory frameworks for Coq that share some of Tealeaves' features but lack the principled theory and flexibility of DTMs.

GMeta [23] is prior art implementing a generic Coq framework for first-order syntax metatheory. Like Tealeaves, it features an extensible architecture supporting multi-sorted syntax and multiple representations of variable binding. However, the implementations differ substantially because GMeta lacks a principled abstraction of syntax like DTMs, considering instead a universe of representable types. In effect, one has a set of type expressions and a denotation mapping these into Coq's types; generic proofs proceed by induction on an expression denoting a type. By contrast, we showed in Section 3.3 how infrastructural lemmas with Tealeaves proceed by the equational theory of DTMs. The user's cost of entry for GMeta is to prove the type of their syntax is representable up to isomorphism, which is supported with automation.

DBlib [31] is a community-maintained Coq library that supports reasoning about de Bruijn indices. Like Tealeaves, it is based on a structured recursion combinator subject to axioms, but these axioms are ad-hoc and not pure equations, whereas (1)–(4) are equations derived from a principled theory of structured monads as manifest in Theorem 3.2. Using results from Section 3.2, it is easy to see that DBlib's axioms are immediate corollaries of DTMs. For instance, their axiom `TraverseVarIsIdentity` can be derived by specializing Theorem 3.4 to $\mathtt{g} = \mathtt{ret} \circ \mathtt{extract}^{\mathtt{W}\times}$ and simplifying with (2).

The application of monads to formal syntax metatheory was proposed by Bellegarde and Hook [7], who considered a combinator `Ewp` ("extension with policy") that is reminiscent of `binddt` but less expressive and lacking an axiomatization. Work building on the monadic approach, typically using a de Bruijn representation, has emphasized well-scoped [9, 4] and well-scoped, well-typed [3] syntax. Fiore and Szamozvancev have recently introduced an Agda framework for well-typed syntax that is inspired by work on presheaf-theoretic models of syntax [16, 15]. The heavy use of dependent types in this work leads to a workflow in which the types of operations are very nearly their own correctness properties, whereas our "raw" approach separates the definition of operations from their metatheory. Investigating the theoretical relation between the two approaches may be an interesting direction for future work.

Two fundamentally different formalization strategies for abstract syntax are higher-order abstract syntax (HOAS) [26] and techniques using nominal sets [28], both of which are closely associated with dedicated-purpose proof assistants. Implementing HOAS in Coq requires using a variation like parametric higher-order abstract syntax (PHOAS) [12]. Nominal sets are generally used with first-class support from the proof assistant, such as in Nominal Isabelle [37].

6 Conclusion and future work

We have presented Tealeaves, a generic Coq framework for reusable syntax metatheory. We showed how a user instantiates Tealeaves by proving their syntax forms a DTM, allowing them to specialize a body of generic infrastructure lemmas to their syntax. We evaluated Tealeaves with case studies instantiating locally nameless infrastructure to languages with multiple sorts of variables, mutually-inductive grammatical categories, and variadic binders.

Tealeaves offers a number of interesting directions for future investigation. Currently we are investigating precisely which kinds of syntax and reasoning work well with Tealeaves. More precisely, we are exploring how the core theory of DTMs can be modified to accommodate more sophisticated situations than raw terms with locally nameless variables or de Bruijn indices.

Well-scoped syntax

We initially sought an abstraction for raw syntax, largely because this representation is simple and commonly used. However, there are convincing theoretical and practical reasons to consider intrinisically well-scoped syntax. We are investigating how to extend DTMs to the well-scoped setting. As a first step, let `LN` be parameterized by a context `ctx` of free variables and by a maximum value `n` for de Bruijn indices using Coq's type `Fin.t` of finite sets, as follows.

```
Inductive LN (ctx : list atom) (n : nat) : Type :=
| Fr : forall (a : atom), In a ctx -> LN ctx n
| Bd : Fin.t n -> LN ctx n.
```

The type of lambda terms is generalized to allow the set of variables to be parameterized by the number of entities in scope (here, `preincr V n` maps `m` to the set `V (n + m)`).

```
Inductive Lam (V : nat -> Type) :=
| Var : V 0 -> Lam V
| Abs : Lam (preincr V 1) -> Lam V
| App : Lam V -> Lam V -> Lam V.
```

The type of locally closed terms with free variables in `ctx` is then `Lam (LN ctx)`. For example, the term `Var (Fr x)` can be given this type if `x ∈ ctx`. On the other hand, the open term `Var (Bd 0)` cannot be given this type, while it can be given type `Lam (preincr (LN ctx) 1)`.

A generalization of Definition 3.1 can be formulated for this situation on paper. An unfortunate limitation of Coq's type theory is that types like `LN ((n + m) + p)` and `LN (n + (m + p))` are not defintionally equal, hence their terms cannot even be compared for equality, obstructing a naïve attempt to formalize this definition in Coq. Further parameterizing `V` by types would also move closer towards the type-preserving approach of McBride [24].

Fully named variables

The Tealeaves repository includes a generalization of Definition 3.1 that additionally takes a binder-renaming operation, with which we intend to implement a fully named Tealeaves backend. With such an extension, our aim is to give a certified change in representation between locally nameless and fully named variables. One use case would be to implement a verified programming language in Coq using locally nameless while allowing programmers to write code with named variables, assured that the change in representation introduces no bugs.

References

1 M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991. doi:10.1017/S0956796800000186.

2 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1995.

3 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. *Journal of Functional Programming*, 31:e22, 2021. doi:10.1017/S0956796820000076.

4 Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings*, volume 1683 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 1999. doi:10.1007/3-540-48168-0_32.

5 Brian Aydemir and Stephanie Weirich. LNgen: Tool Support for Locally Nameless Representations. Technical report, University of Pennsylvania, Department of Computer and Information Science, June 2010. URL: https://repository.upenn.edu/cis_reports/933/.

6 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 *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs'05, pages 50–65, Berlin, Heidelberg, 2005. Springer-Verlag. doi:10.1007/11541868_4.

7 Françoise Bellegarde and James Hook. Substitution: A formal methods case study using monads and transformations. *Sci. Comput. Program.*, 23(2–3):287–311, December 1994. doi:10.1016/0167-6423(94)00022-0.

8 Richard Bird, Jeremy Gibbons, Stefan Mehner, Janis Voigtländer, and Tom Schrijvers. Understanding idiomatic traversals backwards and forwards. *SIGPLAN Not.*, 48(12):25–36, September 2013. doi:10.1145/2578854.2503781.

9 Richard S. Bird and Ross Paterson. De bruijn notation as a nested datatype. *J. Funct. Program.*, 9(1):77–91, 1999. doi:10.1017/s0956796899003366.

10 Rod M. Burstall. Proving properties of programs by structural induction. *Comput. J.*, 12(1):41–48, 1969. doi:10.1093/comjnl/12.1.41.

11 Arthur Charguéraud. The locally nameless representation. *J. Autom. Reason.*, 49(3):363–408, 2012. doi:10.1007/s10817-011-9225-2.

12 Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 143–156, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1411204.1411226.

13 N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. doi:10.1016/1385-7258(72)90034-0.

14 David Delahaye. A tactic language for the system coq. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95. Springer, 2000. doi:10.1007/3-540-44404-1_7.

15 Marcelo Fiore. Second-order and dependently-sorted abstract syntax. In *Proceedings of the 2008 23rd Annual IEEE Symposium on Logic in Computer Science*, LICS '08, pages 57–68, USA, 2008. IEEE Computer Society. doi:10.1109/LICS.2008.38.

16 Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 193–202. IEEE Computer Society, 1999. doi:10.1109/LICS.1999.782615.

17 Marcelo Fiore and Dmitrij Szamozvancev. Formal metatheory of second-order abstract syntax. *Proc. ACM Program. Lang.*, 6(POPL), January 2022. doi:10.1145/3498715.

18 Jeremy Gibbons and Bruno Oliveira. The essence of the iterator pattern. *J. Funct. Program.*, 19:377–402, July 2009. doi:10.1017/S0956796809007291.

19 Yuri Gurevich and Saharon Shelah. Fixed-point extensions of first-order logic. *Annals of Pure and Applied Logic*, 32(nil):265–280, 1986. doi:10.1016/0168-0072(86)90055-2.

20 Mauro Jaskelioff and Russell O'Connor. A representation theorem for second-order functionals. *Journal of Functional Programming*, 25, February 2014. doi:10.1017/S0956796815000088.

21 Mauro Jaskelioff and Ondrej Rypacek. An investigation of the laws of traversals. *Electronic Proceedings in Theoretical Computer Science*, 76, February 2012. doi:10.4204/EPTCS.76.5.

22 C. Barry Jay and J. Robin B. Cockett. Shapely types and shape polymorphism. In Donald Sannella, editor, *Programming Languages and Systems - ESOP'94, 5th European Symposium on Programming, Edinburgh, UK, April 11-13, 1994, Proceedings*, volume 788 of *Lecture Notes in Computer Science*, pages 302–316. Springer, 1994. doi:10.1007/3-540-57880-3_20.

23 Gyesik Lee, Bruno C. D. S. Oliveira, Sungkeun Cho, and Kwangkeun Yi. GMeta: A generic formal metatheory framework for first-order representations. In Helmut Seidl, editor, *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 436–455. Springer, 2012. doi:10.1007/978-3-642-28869-2_22.

24 Conor McBride. Type-preserving renaming and substitution. Unpublished note, 2005. URL: http://strictlypositive.org/ren-sub.pdf.

25 Conor McBride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, January 2008. doi:10.1017/S0956796807006326.

26 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.

27 Clément Pit-Claudel. Untangling mechanized proofs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2020, pages 155–174, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3426425.3426940.

28 Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186(2):165–193, 2003. Theoretical Aspects of Computer Software (TACS 2001). doi:10.1016/S0890-5401(03)00138-X.

29 Randy Pollack. Closure under alpha-conversion. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, pages 313–332, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

30 Randy Pollack. Reasoning about languages with binding: Can we do it yet?, February 2006. URL: https://web.archive.org/web/20101122040606/http://homepages.inf.ed.ac.uk/rpollack/export/bindingChallenge_slides.pdf.

31 François Pottier and Coq maintainers. DBlib. https://github.com/coq-community/dblib, 2019.

32 Steven Schäfer, Tobias Tebbi, and Gert Smolka. Autosubst: Reasoning with de bruijn terms and parallel substitutions. In Xingyuan Zhang and Christian Urban, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015*, LNAI. Springer-Verlag, August 2015. doi:10.1007/978-3-319-22102-1_24.

33 Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 1–12, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1291151.1291155.

34 Matthieu Sozeau and Nicolas Oury. First-class type classes. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '08, pages 278–293, Berlin, Heidelberg, 2008. Springer-Verlag. doi:10.1007/978-3-540-71067-7_23.

35 Bas Spitters and Eelis Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21, February 2011. doi:10.1017/S0960129511000119.

36 Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: Reasoning with multi-sorted de bruijn terms and vector substitutions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, pages 166–180, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293880.3294101.

37 Christian Urban and Christine Tasson. Nominal techniques in isabelle/hol. In Robert Nieuwenhuis, editor, *Automated Deduction – CADE-20*, pages 38–53, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. doi:10.1007/11532231_4.

38 Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 1–14, New York, NY, USA, 1992. Association for Computing Machinery. doi:10.1145/143165.143169.

Closure Properties of General Grammars – Formally Verified

Martin Dvorak
Institute of Science and Technology Austria, Klosterneuburg, Austria

Jasmin Blanchette
Ludwig-Maximilians-Universität München, Germany

Abstract

We formalized general (i.e., type-0) grammars using the Lean 3 proof assistant. We defined basic notions of rewrite rules and of words derived by a grammar, and used grammars to show closure of the class of type-0 languages under four operations: union, reversal, concatenation, and the Kleene star. The literature mostly focuses on Turing machine arguments, which are possibly more difficult to formalize. For the Kleene star, we could not follow the literature and came up with our own grammar-based construction.

2012 ACM Subject Classification Theory of computation → Rewrite systems

Keywords and phrases Lean, type-0 grammars, recursively enumerable languages, Kleene star

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.15

Related Version *Previous Version*: `https://arxiv.org/abs/2302.06420`

Supplementary Material *Software (Source code)*: `https://github.com/madvorak/grammars/tree/publish`, archived at `swh:1:dir:232a6421be2d20d29e54fea05cebdc865bd9c489`

Funding *Jasmin Blanchette*: This research has received funding from the Netherlands Organization for Scientific Research (NWO) under the Vidi program (project No. 016.Vidi.189.037, Lean Forward).

Acknowledgements We thank Vladimir Kolmogorov for making this collaboration possible. We thank Václav Končický for discussing ideas about the Kleene star construction. We thank Patrick Johnson, Floris van Doorn, and Damiano Testa for their small yet very valuable contributions to our code. We thank Eric Wieser for simplifying one of our proofs. We thank Mark Summerfield for suggesting textual improvements. We thank the anonymous reviewers for very helpful comments. Finally, we thank the Lean community for helping us with various technical issues and answering many questions.

1 Introduction

The notion of formal languages lies at the heart of computer science. There are several formalisms that recognize formal languages, including Turing machines and formal grammars. In particular, both Turing machines and general grammars (also called type-0 grammars or unrestricted grammars) characterize the same class of languages, namely, the recursively enumerable or type-0 languages.

There has been work on formalizing Turing machines in proof assistants [7, 2, 26, 6, 15, 3]. General grammars are an interesting alternative because they are easier to define than Turing machines, and some proofs about general grammars are much easier than the proofs of similar properties of Turing machines.

We therefore chose general grammars as the basis for our Lean 3 [9] library of results about recursively enumerable or type-0 languages. The definition of grammars consists of several layers of concepts (Section 2):

- the type of symbols is the disjoint union of terminals and nonterminals;

© Martin Dvorak and Jasmin Blanchette;
licensed under Creative Commons License CC-BY 4.0

- rewrite rules are pairs of the form $u \to v$, where u and v are strings over symbols and u contains at least one nonterminal [1];
- a grammar is a tuple consisting of a type of terminals, a type of nonterminals, an initial symbol S, and a set of rewrite rules;
- application of a rewrite rule $u \to v$ to a string $\alpha u \beta$ is written $\alpha u \beta \Rightarrow \alpha v \beta$;
- the derivation relation $\Rightarrow^*$ is the reflexive transitive closure of the $\Rightarrow$ relation;
- a grammar derives a word w if $S \Rightarrow^* w$;
- the language generated by a grammar is the set of words derived by it;
- a language is type 0 if there exists a grammar that generates it.

We formalized four closure properties of type-0 languages.

The first such property we present is closure of type-0 languages under union (Section 3). We followed the standard construction for context-free grammars, which incidentally works for general grammars as well.

The second closure property we formalized is closure under reversal (Section 4). This was straightforward.

The third closure property we formalized is closure under concatenation (Section 5). The main difficulty was to avoid matching strings on the boundary of the concatenation. This issue does not arise with context-free grammars because only single symbols are matched and these are tidily located on either side of the boundary.

The fourth and last closure property we formalized is closure under the Kleene star (Section 6). This was the most difficult part of our work. Because the literature mostly focuses on Turing machine arguments, we needed to invent our own construction. We first developed a detailed proof sketch and then formalized it. The sketch is included in this paper.

One closure property we did not formalize is closure under intersection. The reason is that we are not aware of any elegant construction based on grammars only. Recall that type-0 languages are not closed under complement, as witnessed by the halting problem [18].

Our development is freely available online.[1] It consists of about 12 500 lines of spaciously formatted Lean code. It uses the Lean 3 mathematical library `mathlib` [22].[2] We also benefited from the metaprogramming framework [10], which allowed us to easily develop small-scale automation that helped make some proofs less verbose.

Although Lean is based on dependent type theory [21], our code uses only nondependent types for data. We still found dependent type theory useful for bound-checked indexing of lists using the function `list.nth_le` (which takes a list, an index, and a proof that the index is within bounds as arguments). We did not attempt to make our development constructive.

2 Definitions

2.1 Grammars

As outlined in the introduction, the definition of grammars consists of several layers of declarations.

Symbols are essentially defined as a sum type of terminals `T` and nonterminals `N`. However, we want to refer to terminals and nonterminals by name (using `symbol.terminal` and `symbol.nonterminal` instead of `sum.inl` and `sum.inr`), so we define symbols as an inductive type:

[1] `https://github.com/madvorak/grammars/tree/publish`
[2] `https://github.com/leanprover-community/mathlib/tree/7ed4f2cec2`

```
inductive symbol (T : Type) (N : Type)
| terminal    : T → symbol
| nonterminal : N → symbol
```

We do not require `T` and `N` to be finite. As a result, we do not need to copy the typeclass instances `[fintype T]` and `[fintype N]` alongside our type parameters (which would appear in almost every lemma statement). Instead, later we work in terms of a list of rewrite rules, which is finite by definition and from which we could infer that only a finite set of terminals and a finite set of nonterminals can occur.

The left-hand side u of a rewrite rule $u \to v$ consists of three parts (an arbitrary string α, a nonterminal A, and another arbitrary string β, such that $u = \alpha A \beta$):

```
structure grule (T : Type) (N : Type) :=
(input_L : list (symbol T N))
(input_N : N)
(input_R : list (symbol T N))
(output_string : list (symbol T N))
```

An advantage of this representation is that we do not need to carry the proposition "the left-hand side contains a nonterminal" around. A disadvantage is that we subsequently need to concatenate more terms.

A definition of a general grammar follows. Notice that only the type argument `T` is part of its type:

```
structure grammar (T : Type) :=
(nt : Type)
(initial : nt)
(rules : list (grule T nt))
```

Later we can use the dot notation to access individual fields. For example, if `g` is a term of the type `grammar T`, we can write `g.nt` to access the type of its nonterminals. By writing `(g.rules.nth_le 0 _).output_string` we obtain the right-hand side of the first rewrite rule in `g`. The underscore, when not inferred automatically, must be replaced by a term of the type `0 < g.rules.length`, which is a proof that the list `g.rules` is not empty.

The next line adds an implicit type argument `T` to all declarations that come after:

```
variables {T : Type}
```

The following definition captures the application $\Rightarrow$ of a rewrite rule:

```
def grammar_transforms (g : grammar T)
    (w₁ w₂ : list (symbol T g.nt)) :
  Prop :=
∃ r : grule T g.nt,
  r ∈ g.rules    ∧
  ∃ u v : list (symbol T g.nt),
    w₁ = u ++ r.input_L ++ [symbol.nonterminal r.input_N]
           ++ r.input_R ++ v    ∧
    w₂ = u ++ r.output_string ++ v
```

The operator `++` concatenates two lists. We can view `grammar_transforms` as a function that takes a grammar `g` over the terminal type `T` and outputs a binary relation over strings of the type that `g` works internally with.

The part `r.input_L ++ [symbol.nonterminal r.input_N] ++ r.input_R` represents the left-hand side of the rewrite rule `r`. Note that the terms `r.input_L` and `r.input_N` cannot be concatenated directly, since they have different types. The term `r.input_N` must first be wrapped in `symbol.nonterminal` to go from the type `g.nt` to the type `symbol T g.nt` and then surrounded by `[]` to become a (singleton) list.

The derivation relation $\Rightarrow^*$ is defined from `grammar_transforms` using the reflexive transitive closure:

```
def grammar_derives (g : grammar T) :
  list (symbol T g.nt) → list (symbol T g.nt) → Prop :=
relation.refl_trans_gen (grammar_transforms g)
```

Consequently, proofs about derivations will use structural induction.

The predicate "to be a word generated by the grammar `g`" is defined as the special case of the relation `grammar_derives g` where the left-hand side is fixed to be the singleton list made of the initial symbol of `g` and the right-hand side is required to consist of terminal symbols only:

```
def grammar_generates (g : grammar T) (w : list T) : Prop :=
grammar_derives g [symbol.nonterminal g.initial]
  (list.map symbol.terminal w)
```

2.2 Languages

In our entire project, we work with the following definition of languages provided by `mathlib` in the `computability` package:

```
def language (α : Type*) := set (list α)
```

The type argument α is instantiated by our terminal type `T` in all places where we work with languages. We do not mind restricting `T` to be `Type` since we are not interested in languages over types from `Type 1` and higher universes.

The language of the grammar `g` is defined as the set of all `w` that satisfy the predicate `grammar_generates g w` declared above:

```
def grammar_language (g : grammar T) : language T :=
set_of (grammar_generates g)
```

Note that the type parameter `T` is preserved, but `g.nt` does not matter in the description of what words are generated. It corresponds to our intuition that the type of terminals is a part of the interface, but the type of nonterminals is an implementation matter.

This is the first time that our custom types meet the standard `mathlib` type `language`, which is already connected to many useful types, such as the type of regular expressions.

Finally, we define the class of type-0 languages:

```
def is_T0 (L : language T) : Prop :=
∃ g : grammar T, grammar_language g = L
```

All top-level theorems about type-0 languages are expressed in terms of the `is_T0` predicate.

Note that the type system distinguishes between a list of terminals and a list of symbols that happen to be terminals. Languages are defined as sets of the former, whereas derivations in the grammar work with the latter.

In a similar way, we define `CF_grammar`, `CF_transforms`, `CF_derives`, `CF_generates`, `CF_language`, and the `is_CF` predicate for the formal definition of context-free languages.

The theorem `CF_subclass_T0` connects the context-free languages to the type-0 languages. Type-0 languages remain the main focus of our work.

2.3 Operations

The operations under which we prove closure are defined below.

Union is defined in `mathlib` as follows:

```
protected def set.union (s₁ s₂ : set α) : set α :=
{a | a ∈ s₁ ∨ a ∈ s₂}
```

The following declaration in `mathlib` states that the union of languages is denoted by writing the `+` operator between two terms of the `language` type:

```
instance : language.has_add (language α) := ⟨set.union⟩
```

We define the reversal of a language as follows:

```
def reverse_lang (L : language T) : language T :=
λ w : list T, w.reverse ∈ L
```

We do not declare any syntactic sugar for reversal.

Concatenation is defined using the following general `mathlib` definition:

```
def set.image2 (f : α → β → γ) (s : set α) (t : set β) : set γ :=
{c | ∃ a b, a ∈ s ∧ b ∈ t ∧ f a b = c}
```

The next `mathlib` declaration states that concatenation of languages is denoted by writing the `*` operator between two terms of the `language` type:

```
instance : language.has_mul (language α) := ⟨set.image2 (++)⟩
```

The Kleene star of a language is defined in `mathlib` as follows:

```
def language.star (l : language α) : language α :=
{x | ∃ S : list (list α), x = S.join ∧ ∀ y ∈ S, y ∈ l}
```

We do not declare any syntactic sugar for the Kleene star.

3 Closure under Union

In this section, we prove the following theorem:

```
theorem T0_of_T0_u_T0 (L₁ : language T) (L₂ : language T) :
  is_T0 L₁ ∧ is_T0 L₂ → is_T0 (L₁ + L₂)
```

The proof of closure of type-0 languages under union consists of three main ingredients:

(1) a construction of a new grammar $\mathtt{g}$ from any two given grammars $\mathtt{g}_1$ and $\mathtt{g}_2$;
(2) a proof that any word generated by $\mathtt{g}_1$ or $\mathtt{g}_2$ can also be generated by $\mathtt{g}$;
(3) a proof that any word generated by $\mathtt{g}$ can be equally generated by $\mathtt{g}_1$ or $\mathtt{g}_2$.

Proofs of the other closure properties are organized analogously. We describe the proof of closure under union in more detail; it allows us to outline the main ideas of proving closure properties formally in a simple setting. Since (3) is usually much more difficult than (2), we refer to (2) as the "easy direction" and to (3) as the "hard direction".

The proof of the closure of type-0 languages under union follows the standard construction, which usually states only (1) explicitly, and leaves (2) and (3) to the reader. We begin (1) by defining a new type of nonterminals. The nonterminals of `g` consist of

- the nonterminals of g_1 including a mark indicating their origin;
- the nonterminals of g_2 including a mark indicating their origin;
- one new distinguished nonterminal.

The Lean type `option (g₁.nt ⊕ g₂.nt)` encodes this disjoint union. If `m` is a nonterminal of type `g₁.nt`, its corresponding nonterminal of type `g.nt` is `some (sum.inl m)`. If `n` is a nonterminal of type `g₂.nt`, its corresponding nonterminal of type `g.nt` is `some (sum.inr n)`. The new distinguished nonterminal is called `none` and becomes the initial symbol of `g`. The rewrite rules of `g` consist of

- the rewrite rules of g_1 with all nonterminals mapped to the larger nonterminal type;
- the rewrite rules of g_2 with all nonterminals mapped to the larger nonterminal type;
- two additional rules that rewrite the initial symbol of `g` to the initial symbol of g_1 or g_2.

To reduce the amount of repeated code in the proof, we developed lemmas that allow us to "lift" a grammar with a certain type of nonterminals to a grammar with a larger type of nonterminals while preserving what the grammar derives. Under certain conditions, we can also "sink" the larger grammar to the original grammar and preserve its derivations.

These lemmas operate on a structure called `lifted_grammar` that consists of the following fields:

- a smaller grammar g_0 that represents either g_1 or g_2 in case of the proof for union;
- a larger grammar `g` with the same type of terminals;
- a function `lift_nt` from `g₀.nt` to `g.nt`;
- a partial function `sink_nt` from `g.nt` to `g₀.nt`;
- a proposition `lift_inj` that guarantees that `lift_nt` is injective;
- a proposition `sink_inj` that guarantees that `sink_nt` is injective on inputs for which g_0 has a corresponding nonterminal;
- a proposition `lift_nt_sink` that guarantees that `sink_nt` is essentially an inverse of `lift_nt`;
- a proposition `corresponding_rules` that guarantees that `g` has a rewrite rule for each rewrite rule g_0 has (with different type but the same behavior);
- a proposition `preimage_of_rules` that guarantees that g_0 has a rewrite rule for each rewrite rule of `g` whose nonterminal has a preimage on the g_0 side.

Thanks to this structure, we can abstract from the specifics of how the larger grammar is constructed in concrete proofs and care only about the properties that are required to follow analogous derivations.

To illustrate how we work with this abstraction, we review the proof of the following lemma:

```
private lemma lift_tran {lg : lifted_grammar T}
    {w₁ w₂ : list (symbol T lg.g₀.nt)}
    (hyp : grammar_transforms lg.g₀ w₁ w₂) :
  grammar_transforms lg.g
    (lift_string lg.lift_nt w₁)
    (lift_string lg.lift_nt w₂)
```

We need to show that if g_0 has a rewrite rule that transforms `w₁` to `w₂`, then `g` has a rewrite rule that transforms `lift_string lg.lift_nt w₁` to `lift_string lg.lift_nt w₂`.

We start by deconstructing `hyp` according to the `grammar_transforms` definition. To go from g_0 to g, we first "lift" the rewrite rule (i.e., translate its nonterminals in all fields) that g_0 used. We call `corresponding_rules` to show that g has such a rule. Then we use the function `lift_string` to lift `u` and `v`, which are the parts of the string w_1 that were not matched by the rule. We are then left with the proof obligation

```
lift_string lg.lift_nt w₁ =
lift_string lg.lift_nt u ++ (lift_rule lg.lift_nt r).input_L
  ++ [symbol.nonterminal (lift_rule lg.lift_nt r).input_N]
  ++ (lift_rule lg.lift_nt r).input_R ++ lift_string lg.lift_nt v
```

where

```
w₁ =
u ++ r.input_L ++ [symbol.nonterminal r.input_N] ++ r.input_R ++ v
```

and with the proof obligation

```
lift_string lg.lift_nt w₂ =
lift_string lg.lift_nt u ++ (lift_rule lg.lift_nt r).output_string
  ++ lift_string lg.lift_nt v
```

where

```
w₂ = u ++ r.output_string ++ v
```

These two obligations originate from the two identities in the definition `grammar_transforms` from Section 2. Essentially, we discharge them using the distributivity of `lift_string` over the `++` operation.

The abstraction provided by `lifted_grammar` takes care of the vast majority of our proof of the closure of type-0 languages under union. It remains to separately analyze what was the first step of the derivation that g did in the hard direction. We need to exclude all rules that are inherited from g_1 and g_2 and perform a case analysis on the two special rules.

The two additional rules of g are context-free. Therefore, if g_1 and g_2 have context-free rules only, then all rules of g are context-free as well. As a consequence, our result about type-0 languages can easily be reused to prove the closure of context-free languages under union:

```
theorem CF_of_CF_u_CF (L₁ : language T) (L₂ : language T) :
  is_CF L₁ ∧ is_CF L₂ → is_CF (L₁ + L₂)
```

Not much Lean code needs to be duplicated to obtain the result about context-free grammars. We need to write the construction of g again and the main result again. The remaining parts are achieved by reusing lemmas from the proof for general grammars. The main overhead is proving

```
private lemma union_grammar_eq_union_CF_grammar
    {g₁ g₂ : CF_grammar T} :
  union_grammar (grammar_of_cfg g₁) (grammar_of_cfg g₂) =
  grammar_of_cfg (union_CF_grammar g₁ g₂)
```

Even though the statement might look complicated, the proof has only five lines, making it one of the shortest tactic-based proofs in our project.

4 Closure under Reversal

In this section, we prove the following theorem:

```
theorem T0_of_reverse_T0 (L : language T) :
  is_T0 L → is_T0 (reverse_lang L)
```

The proof is very easy. Simply speaking, everything gets reversed. We start with the rewrite rules:

```
private def reversal_grule {N : Type} (r : grule T N) : grule T N :=
grule.mk r.input_R.reverse r.input_N r.input_L.reverse
  r.output_string.reverse
```

The constructor `grule.mk` takes arguments in the same order as they are written in the definition:

- its `input_L` is instantiated by `r.input_R.reverse`;
- its `input_N` is instantiated by `r.input_N`;
- its `input_R` is instantiated by `r.input_L.reverse`;
- its `output_string` is instantiated by `r.output_string.reverse`.

The new grammar is constructed as follows:

```
private def reversal_grammar (g : grammar T) : grammar T :=
grammar.mk g.nt g.initial (list.map reversal_grule g.rules)
```

The rest is essentially a repeated application of lemma `list.reverse_append_append`, which is just a repeated application of lemma `list.reverse_append`, which states that reversing two concatenated lists is equivalent to reversing both parts and concatenating them in the opposite order, and lemma `list.reverse_reverse`, which states that `list.reverse` is a dual operation.

5 Closure under Concatenation

In this section, we prove the following theorem:

```
theorem T0_of_T0_c_T0 (L₁ : language T) (L₂ : language T) :
  is_T0 L₁ ∧ is_T0 L₂ → is_T0 (L₁ * L₂)
```

Because the proof is highly technical, we only outline the main idea here.

We first review the classical construction for context-free grammars. Let $L_1 \subseteq T^*$ be a language generated by a grammar $G_1 = (N_1, T, P_1, S_1)$. Let $L_2 \subseteq T^*$ be a language generated by a grammar $G_2 = (N_2, T, P_2, S_2)$. Without loss of generality, the sets N_1 and N_2 are disjoint. We create a new initial symbol S that appears only in the rule $S \to S_1 S_2$. The new grammar is $(N_1 \cup N_2 \cup \{S\},\ T,\ P_1 \cup P_2 \cup \{S \to S_1 S_2\},\ S)$. This construction works for context-free grammars because S_1 gives rise to a word from L_1 and, independently, S_2 gives rise to a word from L_2.

For general grammars, the construction above does not work, as the following counterexample over $T = \{a, b\}$ illustrates. Let the rule sets be $P_1 = \{S_1 \to S_1 a,\ S_1 \to \epsilon\}$ and $P_2 = \{S_2 \to S_2 a,\ S_2 \to \epsilon,\ a S_2 \to b\}$. We obtain $L_1 = L_2 = \{a^n \mid n \in \mathbb{N}_0\}$ and so $L_1 L_2$ is $\{a^n \mid n \in \mathbb{N}_0\}$ as well. We can now derive $S \Rightarrow S_1 S_2 \Rightarrow S_1 a S_2 \Rightarrow S_1 b \Rightarrow b \notin L_1 L_2$ and obtain a contradiction.

We need to avoid matching strings that span across the boundary of the concatenation. Since the nonterminal sets are disjoint, the issue arises only with terminals in the left-hand side of rules, which are not present in context-free grammars. We provide a solution below.

Let $\mathtt{g}_1$ and $\mathtt{g}_2$ generate $\mathtt{L}_1$ and $\mathtt{L}_2$ respectively. The nonterminals of our new grammar `g` consist of

- the nonterminals of $\mathtt{g}_1$ including a mark indicating their origin;
- the nonterminals of $\mathtt{g}_2$ including a mark indicating their origin;
- a proxy nonterminal for every terminal from `T` marked for use by $\mathtt{g}_1$ only;
- a proxy nonterminal for every terminal from `T` marked for use by $\mathtt{g}_2$ only;
- one new distinguished nonterminal.

The new nonterminal type is encoded by the Lean type `option (`$\mathtt{g}_1$`.nt ⊕ `$\mathtt{g}_2$`.nt) ⊕ (T ⊕ T)`. The new distinguished nonterminal becomes the initial symbol of `g`.

In this way, we ensure that the nonterminals used by `g` to simulate $\mathtt{g}_1$ are disjoint from the nonterminals used by `g` to simulate $\mathtt{g}_2$. There are still real terminals used by both grammars, but `g` never has these terminals on the left-hand side of a rule, since the rewrite rules of `g` consist of

- the rewrite rules of $\mathtt{g}_1$ with all nonterminals mapped to the new nonterminal type and all terminals replaced by proxy nonterminals of the first kind;
- the rewrite rules of $\mathtt{g}_2$ with all nonterminals mapped to the new nonterminal type and all terminals replaced by proxy nonterminals of the second kind;
- for every terminal from `T`, a rule that rewrites the proxy nonterminal of the first kind to the corresponding terminal and a rule that rewrites the proxy nonterminal of the second kind to the corresponding terminal;
- a special rule that rewrites `g.initial` to a two-symbol string `[`$\mathtt{g}_1$`.initial, `$\mathtt{g}_2$`.initial]` wrapped to use the new nonterminal type.

Using this construction, we ensure that all rules of `g` avoid matching strings on the boundary of the concatenation.

Proving that `g` generates a superset of $\mathtt{L}_1$ `*` $\mathtt{L}_2$ is easy because we can apply the rewrite rules in the following order, regardless of the languages:

(1) use the special rule to obtain `[`$\mathtt{g}_1$`.initial, `$\mathtt{g}_2$`.initial]` with the necessary wrapping;
(2) generate the string of proxy nonterminals corresponding to the word from $\mathtt{L}_1$ while $\mathtt{g}_2$`.initial` remains unchanged;
(3) replace all proxy nonterminals of the first kind by the corresponding terminals, which results in deriving a word from $\mathtt{L}_1$ followed by $\mathtt{g}_2$`.initial` as the last symbol;
(4) generate the string of proxy nonterminals corresponding to the word from $\mathtt{L}_2$ while the first part of the string remains unchanged;
(5) replace all proxy nonterminals of the second kind by the corresponding terminals, which results in deriving a word from $\mathtt{L}_2$ that follows the word from $\mathtt{L}_1$ obtained before.

Step (1) is trivial. Steps (2) and (4) are done by following the derivations by $\mathtt{g}_1$ and $\mathtt{g}_2$, respectively. Steps (3) and (5) are straightforward proofs by induction.

Proving that `g` generates a subset of $\mathtt{L}_1$ `*` $\mathtt{L}_2$ is much harder because we do not know in which order the rules of `g` are applied. We had to come up with an invariant that relates intermediate strings derived by `g` to strings that can be derived by $\mathtt{g}_1$ and $\mathtt{g}_2$ from their respective initial symbols.

Very roughly speaking, we prove that there are strings `x` and `y` for every string `w` that `g` can derive, such that the grammar $\mathtt{g}_1$ can derive `x`, the grammar $\mathtt{g}_2$ can derive `y`, and `x ++ y` corresponds to `w`. As usual, we employ structural induction. Looking at the last rule `g` used,

we update `x` or `y` or neither. In particular, we want to point out the following declarations in the formalization:

- function `nst` provides the new symbol type which `g` operates with;
- functions `wrap_symbol`$_1$ and `wrap_symbol`$_2$ convert symbols for use by `g`;
- relation `corresponding_strings` built on top of relation `corresponding_symbols` is used to define how the strings `x` and `y` are precisely related to `w` after each step by `g`;
- lemma `induction_step_for_lifted_rule_from_g`$_1$ characterizes the `x` update;
- lemma `induction_step_for_lifted_rule_from_g`$_2$ characterizes the `y` update;
- lemma `big_induction` states the invariant for proving the hard direction;
- lemma `in_concatenated_of_in_big` puts the proof of the hard direction together.

Note that the added rules have only one symbol on the left-hand side. Therefore, if the two original grammars are context-free, our constructed grammar is also context-free. We thereby obtain, as a bonus, a proof that context-free languages are closed under concatenation. It is implemented in a similar fashion to the proof that context-free languages are closed under union.

6 Closure under Kleene Star

In this section, we prove the following theorem:

```
theorem T0_of_star_T0 (L : language T) :
  is_T0 L → is_T0 L.star
```

This is usually demonstrated by a hand-waving argument about a two-tape nondeterministic Turing machine. The language to be iterated is given by a single-tape (nondeterministic) Turing machine. The new machine scans the input on the first tape, copying it onto the second tape as it progresses, and nondeterministically chooses where the first word ends. Next, the original machine is simulated on the second tape. If the simulated machine accepts the word on the second tape, the process is repeated with the current position of the first head instead of returning to the beginning of the input. Finally, when the first head reaches the end of the input, the second tape contains a suffix of the first tape. The original machine is simulated once more on the second tape. If it accepts, the new machine accepts.

Unfortunately, we did not find any proof based on grammars in the literature. Therefore, we had to invent our own construction. In Section 6.1, we present the construction and the idea underlying its correctness using traditional mathematical notation. In Section 6.2, we comment on its formalization.

6.1 Proof Sketch

Let $L \subseteq T^*$ be a language generated by the grammar $G = (N, T, P, S)$. We construct a grammar $G_* = (N_*, T, P_*, Z)$ to generate the language L^*. The new nonterminal set

$$N_* = N \cup \{Z, \#, R\}$$

expands N with three additional nonterminals: a new starting symbol (Z), a delimiter ($\#$), and a marker for final rewriting (R). The new set of rules is

$$P_* = P \cup \{Z \to ZS\#,\ Z \to R\#,\ R\# \to R,\ R\# \to \epsilon\} \cup \{Rt \to tR \mid t \in T\}$$

Intuitively, $\#$ builds compartments that isolate the words from the language L, and then R acts as a cleaner that traverses the string from beginning to end and removes the compartment delimiters $\#$, thereby ensuring that only terminals are present to the left of R.

To see how G_* works, consider the following grammar over $T = \{a, b\}$. Let $N = \{S\}$ and $P = \{S \to aSb,\ S \to \epsilon\}$. The set of rules becomes

$$P_* = \{S \to aSb,\ S \to \epsilon,\ Z \to ZS\#,\ Z \to R\#,\ R\# \to R,\ R\# \to \epsilon,\ Ra \to aR,\ Rb \to bR\}$$

The following is an example of G_* derivation:

$$Z \Rightarrow ZS\# \Rightarrow ZS\#S\# \Rightarrow ZaSb\#S\# \Rightarrow ZaaSbb\#S\# \Rightarrow ZS\#aaSbb\#S\# \Rightarrow$$
$$ZaSb\#aaSbb\#S\# \Rightarrow ZaSb\#aaaSbbb\#S\# \Rightarrow ZaSb\#aaabbb\#S\# \Rightarrow$$
$$R\#aSb\#aaabbb\#S\# \Rightarrow RaSb\#aaabbb\#S\# \Rightarrow aRSb\#aaabbb\#S\# \Rightarrow$$
$$aRb\#aaabbb\#S\# \Rightarrow abR\#aaabbb\#S\# \Rightarrow abRaaabbb\#S\# \Rightarrow$$
$$abaRaabbb\#S\# \Rightarrow abaaRabbb\#S\# \Rightarrow abaaaRbbb\#S\# \Rightarrow$$
$$abaaabRbb\#S\# \Rightarrow abaaabRbb\#aSb\# \Rightarrow abaaabbRb\#aSb\# \Rightarrow$$
$$abaaabbRb\#ab\# \Rightarrow abaaabbbR\#ab\# \Rightarrow abaaabbbRab\# \Rightarrow$$
$$abaaabbbaRb\# \Rightarrow abaaabbbabR\# \Rightarrow abaaabbbab$$

▶ **Lemma 1.** *Let $w_1, w_2, \ldots, w_n \in L$. Then G_* can derive $Zw_1\#w_2\#\ldots w_n\#$.*

Proof. By induction on n. The base case $Z \Rightarrow^* Z$ is trivial.

Now assume $Z \Rightarrow^* Zw_1\#w_2\#\ldots w_n\#$ and $S \Rightarrow^* w_{n+1}$. We start with the rule $Z \to ZS\#$. We observe $ZS\# \Rightarrow^* Zw_1\#w_2\#\ldots w_n\#S\# \Rightarrow^* Zw_1\#w_2\#\ldots w_n\#w_{n+1}\#$. By transitivity, we obtain $Z \Rightarrow^* Zw_1\#w_2\#\ldots w_n\#w_{n+1}\#$. ◀

From now on, let $[m]$ denote the set of m natural numbers $\{1, 2, \ldots, m\}$.

▶ **Lemma 2.** *If $\alpha \in (T \cup N)^*$ can be derived by G_*, then one of these conditions holds:*

1. $\exists x_1, x_2, \ldots, x_m \in (T \cup N)^*\ (\forall i \in [m]\ (S \Rightarrow^* x_i)\ \wedge\ \alpha = Zx_1\#x_2\#\ldots x_m\#)$;
2. $\exists x_1, x_2, \ldots, x_m \in (T \cup N)^*\ (\forall i \in [m]\ (S \Rightarrow^* x_i)\ \wedge\ \alpha = R\#x_1\#x_2\#\ldots x_m\#)$;
3. $\exists w_1, w_2, \ldots, w_n \in L\ (\exists \beta \in T^*\ (\exists \gamma, x_1, x_2, \ldots, x_m \in (T \cup N)^*$
   $(S \Rightarrow^* \beta\gamma\ \wedge\ \forall i \in [m]\ (S \Rightarrow^* x_i)\ \wedge\ \alpha = w_1w_2\ldots w_n\,\beta R\gamma\#x_1\#x_2\#\ldots x_m\#)))$;
4. $\alpha \in L^*$;
5. $\exists \sigma \in (T \cup N)^*\ (\alpha = \sigma R)$;
6. $\exists \omega \in (T \cup N \cup \{\#\})^*\ (\alpha = \omega\#)$.

In the example above, case 1 arises when $\alpha = ZaaSbb\#S\#$. We can check that $m = 2$, $x_1 = aaSbb$, $x_2 = S$, and condition 1 holds.

In the example above, case 2 arises when $\alpha = R\#aSb\#aaabbb\#S\#$. We can check that $m = 3$, $x_1 = aSb$, $x_2 = aaabbb$, $x_3 = S$, and condition 2 holds.

In the example above, case 3 arises when $\alpha = abaaabRbb\#aSb\#$. We can check that $n = 1$, $m = 1$, $w_1 = ab$, $\beta = aaab$, $\gamma = bb$, $x_1 = aSb$, and condition 3 holds.

Case 4 arises only at the end of a successful computation, which is $\alpha = abaaabbbab$ in the example above.

The remaining two cases do not arise in the example above because they describe an unsuccessful computation (like taking a one-way street ending in a blind alley).

Case 5 arises if the rule $R\# \to R$ is used in the final position (where $R\# \to \epsilon$ should be used instead). The nonterminal R in the final position prevents the derivation from terminating.

Case 6 arises if the rule $R\# \to \epsilon$ is used too early (that is, anywhere but the final $\#$ position). The nonterminal $\#$ in the final position during the absence of R and Z in α prevents the derivation from terminating.

Proof. By induction on G_* derivation steps. The base case $\alpha = Z$ satisfies condition 1 by setting $m = 0$.

Now assume $Z \Rightarrow^* \alpha \Rightarrow \alpha'$ and proceed by case analysis on the conditions.

1. $\exists x_1, x_2, \ldots, x_m \in (T \cup N)^* \, (\forall i \in [m] \, (S \Rightarrow^* x_i) \ \wedge \ \alpha = Z x_1 \# x_2 \# \ldots x_m \#)$:
   - If $\alpha \Rightarrow \alpha'$ used a rule from P, it could be applied only in some x_i. Hence $S \Rightarrow^* x_i \Rightarrow x_i'$, so the same condition holds after replacing x_i by x_i'.
   - If $\alpha \Rightarrow \alpha'$ used the rule $Z \to ZS\#$, it was applied at the beginning of α. Therefore, we set $m' := m + 1$, we set $x_1' := S$, and we increase all indices by one, that is, $x_2' := x_1$, $x_3' := x_2$, $\ldots$, $x_{m'}' := x_m$. The same condition holds.
   - If $\alpha \Rightarrow \alpha'$ used the rule $Z \to R\#$, we keep all variables the same and condition 2 holds.
   - The rules $R\# \to R$, $R\# \to \epsilon$, and $Rt \to tR$ are not applicable (since α does not contain R).
2. $\exists x_1, x_2, \ldots, x_m \in (T \cup N)^* \, (\forall i \in [m] \, (S \Rightarrow^* x_i) \ \wedge \ \alpha = R\# x_1 \# x_2 \# \ldots x_m \#)$:
   - If $\alpha \Rightarrow \alpha'$ used a rule from P, it could be applied only in some x_i. Hence $S \Rightarrow^* x_i \Rightarrow x_i'$, so the same condition holds after replacing x_i by x_i'.
   - The rules $Z \to ZS\#$ and $Z \to R\#$ are not applicable (since α does not contain Z).
   - If $\alpha \Rightarrow \alpha'$ used the rule $R\# \to R$, it was applied at the beginning of α. If $m = 0$, condition 5 holds (a dead end). Otherwise, we set $m' := m - 1 \geq 0$ and $\gamma := x_1$, and we decrease all indices by one, that is, $x_1' := x_2$, $x_2' := x_3$, $\ldots$, $x_{m'}' := x_m$. Since there is nothing before the nonterminal R, we set $n := 0$ and $\beta := \epsilon$. Now, condition 3 holds.
   - If $\alpha \Rightarrow \alpha'$ used the rule $R\# \to \epsilon$ then: if $m = 0$, we obtain the empty word (which belongs to L^*, satisfying condition 4); if $m > 0$, condition 6 holds (because # remained at the end of α'; at the same time R disappeared, and Z did not appear).
   - The rule $Rt \to tR$ is not applicable (the only R in α is immediately followed by #).
3. $\exists w_1, w_2, \ldots, w_n \in L \, (\exists \beta \in T^* \, (\exists \gamma, x_1, x_2, \ldots, x_m \in (T \cup N)^*$
   $(S \Rightarrow^* \beta\gamma \ \wedge \ \forall i \in [m] \, (S \Rightarrow^* x_i) \ \wedge \ \alpha = w_1 w_2 \ldots w_n \, \beta R \gamma \# x_1 \# x_2 \# \ldots x_m \#)))$:
   - If $\alpha \Rightarrow \alpha'$ used a rule from P, it could be applied in γ or in some x_i. In the first case, $\gamma \Rightarrow \gamma'$ implies $\beta\gamma \Rightarrow \beta\gamma'$, hence $S \Rightarrow^* \beta\gamma \Rightarrow \beta'\gamma'$. In the remaining cases, we observe $S \Rightarrow^* x_i \Rightarrow x_i'$ as we did at the beginning of our case analysis. As a result, the same condition still holds.
   - The rules $Z \to ZS\#$ and $Z \to R\#$ are not applicable (α does not contain Z).
   - If $\alpha \Rightarrow \alpha'$ used the rule $R\# \to R$, then γ must have been empty. If $m = 0$, condition 5 holds (a dead end). Otherwise, we set $n' := n + 1$, $w_{n'} := \beta$, $\beta' := \epsilon$, $\gamma' := x_1$, and $m' := m - 1$, and we decrease the indices of x_i by one, that is, $x_1' := x_2$, $x_2' := x_3$, $\ldots$, $x_{m'}' := x_m$. Since $w_{n'} = \beta = \beta\gamma \in T^*$ and $S \Rightarrow^* \beta\gamma$, we have $w_{n'} \in L$. The same condition holds.
   - If $\alpha \Rightarrow \alpha'$ used the rule $R\# \to \epsilon$, then γ must have been empty. If $m = 0$, we get $\alpha = w_1 w_2 \ldots w_n \beta$ and $\beta \in L$; hence condition 4, $\alpha' \in L^*$, is satisfied. If $m > 0$, condition 6 now holds (because # remained at the end of α'; at the same time R disappeared, and Z did not appear).
   - If $\alpha \Rightarrow \alpha'$ used a rule of the form $Rt \to tR$ ($t \in T$), we have $\delta \in (T \cup N)^*$ such that $\gamma = t\delta$. We put $\beta' := \beta t$ and $\gamma' := \delta$. Since $\beta\gamma = \beta t \delta = \beta'\gamma'$, the same condition holds.
4. $\alpha \in L^*$:
   - No rule is applicable (since α contains only terminals). The step $\alpha \Rightarrow \alpha'$ cannot have happened.
5. $\exists \sigma \in (T \cup N)^* \, (\alpha = \sigma R)$:
   - No matter which rule was applied, it happened within σ. No rule could match the final R. The same condition holds for $\alpha' = \sigma' R$.

6. $\exists \omega \in (T \cup N \cup \{\#\})^* \, (\alpha = \omega\#)$:
   - If $\alpha \Rightarrow \alpha'$ used a rule from P, the same condition still holds because $\#$ is not on the left-hand side of any rule from P and neither Z nor R is on the right-hand side of any rule from P.
   - The rules $Z \to ZS\#$, $Z \to R\#$, $R\# \to R$, $R\# \to \epsilon$, and $Rt \to tR$ are not applicable (since α contains neither Z nor R). ◀

▶ **Theorem 3.** *The class of type-0 languages is closed under the Kleene star.*

Proof. We need to show that the language of G_* equals L^*. We prove two inclusions.

For "$\supseteq L^*$", we use Lemma 1. If $w \in L^*$, there exist words $w_1, w_2, \ldots, w_n \in L$ such that $w_1 w_2 \ldots w_n = w$. We see $Z w_1 \# w_2 \# \ldots w_n \# \Rightarrow R \# w_1 \# w_2 \# \ldots w_n \#$. Since all words w_i are made of terminals only, by repeated application of $R\# \to R$ and $Rt \to tR$ (for all $t \in T$) we get $R \# w_1 \# w_2 \# \ldots w_n \# \Rightarrow^* w_1 w_2 \ldots w_n R \#$. Finally, $w_1 w_2 \ldots w_n R\# \Rightarrow w_1 w_2 \ldots w_n$ is obtained by the rule $R\# \to \epsilon$. We conclude that G_* generates w.

For "$\subseteq L^*$", we use Lemma 2 and observe that if G_* generates $\alpha \in T^*$, then $\alpha \in L^*$ because all the remaining cases require α to contain a nonterminal. ◀

6.2 Formalization

The formalization closely follows the proof sketch. The main difference between the two is that where the proof sketch states that an expression belongs to a set, the formalization specifies a type for a term and sometimes a condition that further restricts the term's values.

Lemma 1 is implemented by lemma `short_induction`, which takes w in reverse order for technical reasons. Its proof uses the `lifted_grammar` approach outlined in Section 3. The part $R \# w_1 \# w_2 \# \ldots w_n \# \Rightarrow^* w_1 w_2 \ldots w_n R \#$ is implemented by lemma `terminal_scan_ind`, which employs a nested induction to pass R to the right. The final step of the easy direction is performed inside the theorem `T0_of_star_T0` itself.

Lemma 2 is implemented by lemma `star_induction`, whose formal proof spans over 3000 lines. The base case is discharged immediately. For the induction step, we developed six lemmas `star_case_1` to `star_case_6` distinguished by which of the six conditions α satisfies. In each of them, except for `star_case_4`, which took only four lines to prove, we perform a case analysis on which rule was used for the $\alpha \Rightarrow \alpha'$ transition.

For each case, unless a short ex-falso-quodlibet proof suffices, we need to narrow down where in α the rule could be applied. This analysis is challenging for the rules that were inherited from the original grammar. Consider `case_1_match_rule`, where the informal argument literally says: "If $\alpha \Rightarrow \alpha'$ used a rule from P, it could be applied only in some x_i."

It turns out that this deduction is so complicated that it was worth creating an auxiliary lemma `cases_1_and_2_and_3a_match_aux` to detach the head Z from α and perform the analysis on $x_1 \# x_2 \# \ldots x_m \#$ in order to make the proof easier. As a useful side effect, the auxiliary lemma becomes applicable to similar situations in `star_case_2` and `star_case_3`, as shown in `case_2_match_rule` and `case_3_match_rule`, where more adaptations are needed but the same core argument is used.

From a formal point of view, we abused the symbol "…" in the proof sketch. Replacing it by a formal statement usually leads to `list.join` of `list.map` of something. For example, compare case 1 in the proof sketch

$$\exists x_1, x_2, \ldots, x_m \in (T \cup N)^* \, (\forall i \in [m] \, (S \Rightarrow^* x_i) \ \wedge \ \alpha = Z x_1 \# x_2 \# \ldots x_m \#)$$

to its formal counterpart:

```
∃ x : list (list (symbol T g.nt)),
  (∀ xᵢ ∈ x, grammar_derives g [symbol.nonterminal g.initial] xᵢ) ∧
  (α = [Z] ++ list.join
    (list.map (++ [H]) (list.map (list.map wrap_sym) x)))
```

The nonterminal # is represented by the letter `H` in the code. Notice how easy it is to write the quantification $\exists x_1, x_2, \ldots, x_m \in (T \cup N)^*$ in Lean. The part $\forall i \in [m]\ (S \Rightarrow^* x_i)$ is also elegant. However, the expression $Zx_1\#x_2\#\ldots x_m\#$ leads to a fairly complicated Lean term.

Because many lemmas need to work with expressions like the above, it is important to master how to manipulate terms that combine `list.join` with other functions. For example, the following lemma is useful:

```
lemma append_join_append {s : list α} (L : list (list α)) :
  s ++ (list.map (λ l, l ++ s) L).join =
       (list.map (λ l, s ++ l) L).join ++ s
```

This lemma allows us to move the parentheses in $s(l_1s)(l_2s)\ldots(l_ns)$ to get $(sl_1)(sl_2)\ldots(sl_n)s$ and vice versa.

Working with expressions such as $Zx_1\#x_2\#\ldots x_m\#$ is tedious in Lean. We see this, however, not as a weakness of Lean but rather as an indication that the "…" notation is highly informal. Mathematical expressions with "…" tend to be ambiguous and require the reader's cooperation to make sense of them. In the absence of support for "…" in the proof assistant [19], it is natural that formalizing such expressions leads to verbose code.

In contrast to concatenation, the above proof cannot be reused to establish the closure of context-free languages under the Kleene star because our construction adds rules with two symbols on their left-hand side. However, there exists an easier construction for context-free languages that could be formalized separately if desired.

7 Related Work

To our knowledge, no one has formalized general grammars before. Context-free grammars were formalized by Carlson et al. [5] using Mizar, by Minamide [23] using Isabelle/HOL, by Barthwal and Norrish [4] using HOL4, by Firsov and Uustalu [11] using Agda, and by Ramos [25] using Coq.

Finite automata have often been subjected to verification. In particular, Thompson and Dillies [22] formalized finite automata, which recognize regular languages, using Lean. Thomson [22] also formalized regular expressions, which recognize regular languages as well.

There is ample verification work also for other models of computation:

- Turing machines were formalized using Mizar [7], Matita [2], Isabelle/HOL [26], Lean [6], Coq [15], and recently again Isabelle/HOL [3]. Of these, the most impressive development is probably the last one, by Balbach. It uses multi-tape Turing machines and culminates with a proof of the Cook–Levin theorem, which states that SAT is **NP**-complete.
- The λ-calculus was formalized by Norrish [24] using HOL4 and later by Forster, Kunze, and their colleagues [16, 20, 12, 13, 14, 17] using Coq. The latter group of authors proposed an untyped call-by-value λ-calculus as a convenient basis for computability and complexity theory because it naturally supports compositionality.
- The partial recursive functions were formalized by Norrish [24] using HOL4 and by Carneiro [6] using Lean.
- Random access machines were formalized by Coen [8] using Coq.

8 Conclusion

We defined general grammars in Lean and used them to establish closure properties of recursively enumerable or type-0 languages. We found that closure under union and reversal were straightforward to formally prove, but had to invest considerable effort to prove closure under concatenation and the Kleene star. Despite the tedium of some of the proofs, we believe that grammars are probably a more convenient formalism than Turing machines for showing closure properties. On the other hand, since grammars do not define any of the important complexity classes (such as **P**), formalization of Turing machines and other computational models is needed to further develop the formal theory of computer science.

As future work, results about context-sensitive, context-free, and regular grammars could be incorporated into our library. A comprehensive Lean library encompassing the entire Chomsky hierarchy would be valuable. We already have some results about context-free grammars, and the `mathlib` results about regular languages could be connected to our library. As a more ambitious goal, we might attempt to prove the equivalence between general grammars and Turing machines.

References

1 Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice Hall, 1st edition, 1972.

2 Andrea Asperti and Wilmer Ricciotti. Formalizing Turing Machines. In *Logic, Language, Information and Computation*, volume 7456 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2012. doi:10.1007/978-3-642-32621-9_1.

3 Frank J. Balbach. The Cook-Levin theorem. *Archive of Formal Proofs*, 2023. , Formal proof development. URL: https://isa-afp.org/entries/Cook_Levin.html.

4 Aditi Barthwal and Michael Norrish. Mechanisation of PDA and Grammar Equivalence for Context-Free Languages. In Anuj Dawar and Ruy de Queiroz, editors, *WoLLIC 2010*, volume 6188 of *Lecture Notes in Computer Science*, pages 125–135. Springer, 2010. doi:10.1007/978-3-642-13824-9_11.

5 Patricia L. Carlson, Grzegorz Bancerek, and Im Pan. Context-Free Grammar — Part 1. *J. Formaliz. Math.*, 1992. URL: http://mizar.org/JFM/pdf/lang1.pdf.

6 Mario Carneiro. Formalizing Computability Theory via Partial Recursive Functions. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *ITP 2019*, volume 141 of *LIPIcs*, pages 12:1–12:17. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ITP.2019.12.

7 Jing-Chao Chen and Yatsuka Nakamura. Introduction to Turing Machines. *J. Formaliz. Math.*, 9(4), 2001. URL: https://fm.mizar.org/2001-9/pdf9-4/turing_1.pdf.

8 Claudio Sacerdoti Coen. A Constructive Proof of the Soundness of the Encoding of Random Access Machines in a Linda Calculus with Ordered Semantics. In Carlo Blundo and Cosimo Laneve, editors, *ICTCS 2003*, volume 6188 of *Lecture Notes in Computer Science*, pages 37–57. Springer, 2003. doi:10.1007/978-3-540-45208-9_5.

9 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, *CADE-25*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015. doi:10.1007/978-3-319-21401-6_26.

10 Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A Metaprogramming Framework for Formal Verification. *Proc. ACM Program. Lang.*, 1(ICFP):1–29, 2017. doi:10.1145/3110278.

11 Denis Firsov and Tarmo Uustalu. Certified Normalization of Context-Free Grammars. In *CPP 2015*, pages 167–174. ACM, 2015. doi:10.1145/2676724.2693177.

12 Yannick Forster and Fabian Kunze. A Certifying Extraction with Time Bounds from Coq to Call-By-Value Lambda Calculus. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *ITP 2019*, volume 141 of *LIPIcs*, pages 17:1–17:19. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ITP.2019.17.

13 Yannick Forster, Fabian Kunze, and Marc Roth. The Weak Call-by-Value Lambda-Calculus is Reasonable for Both Time and Space. *Proc. ACM Program. Lang.*, 4(POPL):27:1–27:23, 2019. doi:10.1145/3371095.

14 Yannick Forster, Fabian Kunze, Gert Smolka, and Maximilian Wuttke. A Mechanised Proof of the Time Invariance Thesis for the Weak Call-By-Value λ-Calculus. In Liron Cohen and Cezary Kaliszyk, editors, *ITP 2021*, volume 193 of *LIPIcs*, pages 19:1–19:20. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ITP.2021.19.

15 Yannick Forster, Fabian Kunze, and Maximilian Wuttke. Verified Programming of Turing Machines in Coq. In *CPP 2020*, pages 114–128. ACM, 2020. doi:10.1145/3372885.3373816.

16 Yannick Forster and Gert Smolka. Weak Call-by-Value Lambda Calculus as a Model of Computation in Coq. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *ITP 2017*, volume 10499 of *Lecture Notes in Computer Science*, pages 189–206. Springer, 2017. doi:10.1007/978-3-319-66107-0_13.

17 Lennard Gäher and Fabian Kunze. Mechanising Complexity Theory: The Cook-Levin Theorem in Coq. In Liron Cohen and Cezary Kaliszyk, editors, *ITP 2021*, volume 193 of *LIPIcs*, pages 20:1–20:18. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ITP.2021.20.

18 John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson/Addison Wesley, 3rd edition, 2007.

19 Fulya Horozal, Florian Rabe, and Michael Kohlhase. Flexary Operators for Formalized Mathematics. In Stephen M. Watt, James H. Davenport, Alan P. Sexton, Petr Sojka, and Josef Urban, editors, *Intelligent Computer Mathematics*, Lecture Notes in Computer Science, pages 312–327. Springer, 2014. doi:10.1007/978-3-319-08434-3_23.

20 Fabian Kunze, Gert Smolka, and Yannick Forster. Formal Small-Step Verification of a Call-by-Value Lambda Calculus Machine. In Sukyoung Ryu, editor, *APLAS 2018*, volume 11275 of *Lecture Notes in Computer Science*, pages 264–283. Springer, 2018. doi:10.1007/978-3-030-02768-1_15.

21 Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990. URL: https://era.ed.ac.uk/bitstream/handle/1842/12487/Luo1990.Pdf.

22 The `mathlib` Community. The Lean Mathematical Library. In Jasmin Blanchette and Cătălin Hriţcu, editors, *CPP 2020*, pages 367–381. ACM, 2020. doi:10.1145/3372885.3373824.

23 Yasuhiko Minamide. Verified Decision Procedures on Context-Free Grammars. In Klaus Schneider and Jens Brandt, editors, *TPHOLs 2007*, volume 4732 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2007. doi:10.1007/978-3-540-74591-4_14.

24 Michael Norrish. Mechanised Computability Theory. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *ITP 2011*, volume 6898 of *Lecture Notes in Computer Science*, pages 297–311. Springer, 2011. doi:10.1007/978-3-642-22863-6_22.

25 Marcus Vinícius Midena Ramos. Formalization of Context-Free Language Theory. *Bull. Symbol. Log.*, 25(2):214–214, 2019. doi:10.1017/bsl.2019.3.

26 Jian Xu, Xingyuan Zhang, and Christian Urban. Mechanising Turing Machines and Computability Theory in Isabelle/HOL. In *Interactive Theorem Proving*, volume 7998, pages 147–162. Springer Berlin Heidelberg, 2013. Lecture Notes in Computer Science. doi:10.1007/978-3-642-39634-2_13.

Formalising Yoneda Ext in Univalent Foundations

Jarl G. Taxerås Flaten
University of Western Ontario, London, Ontario, Canada

Abstract

Ext groups are fundamental objects from homological algebra which underlie important computations in homotopy theory. We formalise the theory of Yoneda Ext groups [12] in homotopy type theory (HoTT) using the Coq-HoTT library [3]. This is an approach to Ext which does not require projective or injective resolutions, though it produces large abelian groups. Using univalence, we show how these Ext groups can be naturally represented in HoTT. We give a novel proof and formalisation of the usual six-term exact sequence via a fibre sequence of 1-types (or groupoids), along with an application. In addition, we discuss our formalisation of the contravariant long exact sequence of Ext, an important computational tool. Along the way we implement and explain the Baer sum of extensions and how Ext is a bifunctor.

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

Keywords and phrases homotopy type theory, homological algebra, Yoneda Ext, formalisation, Coq

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.16

Related Version *Full Version*: `https://arxiv.org/abs/2302.12678`

Supplementary Material *Software (Source Code)*: `https://github.com/HoTT/HoTT/tree/master/theories/Algebra/AbSES`
Software (Source Code): `https://github.com/jarlg/Yoneda-Ext`
archived at `swh:1:dir:636630073a835e6cd355b5cee34fa839986ff73d`

Funding We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), RGPIN-2022-04739.

Acknowledgements The theory of Yoneda Ext in HoTT is joint work with Dan Christensen, to whom I am also grateful for discussions and contributions related to the formalisation. I thank Jacob Ender for contributions to the formalisation of the Baer sum, and the collaborators of the Coq-HoTT library – and Ali Caglayan in particular – for their review of, and contributions to, the various pull requests related to this project. I also thank the anonymous reviewers for valuable feedback.

1 Introduction

The field of homotopy type theory (HoTT) lies at the intersection of type theory and algebraic topology, and serves as a bridge to transfer tools and insights from one domain to the other. In one direction, the formalism of type theory has proven to be a powerful language for reasoning about some of the highly coherent structures occurring in branches of modern algebraic topology. Several of these structures are "natively supported" by HoTT, and we can reason about them much more directly than in classical set-based approaches. This makes HoTT an ideal language in which to formalise results and structures from algebraic topology. Moreover, theorems in HoTT are valid in any ∞-*topos*, not just for ordinary spaces. In a joint paper with Dan Christensen [2], we interpret our constructions into an ∞-topos, and explain the relation between our Ext groups and *sheaf Ext*.

We present a formalisation of Ext groups in HoTT following the approach of Yoneda [12, 13]. Ext groups are fundamental objects in homological algebra, and they permeate computations in homotopy theory. For example, the universal coefficient theorem relates Ext


© Jarl G. Taxerås Flaten;
licensed under Creative Commons License CC-BY 4.0

groups and cohomology, and features in the classical proof that $\pi_5(S^3) \simeq \mathbb{Z}/2$. Much of our formalisation has already been accepted into the Coq-HoTT library under the `Algebra.AbSES` namespace, though we have also contributed to other parts of the library throughout this project. The long exact sequence, along with a few other results we need, are currently in a separate repository named `Yoneda-Ext`. We supply links to formalised statements using a trailing $^\diamond$-sign throughout.

In ordinary mathematics, Ext groups of modules over a ring are usually defined using projective (or injective) resolutions. This is possible because the axiom of choice implies the existence of such projective resolutions, and Ext groups are independent of any particular choice of resolution. (Similarly, categories of sheaves of modules always admit injective resolutions.) In our setting, however, even abelian groups fail to admit projective resolutions. This stems from the fact that some sets fail to be projective, which may be familiar to those working constructively or internally to a topos. Accordingly, to define Ext groups in homotopy type theory we cannot rely on resolutions. Fortunately, Yoneda [12, 13] gave such a general approach, whose theory is detailed in [6], our main reference. A drawback of this approach is that it produces *large* abelian groups, as we explain in Section 3.1.

We build upon the Coq-HoTT library [3], which contains sophisticated homotopy-theoretic results, but which is presently lacking in terms of "basic" algebra. For this reason, we have opted to simply develop Ext groups of abelian groups, instead of for modules over a ring or in a more general setup. Nevertheless, it is clear that everything we do could have been done over an arbitrary ring, given a well-developed library of module theory. Moreover, we emphasise that higher Ext groups in HoTT are interesting even for abelian groups. While in classical mathematics such Ext groups of abelian groups are trivial in dimension 2 and up, in HoTT they may be nontrivial in all dimensions! This is because there are models of HoTT in which these Ext groups are nontrivial [2].

In Section 3 we explain how univalence lets us naturally represent Yoneda's approach to Ext in HoTT. We construct the type $\mathtt{AbSES}(B, A)$ of short exact sequences between two abelian groups A and B, and define $\mathtt{Ext}^1(B, A)$ to be the set of path-components of $\mathtt{AbSES}(B, A)$. This definition is justified by characterising the paths in $\mathtt{AbSES}(B, A)$, which crucially uses univalence. We also show that the loop space of $\mathtt{AbSES}(B, A)$ is isomorphic to the group $\mathtt{Hom}(B, A)$ of group homomorphisms, and that $\mathtt{Ext}^1(P, A)$ vanishes whenever P is projective, in a sense we define. These results all play a role in the subsequent sections.

The main content of Section 4 is a proof and formalisation of the following:

▶ **Theorem 13.** *Let $A \xrightarrow{i} E \xrightarrow{p} B$ be a short exact sequence of abelian groups. For any abelian group G, pullback yields a fibre sequence: $\mathtt{AbSES}(B, G) \xrightarrow{p^*} \mathtt{AbSES}(E, G) \xrightarrow{i^*} \mathtt{AbSES}(A, G)$.*$^\diamond$

We give a novel, direct proof of this result which requires managing considerable amounts of coherence. The formalisation is done for abelian groups, but the proof applies to modules over a general ring. Its formalisation benefited from the `WildCat` library of Coq-HoTT (see Section 2.2), which makes it convenient to work with types equipped with an imposed notion of paths. This allows us to work with *path data* in $\mathtt{AbSES}(B, A)$ with better computational properties than actual paths, but which correspond to paths via the aforementioned characterisation. From the fibre sequence of the theorem we deduce the usual six-term exact sequence (Proposition 19), which we then use to compute Ext groups of cyclic groups:

$$\mathtt{Ext}^1(\mathbb{Z}/n, A) \;\cong\; A/n$$

for any nonzero $n : \mathbb{N}$ and abelian group A (Corollary 21).$^\diamond$ The six-term exact sequence, along with this corollary, have already been applied in [1]. We also discuss how $\mathtt{Ext}^1$ becomes a bifunctor into abelian groups using the Baer sum.

Finally, in Section 5 we define $\texttt{Ext}^n$ for any $n : \mathbb{N}$ and discuss our formalisation of the long exact sequence, in which the connecting maps are given by *splicing*:$^\diamond$

▶ **Theorem 26.** *Let $A \xrightarrow{i} E \xrightarrow{p} B$ be a short exact sequence of abelian groups. For any abelian group G, there is a long exact sequence by pulling back:*$^{\diamond\diamond\diamond}$

$$\cdots \xrightarrow{i^*} \texttt{Ext}^n(A,G) \xrightarrow{-\circledcirc E} \texttt{Ext}^{n+1}(B,G) \xrightarrow{p^*} \texttt{Ext}^{n+1}(E,G) \xrightarrow{i^*} \cdots .$$

At present, we have only formalised this long exact sequence of *pointed sets*. It remains to construct the Baer sum making $\texttt{Ext}^n$ into an abelian group for $n > 1$, however once this is done then we automatically get a long exact sequence of abelian groups. Our proof follows that of Theorem 5.1 in [6], which is originally due to Stephen Schanuel.

Notation and conventions. We use typewriter font for concepts which are defined in the code, such as `AbSES` and `Ext`. In contrast, when we use normal mathematical font, such as $\mathrm{Ext}^n(B,A)$, we mean the classical notion. For mathematical statements we prefer to stay close to mathematical notation by writing for example $\texttt{Ext}^n(B,A)$ for what means `Ext n B A` in Coq. The symbol $^\diamond$ is used to refer to relevant parts of the code.

Our terminology mirrors that of [10]; in particular we say "path types" for what are also called "identity types" or "equality types". We write `pType` for the universe of pointed types, and `pt` for the base point of a pointed type. The $\equiv$-symbol is for definitional equality.

2 Preliminaries

2.1 Homotopy Type Theory

We briefly explain the formal setup of homotopy type theory along with some basic notions that we need. For a thorough introduction to HoTT, the reader may consult [10, 9].

Homotopy type theory (HoTT) extends Martin–Löf type theory (MLTT) with the *univalence axiom* and often various higher inductive types (HITs). Of the latter, we simply need propositional truncation and set truncation, which we explain in more detail below.

The univalence axiom characterises the identity types of universes. In ordinary MLTT, there is always a function

$$\texttt{idtoequiv} : \prod_{X,Y:\texttt{Type}} (X = Y) \to (X \simeq Y)$$

defined by sending the reflexivity path on a type X to the identity self-equivalence on X, using the induction principle of path types. The univalence axiom asserts that `idtoequiv` is an equivalence for all X and Y. In HoTT, the first thing we often do after defining a new type is to characterise its path types. The univalence axiom does this for the universe.

From univalence, a general *structure identity principle* [10, Chapter 9.8] follows which characterises paths between structured types, such as groups and other algebraic structures. In the case of groups, univalence implies that paths between groups correspond to group isomorphisms. Similarly, paths between modules correspond to module isomorphisms.

Propositions, sets, and groupoids

In HoTT there is a hierarchy of ***n*-truncated types** (or ***n*-types**, for short) for any integer $n \geq -2$. In general, a type X is an $(n+1)$-type when all the path types $x_0 =_X x_1$ are n-types. The recursion starts at -2, when the condition is just that the map $X \to 1$ is an equivalence, and in this case X is **contractible**.

We only deal with the bottom four levels of this hierarchy: **contractible types**, **propositions** ((−1)-types), **sets** (0-types) and 1-types. A type X is a proposition when any two points in X are equal (but there may not be any points). A type X is a set when the path types $x_0 =_X x_1$ are all propositions – this amounts to there being "at most" one path between x_0 and x_1. Lastly, a type X is a 1-type when its path types are sets – in particular, for any $x : X$, the **loop space** $\Omega X :\equiv (x =_X x)$ is a set which is a group under path composition. (We leave base points implicit when taking loop spaces.)

There are truncation operations which create a proposition or a set from a given type X. We denote by $\|X\|$ the propositional truncation, and by $\pi_0 X$ the set truncation (or *set of path-components*) of X. In Coq-HoTT, the corresponding notation is `merely X` and `Tr 0 X`. The map `tr` : $X \to \pi_0 X$ sends a point to its connected component. When we say that a type X **merely** holds, then we mean that its propositional truncation $\|X\|$ holds.

2.2 The Coq-HoTT Library

The Coq-HoTT library [3] is an open-source repository of formalised mathematics in homotopy type theory using Coq. It is particularly aimed at developing synthetic homotopy theory, and includes theory about spheres, loop spaces, classifying spaces, modalities, "wild ∞-categories," and basic results about abelian groups, to mention a few things. The library is part of the *Coq Platform* and is available through the standard `opam` package repositories.

Below we explain some of the main features of this library, and of Coq itself, which are important for the present work.

Universes and cumulativity

We assume basic familiarity with universes and universe levels in Coq, and in particular that they are *cumulative*: a type X : `Type@{u}` can be resized to live in `Type@{v}` under the constraint `u` $\leq$ `v`. (Here `u` and `v` are **universe levels**.) Resizing is done implicitly by Coq.

In the Coq-HoTT library, we additionally make most of our structures cumulative. This essentially means that resizing commutes with the formation of a data structure – i.e., it does not matter whether you resize the inputs to the data structure or whether you resize the resulting data structure. As an example, consider the data structure `prod` which forms the product of two types in a common (for simplicity) universe level. Suppose we have two universe levels `u` and `v` with the constraint `u < v`. Given `X Y : Type@{u}`, we can form the product at level `u` and then resize, or first resize and then form the product. By making `prod` a cumulative data structure, the two results agree (with implicit resizing):

```
prod@{u} X Y  ≡  prod@{v} X Y.
```

Cumulativity of data structures is an essential Coq feature which facilitates the kind of formalisation we do in this paper. For example, it lets us resize groups and homomorphisms. It also lets us reduce the number of universes in some of our definitions via the following trick: instead of having separate universes for different inputs, we can often use a single universe (which represents the maximum) and leverage cumulativity.

We also make use of universe constraints since our constructions move between various universe levels. The constraints both document and verify the mathematical intent.

The `WildCat` library

The `WildCat` namespace contains the development of "wild ∞-categories," functors between such, and related things. This library was spearheaded by Ali Caglayan, tslil clingman, Floris van Doorn, Morgan Opie, Mike Shulman, and Emily Riehl. The concepts generalise those

appearing in [11, Section 4.3.1], and are not currently present in the literature. We explain the basics of this library which are especially relevant for our formalisation.

Starting from the notion of **graph**$^\diamond$ – a type A with a binary operation (or *correspondence*) `Hom` into `Type` – the notion of a **0-functor**$^\diamond$ is that of a homomorphism of graphs:

```
Class IsGraph (A : Type) := { Hom : A -> A -> Type }.
Class Is0Functor {A B : Type} `{IsGraph A} `{IsGraph B} (F : A -> B)
  := { fmap : forall {a b : A} (f : Hom a b), Hom (F a) (F b) }.
```

We will often use the notation `Hom` in this text, leaving the graph structure implicit.

From here one could go ahead and define categories by defining a composition operation and using the identity types of the type $\texttt{Hom}(a, b)$ to express the various laws a category needs to satisfy, such as associativity of composition. A more flexible approach is to instead allow $\texttt{Hom}(a, b)$ to itself be a graph, making A into a **2-graph**.$^\diamond$ This is the approach taken by `WildCat`, and this flexibility is important for our formalisation.

```
Class Is2Graph (A : Type) `{IsGraph A}
  := { isgraph_hom : forall (a b : A), IsGraph (Hom a b) }.
```

For a 2-graph A, a category structure can then be defined in a straightforward manner using `isgraph_hom` to express the various laws that need to hold. This structure is bundled into a class called `Is1Cat`.$^\diamond$ For example, associativity is expressed as follows, using the notation `$==` as a shorthand for the 2-graph structure and `$o` for composition:

```
  cat_assoc : forall (a b c d : A)
    (f : Hom a b) (g : Hom b c) (h : Hom c d),
      (h $o g) $o f $== h $o (g $o f);
```

If all the morphisms in A are invertible, then A is a **groupoid**.$^\diamond$ Finally, for the notion of a **1-functor** between categories we also express the laws using the 2-graph structure.$^\diamond$

```
Class Is1Functor {A B : Type} `{Is1Cat A} `{Is1Cat B}
  (F : A -> B) `{!Is0Functor F} := {
    fmap_id : forall a, fmap F (Id a) $== Id (F a);
    fmap_comp : forall a b c (f : Hom a b) (g : Hom b c),
      fmap F (g $o f) $== fmap F g $o fmap F f;
    fmap2 : forall a b (f g : Hom a b),
      (f $== g) -> (fmap F f $== fmap F g) }.
```

The terms `fmap_id` and `fmap_comp` express that the functor `F` respects identities and composition, as usual. If we had used identity types instead of a 2-graph structure, so that `f $== g` simply meant `f = g`, then `F` would automatically respect equality between morphisms, making `fmap2` redundant. However, in the more general 2-graph setup, this needs to be included as a law.

The adjective "wild" is used for the sort of categories just defined to indicate that they do not capture all the coherence needed to represent ∞-categories, only the 1-categorical structure. However, in our usage we will only encounter genuine 1-categories and groupoids. In particular, any type X defines a groupoid via its identity types$^\diamond$, and if X is a 1-type then this groupoid structure captures everything about X. This enables us to impose our own notion of paths, which we call *path data* below, for certain types of interest.

3 Yoneda Ext

As mentioned in the introduction, we will follow Yoneda's approach to Ext groups [12, 13], which does not require projective (or injective) resolutions, though it produces *large* groups. This approach and related theory is explained in [6], which is our main reference. At present, the Coq-HoTT library – with which this work has been formalised – does not contain much theory related to modules over a general ring (nor the theory of abelian categories, or anything of the sort). We therefore only formalise and state our results for abelian groups. It is clear, however, that everything we say could be done for modules over a general ring.

For the classically-minded reader, let us also emphasise that in homotopy type theory the category of abelian groups does *not* have global dimension 1, so that the higher Ext groups we define in Section 5 do not necessarily vanish.

3.1 The Type of Short Exact Sequences

Given two abelian groups A and B, Yoneda defines a group $\mathrm{Ext}^1(B, A)$ by considering the large set (or class) of all short exact sequences $A \xrightarrow{i} E \xrightarrow{p} B$ and taking a quotient by a certain equivalence relation. The sequence being exact means that i is injective, p is surjective, that $p \circ i = 0$, and that the image of i is equal to the kernel of p. We usually simply write E for the short exact sequence $A \to E \to B$ when no confusion can arise. The equivalence relation which Yoneda quotients out by is defined as "$E \sim F$ if and only if there exists an isomorphism $E \cong F$ which respects the maps from A and to B." Equivalently, but more topologically, one can consider the *groupoid* of short exact sequences $A \to E \to B$ and define $\mathrm{Ext}^1(B, A)$ to be the set of path-components of this groupoid – see, e.g., [6, Chapter III] for details about both of these descriptions.

In homotopy type theory, given two abelian groups A and B we form the **type of short exact sequences** from A to B as the Σ-type over all abelian groups E equipped with an injection $\mathtt{inclusion}_E : A \to E$, a surjection $\mathtt{projection}_E : E \to B$, and a witness that these two maps form an exact complex. We represent this data as the following record-type:$^\diamondsuit$

```
Record AbSES@{u v | u < v} (B A : AbGroup@{u}) : Type@{v} := {
    middle : AbGroup@{u};
    inclusion : Hom A middle;
    projection : Hom middle B;
    isembedding_inclusion : IsEmbedding inclusion;
    issurjection_projection : IsSurjection projection;
    isexact_inclusion_projection
      : IsExact (Tr (-1)) inclusion projection;
  }.
```

Note that $\mathtt{AbSES}(B, A)$ denotes short exact sequences *from A to B*. The abelian group `middle` plays the role of E in the prose above. Here, the condition that $\mathtt{projection}_E \circ \mathtt{inclusion}_E = 0$ is baked into the `IsExact` field, which also expresses exactness.[1] We have included universe annotations which express that E lives in the same universe `u` as the abelian groups A and B. Accordingly, the resulting type $\mathtt{AbSES}(B, A)$ lives in a universe `v` which is strictly greater than `u`, as in Yoneda's construction above. The type $\mathtt{AbSES}(B, A)$ is pointed by the **trivial short exact sequence**$^\diamondsuit$ $A \to A \oplus B \to B$.

[1] The term `Tr (-1)` can safely be ignored; it expresses that the induced map from A to the kernel of $\mathtt{projection}_E$ is (-1)-*connected*, which here just means it is a surjection.

We now define $\texttt{Ext}^1(B, A)$ as the set-truncation of the type of short exact sequences.◇

```
Definition Ext (B A : AbGroup) := Tr 0 (AbSES B A).
```

In Section 3.3 we make the set $\texttt{Ext}^1(B, A)$ into an abelian group via the *Baer sum*. These abelian groups, and their higher variants defined in Section 5, are our main objects of study.

Whenever we define a new type in homotopy type theory, the first thing we often do is to characterise its path types. Theorem 7.3.12 of [10] characterises paths in truncations, yielding

$$\left(|E|_0 =_{\text{Ext}^1} |F|_0\right) \simeq \|E = F\|$$

for any $E, F : \texttt{AbSES}(B, A)$. As such, it suffices to understand paths in $\texttt{AbSES}(B, A)$. These are in turn characterised by Theorem 2.7.2 of loc. cit., which characterises paths in general Σ-types, combined with the fact that paths in `AbGroup` are isomorphisms. In our case, the result is that paths between short exact sequences correspond to isomorphisms between the `middle`s making the appropriate triangles commute. We refer to this data as *path data*, and bundle it into a separate type (where `*` denotes products of types):◇

```
Definition abses_path_data_iso {B A : AbGroup} (E F : AbSES B A)
  := {phi : Iso E F & (phi $o inclusion E == inclusion F)
                    * (projection E == projection F $o phi)}.
```

Here `Iso` forms the type of isomorphisms between two groups. From our discussion above, for any $E, F : \texttt{AbSES}(B, A)$, we get an equivalence of types◇

$$(E =_{\texttt{AbSES}(B,A)} F) \simeq \texttt{abses\_path\_data\_iso}(E, F).$$

However, a bit more can be said: the *short five lemma*◇ implies that if we replace `Iso` by `Hom` above, then it still follows that `phi` is an isomorphism. We define `abses_path_data`◇ as `abses_path_data_iso` above, but with `Hom` in place of `Iso`. It is convenient to have both types around: it is easier to construct an element of `abses_path_data`; however we will see situations later on where it is convenient to keep track of a *specific* inverse to the underlying map, which `abses_path_data_iso` lets us do.

▶ **Definition 1.** *The type* $\texttt{AbSES}(B, A)$ *is a groupoid whose graph structure is given by* `abses_path_data_iso` *and a corresponding category structure. For the* 2*-graph structure, we assert that two path data are equal just when their underlying maps are homotopic.*◇

This definition is justified by the preceding discussion, which yields:

▶ **Lemma 2.** *For any* $E, F : \texttt{AbSES}(B, A)$*, there are equivalences of types*◇

$$(E = F) \simeq \texttt{abses\_path\_data\_iso}(E, F) \simeq \texttt{abses\_path\_data}(E, F).$$

Though elementary, this lemma has an interesting consequence. This statement appears as the $n, i = 1$ case of [8, Theorem 1].

▶ **Proposition 3.** *The loop space of* $\texttt{AbSES}(B, A)$ *is naturally isomorphic to* $\texttt{Hom}(B, A)$.◇

Proof. It suffices, by the previous lemma, to give an isomorphism between $\texttt{Hom}(B, A)$ and $\texttt{abses\_path\_data}(A \oplus B, A \oplus B)$. One can easily check that a map $\phi : A \oplus B \to A \oplus B$ subject to the constraints of path data, is uniquely determined by the composite◇

$$B \to A \oplus B \xrightarrow{\phi} A \oplus B \to A.$$

Moreover, this association defines a group isomorphism – details are in the formalisation.◇ ◀

To formalise the previous proposition, we first developed basic theory about biproducts of abelian groups which now live in `Algebra.AbGroups.Biproduct`.

In ordinary homological algebra, an abelian group P is *projective* if for any homomorphism $f : P \to B$ and epimorphism $p : A \to B$, there exists a *lift* $l : P \to A$ such that $f = e \circ l$. It is well-known that $\mathrm{Ext}^1(P, A)$ always vanishes when P is projective, and that this property characterises projectivity. In our setting, we define an abelian group P to be **projective** if for any homomorphism f and epimorphism p as above, there *merely* exists a lift l such that $f = l \circ l$. The propositional truncation makes this into a *property* of an abelian group, and not a structure. In Coq, we express this as a type-class:◇

```
Class IsAbProjective (P : AbGroup) : Type :=
  isabprojective : forall (A B : AbGroup),
    forall (f : Hom P B), forall (e : Hom A B),
    IsSurjection e -> merely (exists l : P $-> A, f == e $o l).
```

As in the classical case, projectives are characterised by the vanishing of Ext:

▶ **Proposition 4.** *An abelian group P is projective if and only if* $\mathtt{Ext}^1(P, A) = 0$ *for all A.*◇

From the induction principle of $\mathbb{Z}$ it follows that $\mathbb{Z}$ is projective◇ in the sense we defined above. Consequently $\mathtt{Ext}^1(\mathbb{Z}, A) = 0$ for any abelian group A, and we will use this later on.

▶ Remark 5. There is a subtle point related to projectivity that merits discussion. Our definition of projectivity only requires the lift l to *merely* exist (a property), but one could have asked for actual existence (a structure). There is no concept of "mere existence" in ordinary mathematics, and when translating concepts into HoTT we have to carefully choose to make something a structure or a property. In this case, our definition of projectivity is justified by Proposition 4. If we had made projectivity a structure, then not even $\mathbb{Z}$ would be projective, which we need it to be.

3.2 Ext as a Bifunctor

Some of the important structure of Ext^1 is captured by the fact that it defines a *bifunctor* $\mathrm{Ext}^1(-,-) : \mathrm{Ab}^{\mathrm{op}} \times \mathrm{Ab} \to \mathrm{Ab}$. This means that $\mathrm{Ext}^1(-,-)$ is a functor in each variable and that the following "bifunctor law" holds:

$$\mathrm{Ext}^1(f,-) \circ \mathrm{Ext}^1(-,g) = \mathrm{Ext}^1(-,g) \circ \mathrm{Ext}^1(f,-). \tag{1}$$

We added a basic implementation of bifunctors to the `WildCat` library for our purposes, asserting the bifunctor law using the 2-graph structure:◇

```
Class IsBifunctor {A B C : Type} `{IsGraph A, IsGraph B, Is1Cat C}
  (F : A -> B -> C) := {
    bifunctor_isfunctor_10 : forall a, Is0Functor (F a);
    bifunctor_isfunctor_01 : forall b, Is0Functor (fun a => F a b);
    bifunctor_isbifunctor :
      forall a0 a1 (f : Hom a0 a1), forall b0 b1 (g : Hom b0 b1),
        fmap (F _) g $o fmap (flip F _) f
          $== fmap (flip F _) f $o fmap (F _) g }.
```

Here `flip` is the map which reverses the order of arguments of a binary function. We note that in order to state the bifunctor law, we only require `F` to be a 0-functor in each variable. As such we only include those instances in this class.

The bifunctor instance of $\texttt{Ext}^1$ will come from a bifunctor instance of $\texttt{AbSES}$, so we work with the latter. First of all, $\texttt{AbSES} : \texttt{AbGroup}^{\mathrm{op}} \to \texttt{AbGroup} \to \texttt{Type}$ becomes a 0-functor in each variable by pulling back and pushing out, respectively.

▶ **Lemma 6.** *Let $g : B' \to B$ be a homomorphism of abelian groups. For any short exact sequence $A \to E \to B$, we have a short exact sequence $A \to g^*(E) \to B'$.*◇ *Moreover, if E is trivial, then so is the short exact sequence $g^*(E)$.*◇

Dually, one can push out a short exact sequence $A \to E \to B$ along a map $f : A \to A'$ to get a short exact sequence $A' \to f_*(A) \to B$.◇

We supply careful proofs that pushout and pullback respect composition of pointed maps◇ and homotopies between maps,◇ and that pushing out along the identity map gives the pointed identity map.◇ These identities could be shown with shorter proofs, however in Section 4 we will have to prove coherences involving the paths constructed here, and these coherences are simpler to solve when phrased in terms of path data. In any case, these proofs make $\texttt{AbSES}$ into a 1-functor in each variable.◇◇

For the bifunctor law we make use of the following proposition, which is remarkably useful for showing that a given extension is a pullback of another one.

▶ **Proposition 7.** *Suppose given the following diagram with short exact rows:*

$$\begin{array}{ccccc} A & \longrightarrow & E' & \longrightarrow & B' \\ \downarrow{\scriptstyle \alpha} & & \downarrow & & \downarrow{\scriptstyle g} \\ A & \longrightarrow & E & \longrightarrow & B\,. \end{array}$$

If $\alpha = \mathrm{id}$ then the top row is equal to the pullback of the bottom row along g.◇

Proof. Since the right square commutes, we get a map $E' \to g^*(E)$ by the universal property of the pullback. This map respects the inclusions and projections, and therefore defines a path by Lemma 2. ◀

There is a dual statement for pushouts in which the rightmost map must be the identity.◇

▶ **Corollary 8.** *Any diagram with short exact rows as follows yields a path $f_*(E) = g^*(F)$.*◇

$$\begin{array}{ccccc} A & \longrightarrow & E & \longrightarrow & B' \\ \downarrow{\scriptstyle f} & & \downarrow & & \downarrow{\scriptstyle g} \\ A' & \longrightarrow & F & \longrightarrow & B\,. \end{array}$$

The corollary lets us swiftly show bifunctoriality:

▶ **Proposition 9.** *The binary map $\texttt{AbSES} : \texttt{AbGroup}^{\mathrm{op}} \to \texttt{AbGroup} \to \texttt{Type}$ is a bifunctor.*◇

Proof. Consider a short exact sequence $A \to E \to B$ along with two homomorphisms $f : A \to A'$ and $g : B' \to B$. There is an obvious diagram with short exact rows:

$$\begin{array}{ccccc} A & \longrightarrow & g^*(E) & \longrightarrow & B' \\ \downarrow{\scriptstyle f} & & \downarrow & & \downarrow{\scriptstyle g} \\ A' & \longrightarrow & f_*(E) & \longrightarrow & B\,. \end{array}$$

which by the previous corollary yields a path $f_*(g^*(E)) = g^*(f_*(E))$, as required. ◀

▶ Remark 10. The results from Section 3.3 will show that $\texttt{AbSES}$ is an *H-space.*◇ Combining this with [1, Lemma 2.6]◇, we deduce that $\texttt{AbSES}$ is a bifunctor into pointed types. This does not play a role in the rest of this paper, however.

3.3 The Baer Sum

The *Baer sum* is a binary operation on $\mathrm{Ext}^1(B, A)$ which makes it into an abelian group. Given two extensions $E, F : \mathrm{Ext}^1(B, A)$ their Baer sum is defined as

$$E + F :\equiv \Delta^* \nabla_* (E \oplus F)$$

where $E \oplus F$ is the point-wise direct sum, $\nabla(a, b) :\equiv a_0 + a_1 : A \oplus A \to A$ is the codiagonal map, and $\Delta(b) :\equiv (b, b) : B \to B \oplus B$ is the diagonal map.

Together with Dan Christensen and Jacob Ender, we have implemented the Baer sum in `Algebra.AbSES.BaerSum`. We define this operation on the level of short exact sequences and then descend the operation to the set `Ext^1` by truncation-recursion.◇

```
Definition abses_baer_sum `{Univalence} {B A : AbGroup}
  : AbSES B A -> ABSES B A AbSES B A := fun E F =>
      abses_pullback ab_diagonal
        (abses_pushout ab_codiagonal (abses_direct_sum E F)).

Definition baer_sum `{Univalence} {B A : AbGroup}
  : Ext B A -> Ext B A -> Ext B A.
Proof.
  intros E F; strip_truncations.
  exact (tr (abses_baer_sum E F)).
Defined.
```

Above, the `strip_truncations` tactic is a helper for doing truncation-recursion; it lets us assume that both E and F are elements of $\mathtt{AbSES}(B, A)$ in order to map into the set $\mathtt{Ext}^1(B, A)$. We then simply form the Baer sum of E and F on the level of short exact sequences before applying `tr` to the result.

The formalisation that the Baer sum makes $\mathtt{Ext}^1(B, A)$ into an abelian group closely follows the "second proof" of [6, Theorem III.2.1].

▶ **Theorem 11.** *The set* $\mathtt{Ext}^1(B, A)$ *is an abelian group under the Baer sum operation.*◇

The proof can be done entirely by chaining together equations once the bifunctoriality of $\mathtt{Ext}^1$ has been established along with its interaction with direct sums. To illustrate this, we prove that pushouts respect the Baer sum:

▶ **Proposition 12.** *Let* $\alpha : A \to A'$ *be a homomorphism of abelian groups. For any abelian group* B*, pushout defines a group homomorphism* $\alpha_* : \mathtt{Ext}^1(B, A) \to \mathtt{Ext}^1(B, A')$.◇

Proof. Using bifunctoriality of $\mathtt{Ext}^1$ and naturality of $\oplus$, we have:

$$\begin{aligned} \alpha_*(E + F) &= \Delta^*(\alpha_* \nabla_*(E \oplus F)) = \Delta^*(\nabla_*(\alpha_* \oplus \alpha_*)_*(E \oplus F)) \\ &= \Delta^*(\nabla_*(\alpha_* E \oplus \alpha_* F)) \equiv \alpha_* E + \alpha_* F. \end{aligned}$$

◀

Similarly, pullback defines a group homomorphism as well.◇ These results make $\mathtt{Ext}^1$ into a bifunctor valued in abelian groups.◇

4 The Pullback Fibre Sequence

The main goal of this section is to explain and prove the following mathematical result, and to discuss its formalisation◇ along with some applications.

▶ **Theorem 13.** *Let $A \xrightarrow{i} E \xrightarrow{p} B$ be a short exact sequence of abelian groups. For any abelian group G, pullback yields a fibre sequence:* $\mathtt{AbSES}(B, G) \xrightarrow{p^*} \mathtt{AbSES}(E, G) \xrightarrow{i^*} \mathtt{AbSES}(A, G)$.◇

In [2], we give a different proof of this result via an equivalence between $\mathtt{AbSES}(B, A)$ and pointed maps between Eilenberg–Mac Lane spaces. However, this different proof seems to only work over $\mathbb{Z}$ whereas our proof below works for a general ring (though it has only been formalised for $\mathbb{Z}$).

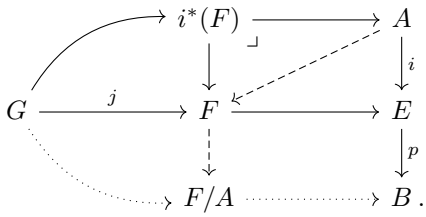
A sequence of pointed maps $F \xrightarrow{i} E \xrightarrow{p} B$ is a **fibre sequence** if $p \circ i$ is pointed-homotopic to the constant map, and the induced map $F \to \mathtt{fib}_p$ is an equivalence. Any fibre sequence induces a long exact sequence of homotopy groups [10, Theorem 8.4.6]:

$$\cdots \to \pi_n(F) \to \pi_n(E) \to \pi_n(B) \to \cdots \to \pi_0(F) \to \pi_0(E) \to \pi_0(B).$$

In the situation of our theorem, it is immediate from functoriality and exactness of E that $i^* \circ p^*$ is constant. Therefore our goal is to show that the induced map $c : \mathtt{AbSES}(B, G) \to \mathtt{fib}_{i^*}$ is an equivalence.[2] We will do this by first constructing a section of c, and then a contraction of the fibres of c to the values of this section. A key part of the formalisation is to work with path data instead of actual paths, since the former has better computational properties. We will simply use $E = F$ to denote path data, and refer to it as such, in this section.

▶ **Lemma 14.** *Let $G \to F \to E$ be a short exact sequence. Given path data $p : i^*(F) = \mathtt{pt}$, we construct a short exact sequence $G \to F/A \to B$.*◇

Proof. The path data p means that the sequence $i^*(F)$ splits. Thus we can form the cokernel F/A as in the diagram:



The two maps $G \to F/A \to B$ are given by composition and the universal property of the cokernel, respectively. It is clear that this forms a complex and that the second map is an epimorphism, since it factors one. To see that the map $G \to F/A$ is an injection, suppose $g : G$ is sent to $0 : F/A$. Then $j(g)$ is in the image of some $a : A$ by $A \to F$. But the map $i^*(F) \to F$ is an injection, being the pullback of one, and so using the path data we get an equality $(g, 0) = (0, a)$ in $G \oplus A$. Of course, this implies that $g = 0$, as required.

Exactness of $G \to F/A \to B$ follows from a straightforward diagram chase. ◀

The diagram above exhibits F as the pullback of F/A along p^*, yielding:

▶ **Lemma 15.** *We have path data $q : p^*(F/A) = F$.*◇

Thus we have given a preimage F/A of F under p^*. To show that the fibre of c is inhabited we will show that $c(F/A) = (F, p)$, which is a path in $\mathtt{fib}_{i^*}$. We express all of this in terms of path data, and such a path in $\mathtt{fib}_{i^*}$ then corresponds to path data $q : p^*(F/A) = F$ which

[2] The map c is called `cxfib` in the code.

makes the following triangle commute:$^\diamond$

$$\begin{array}{ccc} i^*p^*(F/A) & \xrightarrow{\quad i^*(q) \quad} & i^*(F) \\ & \searrow \quad \swarrow_{p} & \\ & G \oplus A & \end{array} \tag{2}$$

where the rightmost map comes from i^*p^* being trivial. The key reason we have formulated things in terms of path data is so that the maps in the triangle above simply compute, because they have all been concretely constructed.

In the following, c refers to the map which lands in $\mathtt{fib}_{i^*}$ expressed in terms of path data.$^\diamond$

▶ **Lemma 16.** *We have $q : c(F/A) = (F, p)$ in $\mathtt{fib}_{i^*}$.*$^\diamond$

Proof. The previous lemma already yields path data $q : p^*(F/A) = F$, thus it remains to show that the triangle in Equation (2) commutes. The way the maps have been constructed, it's easiest to show this after flipping the triangle so that it starts at $G \oplus A$ and ends at $i^*p^*(F/A)$. (This is fine since all the maps are isomorphisms.) Thus we are comparing two maps out of a biproduct into a pullback. To check whether they are equal, we can check it on each inclusion of the biproduct and after projecting out of the pullback. In each of these cases one obtains diagrams which commute, but checking this is somewhat involved. Fortunately, by our having carefully crafted the path data involved, the maps simply compute and Coq is able to reduce the goal to a simple computation. ◀

Combining the three previous lemmas, we get a section of $c : \mathtt{AbSES}(B, G) \to \mathtt{fib}_{i^*}$. To conclude that c is an equivalence, we contract each fibre over some (F, p) to $(F/A, q)$.

▶ **Lemma 17.** *Suppose $G \to Y \to B$ is a short exact sequence, and let $q' : c(Y) = (F, p)$ in $\mathtt{fib}_{i^*}$. Then $(F/A, q) = (Y, q')$ in the fibre of c over (F, p).*$^\diamond$

Proof. Under our assumptions, we have the composite map $\phi : G \oplus A \to i^*p^*(Y) \to p^*(Y)$$^\diamond$ which by a diagram chase can be seen to be the inclusion $G \to p^*(Y)$ on one component, and $(0, p) : A \to p^*(Y)$ on the other.$^\diamond$. Consequently, the composite $\mathtt{pr}_1 \circ \phi \circ \mathtt{in}_A : A \to Y$ is trivial. By the universal property of the cokernel, we get an induced map $F/A \to Y$. Once again, by our careful construction of all the maps involved, it is straightforward to simply compute that this map defines path data $F/A = Y$ and moreover that this path lifts to a path in the fibre of c. There is a coherence between three paths in $\mathtt{AbSES}(A, G)$ which is trivially satisfied, since $\mathtt{AbSES}(A, G)$ is a 1-type. ◀

The final lemma implies that the fibres of c are contractible, which means that c is an equivalence and concludes the proof of Theorem 13. We now turn our attention to two applications of this theorem. The first application requires a lemma.

▶ **Lemma 18.** *Let $g : B' \to B$ be a homomorphism of abelian groups. For any A, the following diagram commutes, where the vertical isomorphisms are all given by Proposition 3:*$^\diamond$

$$\begin{array}{ccc} \Omega\,\mathtt{AbSES}(B, A) & \xrightarrow{\Omega(g^*)} & \Omega\,\mathtt{AbSES}(B', A) \\ \downarrow\sim & & \downarrow\sim \\ \mathtt{Hom}(B, A) & \xrightarrow{\phi \mapsto \phi \circ g} & \mathtt{Hom}(B', A)\,. \end{array} \tag{3}$$

Proof. Let $p : A \oplus B = A \oplus B$ be an element of the upper left corner, seen as path data. By path induction, one can easily show that the action of $\Omega(g^*)$ on paths is given by pulling back the path data. (Formally, one first proves this for paths with free endpoints, then you can specialise to loops.) This means that the following diagram commutes

$$\begin{array}{ccccccc} B' & \longrightarrow & A \oplus B' & \xrightarrow{\Omega(g^*)(p)} & A \oplus B' & \longrightarrow & A \\ \| & & \downarrow{\scriptstyle \mathrm{id} \oplus g} & & \downarrow{\scriptstyle \mathrm{id} \oplus g} & & \| \\ B' & \xrightarrow{(0,g)} & A \oplus B & \xrightarrow{p} & A \oplus B & \longrightarrow & A \end{array}$$

where we have used the functions underlying the path data p and $\Omega(g^*)(p)$, and the unlabeled arrows are the natural ones into or out of a biproduct. The composites of the top and bottom rows above are the results of sending p around the top-right and bottom-left corners of Diagram 3, respectively. Since this latter diagram commutes, so does Diagram 3. ◀

▶ **Proposition 19** ([6, Theorem III.3.4]). *We have an exact sequence of abelian groups:*◇

$$0 \longrightarrow \mathtt{Hom}(B,G) \xrightarrow{p^*} \mathtt{Hom}(E,G) \xrightarrow{i^*} \mathtt{Hom}(A,G) \rightarrowtail \mathtt{Ext}^1(B,G) \xrightarrow{p^*} \mathtt{Ext}^1(E,G) \xrightarrow{i^*} \mathtt{Ext}^1(A,G)\,.$$

Proof. This sequence comes from the long exact sequence of homotopy groups [10, Theorem 8.4.6] associated to the fibre sequence of Theorem 13, using Proposition 3 and the previous lemma to identify $\Omega\,\mathtt{AbSES}(-,G)$ with $\mathtt{Hom}(-,G)$. ◀

▶ Remark 20. The connecting map $\mathtt{Hom}(A,G) \to \mathtt{Ext}^1(B,G)$ in the sequence above is given by $\phi \mapsto \phi_* E$. Showing this from the fibre sequence is somewhat tedious; we have a proof on paper, but not yet a formalisation. Instead, we have formalised a direct proof that the map just stated yields exactness of the sequence.◇◇

We apply the six-term exact sequence to compute Ext groups of cyclic groups:

▶ **Corollary 21** ([6, Proposition III.1.1]). *For any $n > 0$ and abelian group A, we have*◇

$\mathtt{Ext}^1(\mathbb{Z}/n, A) \cong A/n$.

Proof. The short exact sequence $\mathbb{Z} \xrightarrow{n} \mathbb{Z} \to \mathbb{Z}/n$ yields a six-term exact sequence

$$\cdots \to \mathtt{Hom}(\mathbb{Z},A) \xrightarrow{n^*} \mathtt{Hom}(\mathbb{Z},A) \to \mathtt{Ext}^1(\mathbb{Z}/n,A) \to \mathtt{Ext}^1(\mathbb{Z},A) \to \cdots$$

in which the term $\mathtt{Ext}^1(\mathbb{Z},A)$ vanishes since $\mathbb{Z}$ is projective.◇◇ This means that the map $\mathtt{Hom}(\mathbb{Z},A) \to \mathtt{Ext}^1(\mathbb{Z}/n,A)$ is the cokernel of the preceding map. By identifying $\mathtt{Hom}(\mathbb{Z},A)$ with A, the claim follows. ◀

5 The Long Exact Sequence

We describe our formalisation of the higher Ext groups $\mathtt{Ext}^n(B,A)$ and their contravariant long exact sequence, which largely follows [6, Chapter III.5]. The covariant version can be constructed from the arguments in [7, Chapter VII.5], but we have not formalised this. The Baer sum is not yet formalised for $\mathtt{Ext}^n$ $(n > 1)$, so we only have a long exact sequence of *pointed sets*. Nevertheless, exactness for pointed sets and abelian groups coincide, so we automatically get a long exact sequence of the latter once we have the higher Baer sum.

The formalisation of this section is in the separate repository `Yoneda-Ext`, whose `README` file explains how to set up and build the code related to this chapter. There are also comments in the code which explain details beyond what we cover here.

5.1 The Type of Length-n Exact Sequences

We start by defining a type $\mathtt{ES}^n$ which we will equip with an equivalence relation by which $\mathtt{Ext}^n$ will be the quotient. These constructions will yield functors, which we explain.

The type $\mathtt{ES}^n(B, A)$ of **length-n exact sequences** is recursively defined as:$^\diamondsuit$

```
Fixpoint ES (n : nat) : AbGroup^op -> AbGroup -> Type
  := match n with
    | 0%nat => fun B A => Hom B A
    | 1%nat => fun B A => AbSES B A
    | S n => fun B A => exists M, (ES n M A) * (AbSES B M)
    end.
```

Thus $\mathtt{ES}^0(B, A)$ is definitionally $\mathtt{Hom}(B, A)$, and $\mathtt{ES}^1(B, A)$ is definitionally $\mathtt{AbSES}(B, A)$. One could also have started the induction at $n \equiv 1$ instead of $n \equiv 2$, but it is convenient to have this definitional equality at level $n \equiv 1$. The functoriality of $\mathtt{ES}^n$ is inherited from $\mathtt{AbSES}$ and defined in the obvious way by pulling back and pushing out. For $n > 0$, an element of $\mathtt{ES}^{n+1}(B, A)$ is denoted by $(F, E)_M$, with the obvious meaning. The type $\mathtt{ES}^n(B, A)^\diamondsuit$ is pointed by recursion, using the trivial abelian group in the place of M in the inductive step.

▶ **Definition 22.** *The **splice** operation is defined as*$^\diamondsuit$

$$F \circledcirc E :\equiv (F, E)_B : \mathtt{ES}^n(B, A) \to \mathtt{AbSES}(C, B) \to \mathtt{ES}^{n+1}(C, A).$$

By induction one can define a general splicing operation in which the second parameter can have arbitrary length$^\diamondsuit$, but we only need the restricted version above.

Now we equip $\mathtt{ES}^n(B, A)$ with a relation.

▶ **Definition 23.** *We define a relation* $\mathtt{es\_zig} : \mathtt{ES}^n(B, A) \to \mathtt{ES}^n(B, A) \to \mathtt{Type}$ *recursively as follows. For $n = 0, 1$,* $\mathtt{es\_zig}$ *is the identity type. For $n \geq 2$, a relation between two elements $(F, E)_M$ and $(Y, X)_N$ consists of a homomorphism $f : \mathtt{Hom}(M, N)$ along with a path $f_*(E) = X$ and a relation* $\mathtt{es\_zig}(F, f^*(Y))$ *(using functoriality of* $\mathtt{ES}^n$*).*$^\diamondsuit$

The relation $\mathtt{es\_zig}$ generates an equivalence relation $\mathtt{es\_eqrel}^\diamondsuit$ (denoted $\sim$ in the code) whose propositional truncation is $\mathtt{es\_meqrel}^\diamondsuit$. The functoriality of $\mathtt{ES}^n$ respects these relations.$^{\diamondsuit\diamondsuit}$ Basic results on equivalence relations are contained in `EquivalenceRelation.v`.

We emphasise that equivalence relation $\mathtt{es\_eqrel}$ is *not* equivalent to the identity type of $\mathtt{ES}^n$. Rather, it is an approximation of the identity type of the classifying space of the category $\mathtt{ES}^n$ (which we do not know if one can construct in HoTT). See, e.g., [6, Chapter III.5] for related discussion.

▶ **Definition 24.** *The pointed set* $\mathtt{Ext}^n(B, A)$ *is the quotient of* $\mathtt{ES}^n(B, A)$ *by the equivalence relation* $\mathtt{es\_meqrel}$*.*$^\diamondsuit$

The splice operation descends to this quotient.$^\diamondsuit$ By pushing out$^\diamondsuit$ and pulling back$^\diamondsuit$ extensions, Ext^n becomes a functor in each variable as well. Moreover, we have equalities $f^*(F) \circledcirc E = F \circledcirc f_*(E)$ whenever this expression makes sense, by the definition of $\mathtt{es\_zig}$.$^\diamondsuit$

▶ Remark 25. The definition of $\mathtt{Ext}^{n+1}(B, A)$ is, more conceptually, the $(n+1)$-fold tensor product of functors $\mathtt{Ext}^{n+1}(B, A) = \mathtt{Ext}^n(-, A) \otimes \mathtt{Ext}^1(B, -)$ (see, e.g., [5, Theorem 9.20] or [13, Eq. 4.3.4]). In our setup, this is a tensor product of Set-valued functors, which can be made into an abelian group by a construction similar to the Baer sum of Section 3.3 (though we have not yet formalised this). Alternatively, one could define $\mathtt{Ext}^{n+1}(B, A)$ as the $(n+1)$-fold tensor product of functors *into abelian groups*. [4, Lemma 2.1] implies that these two definitions coincide. We have chosen the present approach because we do not know of a direct construction of the long exact sequence for the latter approach.

5.2 The Long Exact Sequence

We now begin working towards the long exact sequence, following the proof of [6, Theorem XII.5.1]. As explained at the beginning of this section, we have only formalised the long exact sequence of *pointed sets* – however, exactness for pointed sets is the same as for abelian groups. Let us first recall the statement:

▶ **Theorem 26.** *Let $A \xrightarrow{i} E \xrightarrow{p} B$ be a short exact sequence of abelian groups. For any abelian group G, there is a long exact sequence by pulling back:*$^{\diamond\diamond\diamond}$

$$\cdots \xrightarrow{i^*} \mathtt{Ext}^n(A,G) \xrightarrow{-\circledcirc E} \mathtt{Ext}^{n+1}(B,G) \xrightarrow{p^*} \mathtt{Ext}^{n+1}(E,G) \xrightarrow{i^*} \cdots .$$

The proof in [6] first discusses the six-term exact sequence, which we proved as Proposition 19. It then reduces the question to exactness at the domain of the connecting map (Lemma XII.5.2, loc. cit.), and proves exactness at that spot using Lemmas XII.5.3, XII.5.4, and XII.5.5. We will show the three latter lemmas, then directly prove exactness at the other spots, essentially "in-lining" Lemma XII.5.2.

The various constructions we need to do are simpler to carry out on the level of $\mathtt{ES}^n$ as opposed to $\mathtt{Ext}^n$. For this reason we work and formulate things in terms of the former, and then deduce the desired statement for the latter.

Before attacking Lemma XII.5.3, we show the following:

▶ **Lemma 27.** *Consider two pairs of short exact sequences which can be spliced:*

$$(A \xrightarrow{l} Y \xrightarrow{s} B',\ B' \xrightarrow{k} X \xrightarrow{r} C), \qquad (A \xrightarrow{j} F \xrightarrow{q} B,\ B \xrightarrow{i} E \xrightarrow{p} C).$$

For any element of $\mathtt{es\_zig}(Y \circledcirc X, F \circledcirc E)$*, we have induced maps* $\mathtt{fib}_{s_*}(X) \to \mathtt{fib}_{q_*}(E)$$^{\diamond}$ *and* $\mathtt{fib}_{i^*}(F) \to \mathtt{fib}_{k^*}(Y)$$^{\diamond}$*.*

Proof. We only describe the first map since the second is analogous. The zig from $Y \circledcirc X$ to $F \circledcirc E$ gives a homomorphism $f : B' \to B$ along with two paths $f^*(F) = Y$ and $f_*(X) = E$. Let $G : \mathtt{fib}_{s_*}(X)$; by path induction we may assume $q_*(G) \equiv X$. The path $f^*(F) = Y$ means we have a commuting diagram:

$$\begin{array}{ccccc} A & \xrightarrow{l} & Y & \xrightarrow{s} & B' \\ \| & & \downarrow{\scriptstyle \phi} & & \downarrow{\scriptstyle f} \\ A & \xrightarrow{j} & F & \xrightarrow{q} & B\,. \end{array}$$

Thus $\phi_*(G)$ defines an element of $\mathtt{fib}_{q_*}(E)$ by $q_*(\phi_*(G)) = f_*(s_*(G)) \equiv f_*(X) = E$. ◀

▶ **Lemma 28** ([6, Lemma XII.5.3])**.** *Given two short exact sequences $A \xrightarrow{j} F \xrightarrow{q} B$ and $B \xrightarrow{i} E \xrightarrow{p} C$, the following types are logically equivalent:*$^{\diamond}$

1. $\mathtt{fib}_{i^*}(F)$*;*
2. $\mathtt{fib}_{q_*}(E)$*;*
3. $\mathtt{es\_eqrel}(\mathtt{pt}, F \circledcirc E)$*.*

Proof. The logical equivalence of between (1) and (2) is as described in [6].$^{\diamond}$ Moreover, the implication (2) to (3) is clear by the definition of `es_zig`. We need to show that (3) implies (1), and we proceed by induction on the length of the zig-zag.

In the base case we have an actual equality $\mathtt{pt} = F \circledcirc E$, in which case (1) clearly holds. For the inductive step, suppose we have two short exact sequences $A \xrightarrow{l} Y \xrightarrow{s} B'$ and $B' \xrightarrow{k} X \xrightarrow{r} C$ such that $Y \circledcirc X$ is related to $\mathtt{pt}$ by a length n zig-zag, and we have either zig

or a zag relating $Y \circledcirc X$ to $F \circledcirc E$. If we have a zig, then we use the induction hypothesis to get an element of $\mathtt{fib}_{s_*}(X)$ to which we apply the map $\mathtt{fib}_{s_*}(X) \to \mathtt{fib}_{q_*}(E)$ from the previous lemma. This suffices since (1) and (2) are logically equivalent.

If we have a zag, then the previous lemma gives a map $\mathtt{fib}_{k^*}(Y) \to \mathtt{fib}_{i^*}(F)$, so we are done by the induction hypothesis. ◀

We reformulate condition (2) in a manner that generalises to $\mathtt{ES}^n$.$^\diamondsuit$

```
Definition es_ii_family `{Univalence} {n : nat} {C B A : AbGroup}
  : ES n.+1 B A -> ES 1 C B -> Type
  := fun E F => { alpha : { B' : AbGroup & B' $-> B }
                        & (es_eqrel pt (es_pullback alpha.2 E))
                          * (hfiber (abses_pushout alpha.2) F) }.
```

▶ **Lemma 29** ([6, Lemma XII.5.4])**.** *In the situation of the previous lemma, the types* $\mathtt{fib}_{q_*}(E)$ *and* $\mathtt{es\_ii\_family}(F, E)$ *are logically equivalent.*$^\diamondsuit$

Mac Lane appeals to the six-term exact sequence to prove this lemma, but we give a direct construction. In order to show Lemma XII.5.3, we prove a higher analogue of Lemma 27. This analogue is phrased in terms of the "relation fibre" `rfiber`, which takes the fibre of a point with respect to a relation.

▶ **Lemma 30.** *Let* $n > 0$ *and consider* $Y : \mathtt{ES}^n(B', A)$, $F : \mathtt{ES}^n(B, A)$, *and two short exact sequences* $B' \xrightarrow{k} X \to C$ *and* $B \xrightarrow{i} E \to C$. *Given* $\mathtt{es\_zig}(Y \circledcirc X, F \circledcirc E)$, *we have maps* $\mathtt{rfiber}_{i^*}(F) \to \mathtt{rfiber}_{k^*}(Y)^\diamondsuit$ *and* $\mathtt{es\_ii\_family}(Y, X) \to \mathtt{es\_ii\_family}(F, E)^\diamondsuit$.

▶ **Lemma 31** ([6, Lemma XII.5.5])**.** *Let* $n > 0$, $F : \mathtt{ES}^n(B, A)$, *and* $E : \mathtt{ES}^1(C, B)$. *The following types are equivalent:*$^\diamondsuit$

1. $\mathtt{fib}_{i^*}(E)$;
2. $\mathtt{es\_ii\_family}(F, E)$;
3. $\mathtt{es\_eqrel}(\mathtt{pt}, F \circledcirc E)$.

Proof. We first prove an auxiliary lemma which shows that if the three statements are equivalent for a given n, then (1) and (2) are equivalent for $n + 1$. The base case for this lemma is simply Lemma 28. For the inductive step, our auxiliary lemma gives us that (1) and (2) are equivalent. It is easy to show that (2) always implies (3), so it remains to show that (3) implies either (1) or (2). For this we induct on the length of a zig-zag, and use the equivalence of (1) and (2) along with the previous lemma, similarly (at least in structure) to the proof of Lemma 28. ◀

Afterwards, we reformulate this lemma in terms of $\mathtt{Ext}^n$.$^\diamondsuit$ With this lemma at hand, and using similar methods to the ones presented here, we follow the proof of [6, Lemma 5.2] to deduce exactness of the long sequence of Theorem 26.

6 Conclusion

We have presented a formalisation of the theory of Yoneda Ext in the novel setting of homotopy type theory, starting from the basic definition of a short exact sequence and arriving at the (contravariant) long exact sequence, with various related results along the way. At present, the long exact sequence is one of pointed sets, and we leave it to future work to formalise the Baer sum on $\mathtt{Ext}^n$ for $n > 1$, which would promote this into a long exact sequence of abelian groups. (The notion of exact sequence coincides for abelian groups and pointed sets.)

For pragmatic reasons we have worked with abelian groups, though it is clear that everything we have done could be applied to general modules. Even so, the higher Ext groups of abelian groups do not necessarily vanish in HoTT [2], so these are already interesting. There are various more general approaches that we would like to consider in the future, such as working with *pure* exact sequences (in which the classes of monomorphisms and epimorphisms are appropriately replaced) in an abelian category.

Many of our results have been contributed to the Coq-HoTT library [3] under the namespace `Algebra.AbSES`, which currently weighs in at about 2900 lines of code (whitespace and comments included). This excludes the various contributions made to other parts of the library; the precise contributions may be seen through the pull requests `#1534`, `#1646`, `#1663`, `#1712`, `#1718`, and `#1738`. In addition, the code for the long exact sequence currently weighs in at about 1350 lines in the separate `Yoneda-Ext` repository.

The formalisation covers a substantial part of chapters III.1-3, III.5, and XII.5 of [6], but also extends beyond the classical theory. In particular, our proof of Theorem 13 is new even for classical Yoneda Ext (though the theorem is known). This theorem presented the most challenging part of this formalisation, as it required managing considerable amounts of coherence. The other challenging part was the long exact sequence, whose proof involves an intricate induction and numerous constructions. By formalising these theorems we have not only established their correctness but also contributed evidence of the feasibility of dealing with sophisticated mathematical structures in a proof assistant like Coq.

References

1. Ulrik Buchholtz, J. Daniel Christensen, Jarl G. Taxerås Flaten, and Egbert Rijke. Central H-spaces and banded types, 2023. `doi:10.48550/ARXIV.2301.02636`.
2. J. Daniel Christensen and Jarl G. Taxerås Flaten. Ext groups in homotopy type theory, 2023. `arXiv:2305.09639`.
3. Coq-HoTT. The Coq-HoTT library. URL: `https://github.com/HoTT/Coq-HoTT`.
4. René Guitart and Luc Van den Bril. Calcul des satellites et présentations des bimodules à l'aide des carrés exacts. *Cahiers de Topologie et Géométrie Différentielle Catégoriques*, 24(3):299–330, 1983. URL: `http://archive.numdam.org/item/CTGDC_1983__24_3_299_0/`.
5. René Guitart and Luc Van den Bril. Calcul des satellites et présentations des bimodules à l'aide des carrés exacts (2e partie). *Cahiers de Topologie et Géométrie Différentielle Catégoriques*, 24(4):333–369, 1983. URL: `http://archive.numdam.org/item/CTGDC_1983__24_4_333_0/`.
6. Saunders Mac Lane. *Homology*. Springer, 1963.
7. Barry Mitchell. *Theory of categories*. Academic Press, 1965.
8. Vladimir S. Retakh. Homotopic properties of categories of extensions. *Russian Mathematical Surveys*, 41(6):217–218, December 1986. `doi:10.1070/rm1986v041n06abeh004237`.
9. Egbert Rijke. *Introduction to Homotopy Type Theory*. Cambridge University Press, 2023. To appear.
10. The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.
11. Floris van Doorn. *On the Formalization of Higher Inductive Types and Synthetic Homotopy Theory*. PhD thesis, Carnegie Mellon University, 2018. `doi:10.48550/arXiv.1808.10690`.
12. Nobuo Yoneda. On the homology theory of modules. *Journal of the Faculty of Science, the University of Tokyo Section I*, 7:193–227, 1954.
13. Nobuo Yoneda. On Ext and exact sequences. *Journal of the Faculty of Science, the University of Tokyo Section I*, 8:507–576, 1960.

LISA – A Modern Proof System

Simon Guilloud
Laboratory for Automated Reasoning and Analysis, EPFL, Lausanne, Switzerland

Sankalp Gambhir
Laboratory for Automated Reasoning and Analysis, EPFL, Lausanne, Switzerland

Viktor Kunčak
Laboratory for Automated Reasoning and Analysis, EPFL, Lausanne, Switzerland

Abstract

We present LISA, a proof system and proof assistant for constructing proofs in schematic first-order logic and axiomatic set theory. The logical kernel of the system is a proof checker for first-order logic with equality and schematic predicate and function symbols. It implements polynomial-time proof checking and uses the axioms of ortholattices (which implies the irrelevance of the order of conjuncts and disjuncts and additional propositional laws). The kernel supports the notion of theorems (whose proofs are not expanded), as well as definitions of predicate symbols and objects whose unique existence is proven. A domain-specific language enables construction of proofs and development of proof tactics with user-friendly tools and presentation, while remaining within the general-purpose language, Scala. We describe the LISA proof system and illustrate the flavour and the level of abstraction of proofs written in LISA. This includes a proof-generating tactic for propositional tautologies, leveraging the ortholattice properties to reduce the size of proofs. We also present early formalization of set theory in LISA, including Cantor's theorem.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Proof assistant, First Order Logic, Set Theory

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.17

Supplementary Material *Software (Source Code)*: `https://github.com/epfl-lara/lisa`
archived at `swh:1:dir:614ac272bee1c2bf21308ad532c3ca3dd3ec3832`

1 Introduction

We present the design and initial implementation of a new proof assistant, named LISA. Much like Mizar [31], LISA aims to use classical mainstream foundations of mathematics with first order logic and set theory. LISA uses (single-sorted) first-order logic (with schematic variables) as the syntactic framework, sequent calculus as the deduction framework and set theory as the semantic framework. On top of this foundation, we can construct mathematical theories without introducing additional axioms. As the target use of LISA we envision a library of theorems, but also correctness proofs of computer systems.

LISA's source code and a reference manual, as well as all the examples in the present paper, are available from

`https://github.com/epfl-lara/lisa`

1.1 Design Goals

Our design is inspired by the LCF line of proof assistants, including HOL Light, HOL4, and Isabelle. The envisioned path for axiomatic foundations is closer to Mizar. LISA's logical kernel is a hybrid between LCF-style encoding of theorems as a sealed Theorem type (similar to HOL Light [22]) and explicit requirement of proofs. Namely, proofs are self-contained


© Simon Guilloud, Sankalp Gambhir, and Viktor Kunčak;
licensed under Creative Commons License CC-BY 4.0

sequences of proof steps that derive a conclusion from assumptions (they are not explicitly in the form of lambda terms). LISA's kernel checks the validity of steps and assumptions, and then creates an instance of a theorem.

As the unified *implementation, proof writing and tactic language*, we use Scala instead of the ML family of languages that are common to many proof assistants. Scala is a high-level functional and object-oriented language. We hope to avoid a sharp boundary between user proofs and tactic developments by using a single language with good support for domain-specific constructs. To provide a flavour of LISA, consider several ways of constructing proofs that are available to LISA users. Figure 2 shows a proof of Pierce's law as an explicit sequence of sequent calculus proof steps. Figure 5 and Figure 6 show proofs built using a higher-level domain-specific language (DSL). This DSL detects high-level errors in incorrect proofs, but always generates the underlying lower-level proof and forwards it to the kernel to obtain a kernel-certified theorem. Finally, Figure 9 shows a solver for propositional tautologies that uses the same mechanisms as the proofs to implement a proof tactic. It was not our immediate goal to create an interactive experience, so our interaction model is more HOL4-like than Isabelle/HOL-like. For us, this means using Scala IDEs, rerunning projects, relying on incremental compilation. As the sizes of theories grow, we plan to develop serializations for proofs and theories to reduce re-execution. We discuss both the kernel and the DSL in the rest of the paper.

The design philosophy of LISA focuses on what one might call the *Six Virtues of Modern Proof Systems*. *Efficiency* say a proof system components should have polynomial complexity (as close as possible to linear). *Trust* means high confidence in the system, through a combination of well understood mathematical foundations, explicit proofs and a concise logical kernel. *Usability* is making it simple, both for human users and automated methods, to formalize mathematics and to develop tools. *Predictability* is the property of systems whose behaviour and output have clear characterizations. *Interoperability*, whose importance has become clear over the years, consists in making it as easy as possible for the system to be used by other systems and to export and import proofs to and from other systems. Finally, *Programmability* implies that as a computer system, a proof assistant should provide all the expressiveness allowed by a programming language. When designing and developing the LISA proof system, we aim to respect the six virtues as much as possible, and, when they oppose each other, to strike for the best balance between them.

1.2 Contributions

The contribution of this paper is to present the design of LISA, a new proof construction system embedded in Scala, based on schematic first-order logic with set theory axioms. We focus on the following aspects.

- We describe how the logical kernel is constructed and how it can be used or interacted with by other tools.
- As the most unusual design aspect, we describe ortholattice-based algorithms implemented in the kernel to make proofs shorter.
- We present a domain-specific language embedded in Scala that makes the writing of proofs easier and generates and checks kernel proofs to obtain kernel-certified theorems.
- We show that the same domain-specific language can scale from writing proofs of specific theorems to writing general tactics. As an example of a tactic, we present a (proof generating) solver for propositional formulas leveraging the ortholattice algorithm.
- We report on the initial steps of developing elementary axiomatic set theory in the system.

2 Logical Kernel

LISA's deductive system is a variant of Gentzen's Sequent Calculus for first-order logic (FOL) [15]. Formally, a sequent in LISA is a pair of sets of formulas Γ and Δ, represented $\Gamma \vdash \Delta$ and its interpretation is $\bigwedge \Gamma \to \bigvee \Delta$. LISA extends the prototypical Sequent Calculus with schematic symbols, substitution rules, and a normalization of formulas.

2.1 Schematic Symbols

Formulas in LISA's kernel are built with the usual variables, constant function and predicate symbols, logical connectors and binders, but also admit the use of schematic function, predicate and connector symbols. These symbols behave like uninterpreted constant symbols which can be substituted by any well-typed term or formula across a whole sequent, or like variables which cannot be bound. We refer to them as second-order schematic symbols, as opposed to regular variables, which are first-order schematic symbols. This gives the system a flavour of second order logic and allows writing axiom and theorem schemas, as the following example illustrates:

▶ **Example 1.** The following sequent (whose proof we show in Figure 5) is provable without additional assumptions on the function symbol f and the predicate symbol P:

$$\forall x.P(x) \to P(f(x)) \vdash \forall x.P(x) \to P(f(f(x)))$$

This means that this sequent with f replaced by a specific term (with a distinguished free variable) remains provable by an analogous proof, and similarly for P replaced by a formula.

In traditional first-order logic, this concept is formalized as a meta-theorem stating that for every f and P, a corresponding proof can be built. However, in a formal setting, this requires duplicating the whole proof for every specific f and P. Instantiation of schematic symbols avoids this issue of proof duplication. This is similar to the schematic variables found in Isabelle [33], and in particular Isabelle/FOL [35], where variables from the meta-logic can be used to represent arbitrary functions, predicates, and connectors in FOL. Crucially, it does not increase the expressive power of the system, because it can, in principle, be simulated.

2.2 Ortholattice Algorithm Applied to First-Order Logic

We find that using a proof system that is sensitive to the order of conjuncts and similar semantically irrelevant syntactic differences can be frustrating and increases proof size unnecessarily. To address this issue, LISA's kernel strengthens sequent calculus with a built-in algorithm to compute normal form and equivalence of formulas with respect to a subset of equational rules of propositional logic. These rules, shown in Table 1, characterize the algebraic theory of ortholattices (abbreviated OL) [3, Chapter II.1], [7].

Ortholattices are a generalization of Boolean algebra where instead of the law of distributivity, the weaker absorption law (L9, Table 1) holds. In particular, every identity in the theory of ortholattices is also a theorem of propositional logic.

This algebraic structure has been shown to possess a quadratic-time normalization algorithm [18] and has been suggested as the basis for normalization of formulas in the context of verification and mechanized proofs. Notably, it subsumes negation normal form.

As a special kind of lattices, ortholattices can be viewed as partially ordered sets, with the ordering relation on two elements a and b of an ortholattice defined as $a \leq b \iff a \wedge b = a$, which, by absorption (L9), is also equivalent to $a \vee b = b$. If s and t are terms over the

Table 1 Laws of ortholattices, an algebraic theory with signature $(S, \wedge, \vee, 0, 1, \neg)$. [18]

L1:	$x \vee y = y \vee x$	L1':	$x \wedge y = y \wedge x$
L2:	$x \vee (y \vee z) = (x \vee y) \vee z$	L2':	$x \wedge (y \wedge z) = (x \wedge y) \wedge z$
L3:	$x \vee x = x$	L3':	$x \wedge x = x$
L4:	$x \vee 1 = 1$	L4':	$x \wedge 0 = 0$
L5:	$x \vee 0 = x$	L5':	$x \wedge 1 = x$
L6:	$\neg\neg x = x$	L6':	same as L6
L7:	$x \vee \neg x = 1$	L7':	$x \wedge \neg x = 0$
L8:	$\neg(x \vee y) = \neg x \wedge \neg y$	L8':	$\neg(x \wedge y) = \neg x \vee \neg y$
L9:	$x \vee (x \wedge y) = x$	L9':	$x \wedge (x \vee y) = x$

signature $(S, \wedge, \vee, 0, 1, \neg)$, we denote $s \leq_{OL} t$ if and only if $OL \vDash s \leq t$, i.e., it holds in all ortholattices. We write $s \sim_{OL} t$ if both $s \leq_{OL} t$ and $s \geq_{OL} t$ hold (or equivalently, if $OL \vDash s = t$). Theorem 1 is the main result we rely on.

▶ **Theorem 1** ([18]). *There exists an algorithm running in worst case quadratic time producing, for any terms s over the signature $(\wedge, \vee, \neg)$, a normal form $NF_{OL}(s)$ such that for any t, $s \sim_{OL} t$ if and only if $NF_{OL}(s) = NF_{OL}(t)$. The algorithm is also capable of deciding if $s \leq_{OL} t$ holds in quadratic time.*

Moreover, the algorithm works with structure sharing with the same complexity, which is very relevant for example when $x \leftrightarrow y$ is expanded to $(x \wedge y) \vee (\neg x \wedge \neg y)$. It can produce a normal form in this case as well.

These properties, along with completeness characterization, make the OL algorithm a good candidate to include in a proof system. LISA's kernel further extends OL inequality algorithm to first order logic formulas as follows. It first expresses the formula using de Bruijn indices [11], then desugars $\exists.\phi$ into $\neg\forall.\neg\phi$. It then extends the OL algorithm with the rules in Table 2.

Table 2 Extension of OL algorithm to first-order logic. We call it the F(OL)^2 algorithm. $=$ denotes the equality predicate in FOL, while $==$ denotes syntactic equality of terms.

	To decide...	Reduce to...
1	$\{\wedge, \vee, \rightarrow, \leftrightarrow, \neg\}(\vec{\phi}) \leq \psi$	Base algorithm
2	$\phi \leq \{\wedge, \vee, \rightarrow, \leftrightarrow, \neg\}(\vec{\psi})$	Base algorithm
3	$s_1 = s_2 \leq t_1 = t_2$	$\{s_1, s_2\} == \{t_1, t_2\}$
4	$\phi \leq t_1 = t_2$	$t_1 == t_2$
5	$\forall.\phi \leq \forall.\psi$	$\phi \leq \psi$
6	$'C(\phi_1, ..., \phi_n) \leq 'C(\psi_1, ..., \psi_n)$	$\phi_i \sim_{OL} \psi_i$, for every $1 \leq i \leq n$
7	Anything else	`false`

When either of the two formulas being compared have a top-level propositional operator (cases 1 and 2), the recursion is done according to the algorithm described in [18], considering any non-propositional expressions (predicates, quantified formulas, and schematic connectors) as propositional variables. The third and fourth rules take into account reflexivity and symmetry of equality. The fifth relies on monotonicity of $\forall$, and the sixth rule applies when $'C$ is a schematic connector, i.e., a logical connector about which we know nothing. These rules extend to the normal-form-producing algorithm, and it is easy to see that if $\leq$ is interpreted as logical implication, they are sound. We decided not to include a rule

such as $\forall.\phi \leq \phi(t)$. The reason is that incorporating such a rule systematically runs risk of introducing higher complexity [27] in the kernel. We instead decided that such steps should be implemented using tactics, outside the kernel in the future (possibly making use of type-like hints encoded in first-order logic [14]).

Using the First Order Logic OrthoLattices algorithm, noted $\mathrm{F(OL)}^2$, the proof checker in LISA's kernel performs every correctness check up to $\mathrm{F(OL)}^2$ equivalence. This does not prevent sequents and formulas from having arbitrary constructions and being inspected in a stable, predictable way by tactics, as formulas are not normalized in-place. The set of LISA deduction rules is shown in Figure 1.

Moreover, the proof checker contains a special `Restate` proof step, which permits $\mathrm{F(OL)}^2$-transformations on the entire sequent, leveraging the interpretation of a sequent as a formula (an implication). We also leverage specifically the partial order computed by $\mathrm{F(OL)}^2$ to expand the usual `Weakening` rule so that the premise sequent only has to be $\leq_{F(OL)^2}$ stronger than the conclusion, with both interpreted as formulas. `Weakening` clearly subsumes `Restate`, but the latter ensures that the transformation is actually an equivalence and hence could be reversed, which can be a useful safeguard in practice. These rules subsume most propositional rules in Figure 1.

2.3 Substitution Rules

The substitution rules substitute equal terms or equivalent formulas inside a formula. They are deduced steps whose simulation from simpler steps can take a number of steps linear in the size of the sequent, yet are very frequent both in human-written proofs and automated reasoning (as done by SAT solvers or in systems with rewrite rules, for example), justifying their inclusion as base steps. A special case of substitution that is particularly important is the following:

$$\frac{\phi \vdash \psi}{\phi \vdash \psi[\phi := \top]}\ \text{SubstIff}$$

This holds in a single step because $\phi \leftrightarrow \top \sim_{F(OL)^2} \phi$. In fact, `Restate` and `SubstIff` form a complete basis for propositional logic that we will leverage in Subsection 4.1 to write a complete proof-producing tactic for propositional logic.

The inclusion of $\mathrm{F(OL)}^2$ and the substitution and instantiation deduced rules in the logical kernel is a slight bend to the trust principle, but as the algorithm is only 300 lines of code, this is largely overshadowed by the increased usability and shorter proofs. In fact, the whole kernel adds up to a grand total of only 1607 lines of code. This comprises the implementation of first-order logic, the $\mathrm{F(OL)}^2$ algorithm, first and second-order substitution, the sequent calculus steps, the proof checker, and a manager for definitions and theorems (detailed in Subsection 2.5). Moreover, LISA's kernel is efficient: except for the quadratic $\mathrm{F(OL)}^2$ algorithm, every procedure in the kernel is linear (up to logarithmic coefficients) in the size of the formulas or proofs being considered.

2.4 Proof Objects

In LISA, a proof is an explicit list of proof steps, where each step can refer to previous steps via their respective position in the list and be referred by multiple subsequent steps. In other words, a proof is represented as a topological linearization of the proof tree, or, more generally, a directed acyclic graph (permitting reuse of intermediate steps). A proof step also contains the arguments that allow the proof checker to efficiently verify it. In particular, LISA's kernel does not rely on a unification algorithm to check correctness of proof steps related to quantifiers.

$$\frac{}{\Gamma, \phi \vdash \phi, \Delta}\ \text{Hypothesis}$$

$$\frac{\Gamma \vdash \phi, \Delta \qquad \Sigma, \phi \vdash \Pi}{\Gamma, \Sigma \vdash \Delta, \Pi}\ \text{Cut}$$

$$\frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \Delta}\ \text{LeftAnd} \qquad \frac{\Gamma \vdash \phi, \Delta \qquad \Sigma \vdash \psi, \Pi}{\Gamma, \Sigma \vdash \phi \wedge \psi, \Delta, \Pi}\ \text{RightAnd}$$

$$\frac{\Gamma, \phi \vdash \Delta \qquad \Sigma, \psi \vdash \Pi}{\Gamma, \Sigma, \phi \vee \psi \vdash \Delta, \Pi}\ \text{LeftOr} \qquad \frac{\Gamma \vdash \phi, \psi\Delta}{\Gamma \vdash \phi \vee \psi, \Delta}\ \text{RightOr}$$

$$\frac{\Gamma \vdash \phi, \Delta \qquad \Sigma, \psi \vdash \Pi}{\Gamma, \Sigma, \phi \to \psi \vdash \Delta, \Pi}\ \text{LeftImplies} \qquad \frac{\Gamma, \phi \vdash \psi, \Delta}{\Gamma \vdash \phi \to \psi, \Delta}\ \text{RightImplies}$$

$$\frac{\Gamma, \phi \to \psi \vdash \Delta}{\Gamma, \phi \leftrightarrow \psi \vdash \Delta}\ \text{LeftIff} \qquad \frac{\Gamma \vdash \phi \to \psi, \Delta \qquad \Sigma \vdash \psi \to \phi, \Pi}{\Gamma, \Sigma \vdash \phi \leftrightarrow \psi, \Delta, \Pi}\ \text{RightIff}$$

$$\frac{\Gamma \vdash \phi, \Delta}{\Gamma, \neg\phi \vdash \Delta}\ \text{LeftNot} \qquad \frac{\Gamma, \phi \vdash \Delta}{\Gamma \vdash \neg\phi, \Delta}\ \text{RightNot}$$

$$\frac{\Gamma, \phi[t := 'x] \vdash \Delta}{\Gamma, \forall 'x.\phi \vdash \Delta}\ \text{LeftForall} \qquad \frac{\Gamma \vdash \phi, \Delta}{\Gamma \vdash \forall 'x.\phi, \Delta}\ \text{RightForall}$$

$$\frac{\Gamma, \phi \vdash \Delta}{\Gamma, \exists 'x.\phi \vdash \Delta}\ \text{LeftExists} \qquad \frac{\Gamma \vdash \phi[t := 'x], \Delta}{\Gamma \vdash \exists 'x.\phi, \Delta}\ \text{RightExists}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma[\psi(\vec{v}) := 'p(\vec{v})] \vdash \Delta[\psi(\vec{v}) := 'p(\vec{v})]}\ \text{InstSchema}$$

$$\frac{\Gamma, \phi[s := 'f] \vdash \Delta}{\Gamma, s = t, \phi[t := 'f] \vdash \Delta}\ \text{LeftSubstEq} \qquad \frac{\Gamma \vdash \phi[s := 'f], \Delta}{\Gamma, s = t \vdash \phi[t := 'f], \Delta}\ \text{RightSubstEq}$$

$$\frac{\Gamma, \phi[a := 'p] \vdash \Delta}{\Gamma, a \leftrightarrow b, \phi[b := 'p] \vdash \Delta}\ \text{LeftSubstIff} \qquad \frac{\Gamma \vdash \phi[a := 'p], \Delta}{\Gamma, a \leftrightarrow b \vdash \phi[b := 'p], \Delta}\ \text{RightSubstIff}$$

$$\frac{\Gamma, t = t \vdash \Delta}{\Gamma \vdash \Delta}\ \text{LeftRefl} \qquad \frac{}{\vdash t = t}\ \text{RightRefl}$$

$$\frac{\Gamma_1 \vdash \Delta_1}{\Gamma_2 \vdash \Delta_2}\ \text{Restate}\ \ \text{if}\ \left(\bigwedge \Gamma_1 \to \bigvee \Delta_1\right) \sim_{F(OL)^2} \left(\bigwedge \Gamma_2 \to \bigvee \Delta_2\right)$$

$$\frac{\Gamma_1 \vdash \Delta_1}{\Gamma_2 \vdash \Delta_2}\ \text{Weakening}\ \ \text{if}\ \left(\bigwedge \Gamma_1 \to \bigvee \Delta_1\right) \leq_{F(OL)^2} \left(\bigwedge \Gamma_2 \to \bigvee \Delta_2\right)$$

Figure 1 Deduction rules allowed by LISA's kernel. Different occurrences of the same symbols need not represent equal elements, but only elements with the same $F(OL)^2$ normal form.

Moreover, proofs are standalone objects checkable and exportable without the need for any kind of context. Figure 2 shows an example of sequent calculus proof as a sequence of steps. Each step lists a sequent with a rule from Figure 1 and a list of (the position of) previous steps from which the sequent follows. Figure 3 shows executable Scala code that denotes the same proof, which can be given directly to the LISA kernel. The kernel can efficiently check its correctness and create a theorem whose statement corresponds to the last sequent, corresponding to the root of the proof tree. Note that, in this particular case, the same conclusion could be reached in a single step using the `Restate` rule.

If the proof relies on external theorems, axioms or definitions, those are stated after the list of proof steps and referred to with negative positions. We call those *imported* sequents (*imports*, for short). We adopt an analogous mechanism to support *subproofs*. A subproof simulates deduced steps by encapsulating an inner proof and appears as a single step in the outer proof. In that case, the premises of the subproof become imports of the inner proof.

0	`Hypothesis`	$\phi \vdash \phi$
1	`Weakening`(0)	$\phi \vdash \phi, \psi$
2	`RightImplies`(1)	$\vdash \phi, (\phi \to \psi)$
3	`LeftImplies`$(2, 0)$	$(\phi \to \psi) \to \phi \vdash \phi$
4	`RightImplies`(3)	$\vdash ((\phi \to \psi) \to \phi) \to \phi$

Figure 2 The proof of Pierce's Law as a sequence of steps using classical Sequent Calculus rules.

```
1 val PierceLawProof = SCProof(IndexedSeq(
2     Hypothesis(               ϕ ⊢ ϕ,                                  ϕ),
3     Weakening(                ϕ ⊢ (ϕ, ψ),                             0),
4     RightImplies(            () ⊢ (ϕ, ϕ ⟹ ψ),                  1, ϕ, ψ),
5     LeftImplies( (ϕ ⟹ ψ) ⟹ ϕ ⊢ ϕ,                  2, 0, (ϕ ⟹ ψ), ϕ),
6     RightImplies(            () ⊢ ((ϕ ⟹ ψ) ⟹ ϕ) ⟹ ϕ,
7                                                  3, (ϕ ⟹ ψ) ⟹ ϕ, ϕ)
8 ), Seq.empty /* no imports */ )
```

Figure 3 The proof from Figure 2 written for LISA's kernel. $\vdash$ and $\Longrightarrow$ are alternative, nicer constructors for sequents and formulas and are not part of the kernel. The second argument (here empty) is the sequence of proof imports.

2.5 Theories

LISA's proof checker can be used as a tool to produce and check proofs, independent of any context, but is not a sufficient tool to develop mathematical theories, as it lacks in particular the ability to make definitions. For this task, the kernel also offers a minimal utility to allow development of mathematical theories with the ability to introduce axioms, theorems, and definitions with guaranteed soundness.

Theorems

This part of the kernel, called the `Theory`, is inspired from the LCF style [16]. It allows checking a proof once, producing a value of a sealed type `Theorem`, which can then be reused many times. The proof can then be forgotten. The `Theory` will also verify that the given

proof's imports are properly justified by existing axioms, theorems or definitions, so that the proven `Theorem` can be considered unconditionally true, unlike its standalone proof. Figure 4 shows how to use the `Theory` to obtain a theorem.

```
1 val theory = new Theory
2 val pierceThm: theory.Theorem = theory.makeTheorem(
3    "Pierce's Law",
4    () ⊢ ((ϕ ⟹ ψ) ⟹ ϕ) ⟹ ϕ,
5    PierceLawProof,
6    Seq.empty
7 )
```

Figure 4 The proof from Figure 3 can be transformed into a `Theorem` by a `Theory`. The arguments are, in order, the name of the theorem, its statement, a proof of the statement and the list of previous theorems, axioms or definitions used to justify the proof's imports, if any.

The `Theory` naturally corresponds to the concept of a "mathematical theory" in first-order logic, containing the language and axioms of said theory. To allow coexistence of multiple different theories with different valid theorems, LISA makes the `Theory` a class that can be instantiated multiple times. The `Theorem` type is dependent on a specific instance of `Theory`, so that two different theories will reject the theorem of the other. In a language without dependent types, this could be replaced by a simple runtime check. Note that in proof development, it is expected that the user will never need to use more than one theory at once, so this aspect is abstracted by the DSL.

Definitions

The theory also allows introducing new definitions for predicate and function symbols.

A *predicate symbol* P definition is of the form $P(x_1, \ldots, x_n) := \phi_{x_1,\ldots,x_n}$, where the $x_1, \ldots, x_n$ are the free variables of a given formula ϕ. To define a *function symbol* f, the definition requires a proof of unique existence of the form:

$$\exists! y.\ \phi_{y,x_1,\ldots,x_n} \tag{1}$$

and introduces a definitional axiom $\phi_{f(x_1,\ldots,x_n),x_1,\ldots,x_n}$, where again $x_1, \ldots, x_n$ are the free variables of the formula ϕ. To make such a definition, the `Theory` checks that the symbol has not already been defined and requires a proof of (1), i.e., of existence and uniqueness.

Remark on unique existence. One may hope that only the existence (but not uniqueness) was needed to obtain conservative extensions in first-order logic. Unfortunately, this is not true in the presence of axiom schemas. In particular, with such a definition principle, it becomes possible to prove the Axiom of Choice in ZF set theory, while they are well known to be independent [8]. Indeed, in ZF it is possible to prove

$$\forall x. \exists y.\ (x \neq \emptyset \implies y \in x)$$

from which we would obtain a function pick with the property

$$\forall x.\ (x \neq \emptyset \implies \text{pick}(x) \in x)$$

If then the symbol pick is allowed in axiom schemas, as would be the case in LISA, it is then easy to use pick and the replacement schema to construct a choice function on any set (see also LISA's Reference Manual [17]).

Abstraction via underspecified definitions. We have seen that we need uniqueness to ensure conservative extensions. On the other hand, such requirement often forces the defining formula to be overly specific and representation-dependant. For example, one may want to define the set of real numbers, $\mathbb{R}$, as a structure that satisfies the axioms of real closed fields. Since there are many isomorphic structures satisfying these, a uniqueness proof cannot be obtained. It is then necessary to use a specific construction, such as Cauchy sequences, as the definition of the set of real numbers. This, however, means that it becomes possible to prove properties of real numbers which are specific to the chosen representation, which is undesirable and especially so when transferring proofs to other proof systems, which may have different representations of reals. Our solution is to allow *underspecified* definitions. An underspecified definition still requires existence and uniqueness (ensuring a conservative extension), but the theorem that the kernel provides is only the desired, weaker, one. This mechanism makes use of the $\leq$ relation of $\mathrm{F}(\mathrm{OL})^2$. Section 3 shows an example of the use of underspecified definitions in set theory.

This issue is addressed in Metamath [29] by assuming a specific construction of the structure to conditionally prove a desired defining property (for example, the axioms of the real field) and then introducing said property independently as an axiom. This mechanism however is not enforced by Metamath itself but only an informal practice. LISA's kernel support for underspecified definitions ensures that the same goal is achieved with guaranteed soundness. Underspecification was also discussed in the context of HOL by Rob Arthan [1]. Our approach is similar but the challenges are different. The use of Hilbert's description operator leads to undesired properties being provable, similarly as definition via unique existence, but for the reason explained above, Lisa can't relax the condition to existence only due to the presence of axiom schemas. Forgetting a part of the definition after it was made was tried in HOL Light, but this made reasoning about the system harder, as it has to take into account not only the state of the system but also the sequence of operation leading to this state. LISA avoid this issue by making underspecified definitions an integral part of the foundation.

3 DSL for LISA in Scala

While the minimality of the kernel makes it tedious to use directly, the tools offered by Scala (and especially Scala 3) allow us to design a more intuitive DSL, similar to other proof assistants, directly within the host language. Moreover, essentially all the verification related to the syntactic construction and writing of the proof are checked at compilation time, leaving only the wrong use of proof steps and tactics (such as when trying to prove an invalid statement) as possible failure at runtime. LISA's interface encapsulates the kernel and provides convenient tools and syntax to make mathematical development easier to write and read. Figure 5 shows a minimal example of how to use the DSL to write a proof. This approach makes LISA programmable. It offers the user the full range of tools of the host language when writing proofs, allowing them to express proofs in novel ways or adapted to different areas of mathematics, similar to writing on paper.

LISA's environment is activated simply by creating an object extending `lisa.Main`. This will make available all the essential features to develop mathematics in LISA. Declarations in lines 2, 3 and 4 define a variable, a schematic predicate of arity one, and a schematic function of arity one, such that their symbols are the same as their Scala name, i.e., respectively `"x"`, `"P"` and `"f"`. This is made possible by implicit arguments and reflection. Line 6 starts the declaration of a theorem. (Note that the kernel itself does not rely on such specific features; we expect the kernel to be straightforward to implement in most languages.)

```
1  object Exercise extends lisa.Main {
2     val x = variable
3     val P = predicate(1)
4     val f = function(1)
5
6     val fixedPointDoubleApplication = Theorem(
7           ∀(x, P(x) ⟹ P(f(x))) ⊢ P(x) ⟹ P(f(f(x)))
8        ) {
9        assume(∀(x, P(x) ⟹ P(f(x))))
10       val step1 = have(P(x) ⟹ P(f(x))) by InstantiateForall
11       val step2 = have(P(f(x)) ⟹ P(f(f(x)))) by InstantiateForall
12       have(thesis) by Tautology.from(step1, step2)
13    }
14 }
```

Figure 5 A small proof written with LISA's DSL. Unicode characters are obtained in practice through ligatures or Scala's direct support for unicode.

3.1 Higher-Level Proofs

LISA's interface defines a proof constructing class. This class uses proof tactics to generate pieces of the final pure sequent calculus proof, which are encapsulated into kernel subproofs. The result from the point of view of the user is the ability to define arbitrarily computed deduced proof steps (here `Tautology` and `InstantiateForall`) from the base steps of sequent calculus. Thanks to Scala 3's implicit functions types [32], the proof constructor is automatically created in the code block following the `Theorem` declaration (line 6 of Figure 5) without the need for the user to even realize it exists. The existence of an implicit proof constructor in scope is necessary for the other keywords (`have`, `assume`, ...) to be well-defined, meaning that using those outside of a theorem environment will fail to compile.

The `assume` keyword (line 9) allows stating a formula that will be assumed true for the rest of the proof. Technically, it will be considered as part of the left-hand-side of any further written sequent in the proof. `have` states a proposition that can be reached using a proof tactic (or a subproof, see next example). If a step requires some premises, they can be given as parameters to the tactic, as in line 12. `have` produces a `Fact` that can be used by later steps.

The example in Figure 6 illustrates a more advanced proof structure, using axioms from set theory. As the proof independently proves both directions of the double implication, it makes use of the `subproof` construction. Similarly to the `Theorem` keyword, this construction implicitly creates a new proof constructor environment, internal to the outer proof and with its own goal. In a proof, a `Fact` is a type that contains external theorems, axioms and definitions, as well as previously proven steps from the current or outer proof, but not from any proof that is not a direct ancestor of the current proof. This is made possible by using recursively defined path-dependent types (see Figure 7) and can be checked at compile-time.

Moreover, a fact can also be one of the above, accompanied by information about a specific instantiation of schematic symbols. The actual instantiation step is then carried automatically. This is done in practice with the `of` keyword, as in line 11.

When a tactic requires a single premise, and this premise is the most recently proven fact, `thenHave` passes said premise directly to the tactic without the step having to be named. For some tactics, such as the `Substitution` step at line 21, the resulting sequent will be inferred by the tactic and isn't required to be given by the user. In this case, `have` and `thenHave`

```
1  val unionOfSingleton = Theorem( (union(singleton(x)) ≡ x) ) {
2     val X = singleton(x)
3     val forward = have( (in(z, x) ⟹ in(z, union(X))) ) subproof {
4        ...
5     }
6     val backward = have( in(z, union(X)) ⟹ in(z, x) ) subproof {
7        have(in(z, y) ⊢ in(z, y)) by Restate
8        val step2 = thenHave((y≡x, in(z, y)) ⊢ in(z, x))
9           by Substitution
10       have(in(z, y) ∧ in(y, X) ⊢ in(z, x))
11          by Tautology.from(pairAxiom of (y→x, z→y), step2)
12       val step4 = thenHave(∃(y, in(z, y) ∧ in(y, X)) ⊢ in(z, x))
13          by LeftExists
14       have( in(z, union(X)) ⟹ in(z, x))
15          by Tautology.from(unionAxiom of (x → X), step4)
16    }
17    have( in(z, union(X)) ⟺ in(z, x))
18       by RightIff(forward, backward)
19    thenHave( forall(z, in(z, union(X)) ⟺ in(z, x)))
20       by RightForall
21    andThen(Substitution(extensionalityAxiom of (x → union(X), y → x)))
22 }
```

Figure 6 A LISA proof with more advanced construction.

takes the tactic as argument. The `Tautology` step proves statements using propositional laws and the `Substitution` makes substitution of equals for equals, either everywhere or using unification to find the specific occurrences to replace.

```
1  class Proof {
2     class ProofStep {...}
3     class InnerProof extends Proof {
4        val parent:Proof.this.type = Proof.this // The encapsulating proof
5        type Fact = parent.Fact | this.ProofStep
6     }
7  }
8  class BaseProof extends Proof {
9     type Fact = Theorem | Axiom | Definition | this.ProofStep
10 }
```

Figure 7 Simplified outline of the type structure for proof constructors and their facts.

Definitions. Transparent definitions come for free with the Scala host language (see line 2 of Figure 6), these are not visible to the kernel. The DSL offers syntax for the *non-transparent definitions*. Predicate symbol and function symbol (of which constant symbols are a special type) definitions can be direct, as illustrated by the two first examples in Figure 8. Function symbols can also be defined by unique existence, as shown in the last example. Note that this is an example of an *underspecified definition*, as mentioned in the previous Section. It defines a constant symbol `nonEmpty` with only the property `¬(nonEmpty≡∅)`, but the given proof shows the existence of a specific non-empty set.

```
1 val succ = DEF(x) → union(uPair(x, singleton(x)))
2 val inductive = DEF(x) → in(∅, x) ∧ ∀(y, in(y, x) ⟹ in(succ(y), x))
3 val nonEmptySetExists = Lemma( ∃!(x, ¬(x ≡ ∅) ∧ (x ≡ uPair(∅, ∅))){...}
4 val nonEmpty = DEF() → The(x, ¬(x ≡ ∅))(nonEmptySetExists)
```

Figure 8 Definitions in LISA.

4 Tactics in LISA and Comparison

Developing proof tactics in proof assistants where the proof-writing language is different from the host language (and sometimes when both are different from the tactic-writing language) tends to exhibit high entry barriers for newcomers. They require learning multiple new languages and how they interact with each other. This difficulty can be observed for example with the length of the tactic-writing tutorial for Isabelle [40], or in the Coq Reference Manual, where the Ltac tactic language [13] is described as *having unclear semantics, being slow, non-uniform, error-prone* and even lacking essential programming features such as *data structures*. Ltac2 [24], yet another tactic language, aims to solve *some* of these problems. Newly developed systems, such as Lean [12], have the advantage of being designed from scratch and addressing these problems. We have similar aims with LISA, but rely on an existing programming language which has already solved those issues, has an active user base that draws on more than the development of theorems and has well-developed and actively maintained IDEs and libraries. In particular, for a LISA user, seeing how a proof tactic works is ever only a ctrl-click away from their proof and when a new tactic is written, using it is as simple as writing `import MyTactic`.

Not unlike in HOL Light, where a proof tactic is essentially any function returning a value of type Theorem, a tactic in LISA is simply a function returning a proof or an error message. The tactic can take arbitrary arguments, such as a target sequent and known facts (which will be imports of the resulting proof) and can access the current state of the proof constructor (if needed). To write a low level or highly optimized proof tactic, the user can directly construct a sequent calculus proof and give it to the kernel, but they can also use LISA's DSL directly inside the body of the function and use pre-existing proof tactics. Writing a tactic then consists in writing a generic LISA proof computationally.

LISA defines tactics that correspond to each basic proof step within the kernel, but with all the parameters automatically inferred. These tactics are intended for didactic purpose. Compared to directly using kernel proof steps, these simple tactics are more convenient to write, but also slightly less efficient to check because the system needs to compute the parameters of the proof step. Moreover, most of these simple tactics are subsumed by more general tactics.

4.1 A Proof-Producing SAT Solver Using F(OL)^2

The Tautology tactic is able to prove any valid sequent that requires only propositional reasoning. It is based on a simple proof-producing DPLL-like [10] procedure complete for propositional logic. The procedure makes decisions on atoms, so the worst case complexity is exponential in the number of unique atoms in the formula. It is a non-clausal solver (like, e.g., [25]) whose unique aspect is that, between each decision, it simplifies the propositional formula using the algorithm presented in Subsection 2.2). In the context of proving validity as in LISA as well as when trying to find a satisfying assignment in a SAT solver, this allows

to close branches early in the exploration of the decision tree, or simply to eliminate atoms before they even need to get decided. Moreover, this procedure does not need to compute Tseytin's normal form, avoiding creating more atoms, and conveniently allows producing a proof of the statement. Figure 9 sketches the proof search procedure as it is implemented in LISA. Our current implementation uses a simple decision heuristics that picks the atom that occurs most frequently. Further work may also include extension of the algorithm with quantifier reasoning, to obtain a complete procedure for FOL.

```
def solveFormula(f: Formula,
            decisionsPos:List[Formula],
            decisionsNeg:List[Formula]): ProofTacticJudgement = {
  val redF = reduceWithFol2(f)
  if (redF == ⊤) {
    Restate(decisionsPos ⊢ f :: decisionsNeg)
  } else if (redF == ⊥) {
    InvalidProofTactic("Sequent is not a propositional tautology")
  } else {
    val atom = findBestAtom(redF)
    val substInRedF: Formula => Formula = (f => RedF[atom:=f])
    TacticSubproof {
      have(solveFormula(substInRedF(⊤), atom::decisionsPos, decisionsNeg))
      val step2 = thenHave(atom :: decisionsPos ⊢ redF :: decisionsNeg)
            by Substitution(⊤ <=> atom)
      have(solveFormula(substInRedF(⊥), decisionsPos, atom::decisionsNeg))
      val step4 = thenHave(decisionsPos ⊢ redF :: atom :: decisionsNeg)
            by Substitution(⊥ <=> atom)
      thenHave(decisionsPos ⊢ redF :: decisionsNeg)
            by Cut(step2, step4)
      thenHave(decisionsPos ⊢ f :: decisionsNeg)
            by Restate
    }
  }
}
```

Figure 9 Outline of the F(OL)^2-based solver. Note that the actual implementation produces directly kernel proofs for optimization. Each recursive call to `solveFormula` adds at most 4 kernel steps to the final proof.

Thanks to properties of ortholattices, the solver is already capable of resolving propositional problems that are too difficult for some proof assistants. As an example, we found that Isabelle's Blast tactic (a general tableau prover, [34]) was in general not able to prove the equivalence of two reasonably large formulas made only of variables, disjunctions and conjunctions which only differed in the ordering of their arguments. On the other hand, this is instantaneous (one step) with our described OL-based approach.

4.2 Error Reporting

LISA's DSL also contains a printer for proof (both kernel and high level) and defines specialized error reporting. Tactics are allowed to fail if they are used incorrectly and return an error. Figure 10 shows LISA's output for an incorrect proof, with the current state of the proof, the faulty step, its line number and the error message from the tactic.

```
∀x.  P(x) ⟹ P(f(x)) ⊢ P(x) ⟹ P(f(f(x)))
  0 Hypothesis ∀x. P(x) ⟹ P(f(x)) ⊢ ∀x. P(x) ⟹ P(f(x))
  1 Hypothesis P(x); ∀x. P(x) ⟹ P(f(x)) ⊢ P(x)
  2 InstantiateForall P(x); ∀x. P(x) ⟹ P(f(x)) ⊢ P(x) ⟹ P(f(x))
  3 InstantiateForall P(x); ∀x. P(x) ⟹ P(f(x)) ⊢ P(f(x)) ⟹ P(f(f(x)))
  have(thesis) by Tautology.from(step1)

Proof tactic Tautology used in (Example.scala:47) did not succeed:
   The statement is not provable within propositional logic.
   The proof search needs the truth of the following sequent:
   P(f(x)); P(x); ∀x. ¬(P(x) ∧ ¬P(f(x))) ⊢ P(f(f(x)))
```

Figure 10 LISA's output when the step in line 12 of proof in Figure 5 is incorrectly modified to not use `step2`. The indicated sequent in fact corresponds to `step2`.

5 Beginning Set Theory Development and Cantor's theorem

In this section, we present a brief overview of the current mathematical development in LISA and outline an example of a short proof in set theory.

Inspired by Mizar[31] and Isabelle/HOTG[6] we make the choice of Tarski-Grothendieck set theory (TG) as the axiomatic foundation for LISA's associated mathematical library. As the main reference for the ZFC aspect of the set theory development, we follow Thomas Jech's book *Set Theory* [26]. In the future, we plan to use the axiom on Grothendieck universes (corresponding to the existence of certain large cardinals) to support the embedding of category theory and of systems such as Coq [44].

5.1 Current Theory Development

The mathematical library in LISA begins with the ZFC (and TG) axioms, defining the basic constructs and operations on sets, the subset relation, the empty set, power sets, and unordered pairs. On top of thes axioms, we define structures such as ordered pairs, relations, and functions. Relations are sets of ordered pairs drawing elements from a set, and functions are relations which contain the graph of the function. Function symbols have as a domain the whole set space and must not be mixed with function objects, which are special sets and considered as constants in the light of first order logic. During exploratory development, proofs involving case analysis on these basic structures required significant manual effort, but the `Tautology` and `Substitution` tactics as well as the quick instantiation of axioms and theorems offered by the `of` keyword tend to automate most tedious manipulations. Formalization of partial orders, well-ordered sets, ordinals and induction [26, Chapters 2, 3] is ongoing.

Technically, we define for sets A and B the set of relations from A to B as the power set of their Cartesian product $P(A \times B)$, and its restriction to functional relations, $A \to B$, the set of all functions with domain equal to A and their codomain included in B.

A function symbol can always apply to any term, meaning we cannot rely on well-definedness of terms to define symbols with partial function semantics. Considering the unique existence requirement for definitions, the standard approach consists in extending the limited domain of the partial function by assigning a default value, for example the empty set, to all inputs where it should be undefined, constructing a unique object. This specific construction and default value can then be forgotten using an underspecified definition. In

particular, interpretations of the function with all combinations of values outside the fixed domain will be valid models for the symbol, and no non-trivial property can be proved about those values.

For example, consider the definition for function application, $\text{app}(f, x)$. When f is not a functional relation, or x is not in its domain, we fix $\emptyset$ as the default value in order to obtain a proof of existence and uniqueness.

```
val appDefinition = Theorem( ∀(f, (∀ x, (∃!(z,
      functional(f) ∧ in(x, dom(f)) ⟹ in(pair(x, z), f)))) )
   ∧ ¬functional(f) ∨ ¬in(x, dom(f)) ⟹ z ≡ ∅)
```

We can then obtain the function symbol `app` with only the desired property using an underspecified definition:

```
val app = DEF (f, x) → The(z,
         functional(f) ∧ in(x, dom(f)) ⟹ in(pair(x, z), f))
         (appDefinition)
```

Cantor's Theorem

Finally, several of these definitions and lemmas build up to the formalization of Cantor's theorem, stating that there is no surjection from any set to its power set:

```
val cantorTheorem = Theorem( ¬surjective(f, x, powerSet(x)) )
```

where f and x are schematic set variables, making the sequent implicitly universally quantified. The proof of Cantor's theorem is about 25 lines of code [1]. Internally, the proof expands to 130 sequent calculus steps.

Cantor's theorem is the first theorem formalized in LISA from the list *Formalizing 100 Theorems* [45]. While not a difficult theorem, it requires some ground development and definitions related to set-theoretic functions and relations. The proof itself requires handling the quantifiers for a contradiction construction and combining lemmas about surjective functions. Much of the latter is achieved using `Tautology`. It shows that LISA is capable of non-trivial mathematical development. We expect future developments to become easier and faster with gradual development of reasoning tools and proofs.

6 Related Work

A polynomial algorithm for free ortholattices was presented in [18]. A weaker structure with log linear complexity was first presented in [19]. In LISA we use the ortholattice normal form for first-order logic formulas. Our F(OL)[2] implementation does not aim to be complete for structures such as quantum monadic algebras that treat extensions of OL (and orthomodular lattices) to monadic first-order logic [21].

Much of what we described is concerned with the schematic first-order logic kernel. We chose to include schematic variables to be able to state explicitly the axiom schemas of Zermelo-Fraenkel set theory and its extensions, as well as theorem schemas. Another way to generalize schematic second-order variables would be to use higher-order logic. This is the approach pursued by Isabelle as a framework, and instantiated in Isabelle/ZF.

[1] https://github.com/epfl-lara/lisa/blob/fc37f2a6e879d5f43679a4476c1d6e4685bb14a2/src/main/scala/lisa/mathematics/SetTheory.scala#L1700

The choice of set theory may be considered unusual by some, as Coq [4], Lean [12], the HOL-family [22] and Isabelle/HOL [43] are based either on type theory or on higher-order logic. We consider HOL to be one of the most elegant formulations for formal proof developments. However, set theory is arguably the most widely recognized foundation of mathematics in the mathematical community, and, despite type-theory based tools having the advantage of being easier to express formalisms in from the get-go, we believe that through the development and use of abstracting tactics, a soft-type system and adequate tools, more familiarity and flexibility in writing proofs can be achieved with a set-theory based mathematical library. We also hope to provide a test bed to explore direct first-order foundations as an alternative to the many current systems based on higher-order logic. Concrete results in Mizar [31], Isabelle/ZF [20], ZF in Isabelle/HOL [6, 36], and TLA$^+$ [9, 37] suggest substantial relevance of set-theoretic foundations. Arguments in favour of set theoretic foundations have also been discussed by John Harrison [23] and Bohua Zhan [46].

Even one more level of indirection than in Isabelle/ZF is present in Isabelle/HOL/TG [6], which develops the Tarski-Grothendieck extension of ZF inside Isabelle/HOL. Whereas our system is less flexible and does not currently connect to such a well-developed ecosystem as Isabelle/HOL has, our hope is that it is conceptually simpler thanks to fewer layers and a kernel that does not rely on unification.

Another modern approach to theorem proving is Lean [12], a proof assistant based on dependent type theory and inspired in part by Coq. We believe Lean makes significant improvements over older proof assistant regarding the *Six Virtues*. In particular, it has a strong focus on programmability, with the new version of Lean [30] even having a compiler written in its own proof language. While LISA and Lean's design objectives share similarities, their strategies and specific choice (foundations, language, interface) are different.

To automate proofs that do not instantiate schematic formulas we hope to make use of proof generating theorem provers, such as Vampire [28], SPASS [42], E [38], as well as SMT solvers [2]. Higher-order provers such as Zipperposition [41], Leo-III [39], and Satallax [5] would further increase automation even in the case of axiom schema instantiation.

7 Conclusion

LISA is both a proof system for automated tools and a proof assistant based on first order logic and set theory. It uses Scala as both a host language and a proof writing language, relying on the advanced features it offers to make the system as programmable as the user desires. LISA is strongly committed to interoperability. In particular, it has a small logical kernel which has guaranteed complexity and completeness characterizations, simple foundations and explicit proofs checkable without context. Moreover, it can be compiled into a Scala and Java library. All these properties should favour transfer of proofs from and to other proof systems and uses of LISA as a tool for program verification. To improve usability and reduce the size of proofs, LISA makes use of an efficient normal form algorithm for propositional logic extended to first order logic. This algorithm is also the basis for a complete propositional proof-producing procedure implemented in LISA as a tactic.

LISA is still under active development, but already proposes an advanced proof writing DSL not entirely dissimilar to already existing interpreted languages in other assistants. LISA also allows defining arbitrary tactics in a simple way and has specialized error reporting. The current embryo of set-theoretic development encompasses properties of relations and functions, and in particular Cantor's theorem has been successfully proven.

References

1 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.

2 Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A Versatile and Industrial-Strength SMT Solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 13243, pages 415–442. Springer International Publishing, Cham, 2022. doi:10.1007/978-3-030-99524-9_24.

3 Ladislav Beran. *Orthomodular Lattices (An Algebraic Approach)*. Springer Dordrecht, 1985. doi:10.1007/978-94-009-5215-7.

4 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-Verlag, Berlin Heidelberg, 2004. doi:10.1007/978-3-662-07964-5.

5 Chad E. Brown. Satallax: An automatic higher-order prover. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning*, pages 111–117, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

6 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.

7 Gunter Bruns and John Harding. Algebraic aspects of orthomodular lattices. In Bob Coecke, David Moore, and Alexander Wilce, editors, *Current Research in Operational Quantum Logic: Algebras, Categories, Languages*, pages 37–65. Springer Netherlands, Dordrecht, 2000. doi:10.1007/978-94-017-1201-9_2.

8 Paul J. Cohen. The independence of the continuum hypothesis. *Proceedings of the National Academy of Sciences of the United States of America*, 50(6):1143–1148, 1963. URL: http://www.jstor.org/stable/71858.

9 Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. TLA+ proofs. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods*, pages 147–154, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

10 Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962. doi:10.1145/368273.368557.

11 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.

12 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*, volume 9195, pages 378–388. Springer International Publishing, Cham, 2015. doi:10.1007/978-3-319-21401-6_26.

13 David Delahaye. A tactic language for the system coq. In *LPAR*, volume 1955, pages 85–95. Springer, 2000.

14 Harald Ganzinger, Christoph Meyer, and Christoph Weidenbach. Soft typing for ordered resolution. In William McCune, editor, *Automated Deduction - CADE-14, 14th International Conference on Automated Deduction, Townsville, North Queensland, Australia, July 13-17, 1997, Proceedings*, volume 1249 of *Lecture Notes in Computer Science*, pages 321–335. Springer, 1997. doi:10.1007/3-540-63104-6_32.

15 G. Gentzen. Untersuchungen über das logische schließen i. *Mathematische Zeitschrift*, 39:176–210, 1935. URL: http://eudml.org/doc/168546.

16 Michael J. C. Gordon, Robin Milner, Christopher P. Wadsworth, and P. Ted Christopher. Edinburgh LCF: a mechanized logic of computation. *Lecture Notes in Computer Science*, 1978.

17 Simon Guilloud. LISA Reference Manual. EPFL-LARA, February 2023.

18 Simon Guilloud, Mario Bucev, Dragana Milovancevic, and Viktor Kunčak. Formula normalizations in verification. Technical Report 297701, EPFL, 2023. URL: http://infoscience.epfl.ch/record/297701.

19 Simon Guilloud and Viktor Kunčak, editors. *Equivalence Checking for Orthocomplemented Bisemilattices in Log-Linear Time*. Springer, 2022. doi:10.48550/arXiv.2110.03315.

20 Emmanuel Gunther, Miguel Pagano, Pedro Sánchez Terraf, and Matías Steinberg. The independence of the continuum hypothesis in isabelle/zf. *Archive of Formal Proofs*, March 2022. , Formal proof development. URL: https://isa-afp.org/entries/Independence_CH.html.

21 J Harding. Quantum monadic algebras. *Journal of Physics A: Mathematical and Theoretical*, 55(39):394001, September 2022. doi:10.1088/1751-8121/ac845b.

22 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.

23 John Harrison. Let's make set theory great again! In *Axiomatic Set Theory*, page 46, Aussois, 2018.

24 CNRS Inria and contributors. Ltac2 — Coq 8.16.1 documentation. URL: https://coq.inria.fr/refman/proof-engine/ltac2.html.

25 Himanshu Jain, Constantinos Bartzis, and Edmund Clarke. Satisfiability checking of non-clausal formulas using general matings. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, pages 75–89, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. doi:10.1007/11814948_10.

26 Thomas Jech. *Set theory: The third millennium edition, revised and expanded*. Springer, 2003.

27 Deepak Kapur and Paliath Narendran. Complexity of unification problems with associative-commutative operators. *J. Autom. Reason.*, 9(2):261–288, 1992. doi:10.1007/BF00245463.

28 Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 1–35, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-39799-8_1.

29 Norman Megill. Metamath. *The Seventeen Provers of the World: Foreword by Dana S. Scott*, pages 88–95, 2006.

30 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.

31 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.

32 Martin Odersky, Aggelos Biboudis, Fengyun Liu, and Olivier Blanvillain. Foundations of implicit function types. Technical report, EPFL, 2017. URL: http://infoscience.epfl.ch/record/229203.

33 Lawrence C. Paulson. Isabelle: The next 700 theorem provers. *CoRR*, cs.LO/9301106, 1993. URL: https://arxiv.org/abs/cs/9301106.

34 Lawrence C. Paulson. A generic tableau prover and its integration with Isabelle. *JUCS - Journal of Universal Computer Science*, 5(3):73–87, 1999. doi:10.3217/jucs-005-03-0073.

35 Lawrence C Paulson. Isabelle's logics: FOL and ZF, 2013.

36 Lawrence C. Paulson. Zermelo Fraenkel set theory in higher-order logic. *Archive of Formal Proofs*, October 2019. , Formal proof development. URL: https://isa-afp.org/entries/ZFC_in_HOL.html.

37 TLA Proof System Project. TLA+ proof system. URL: https://tla.msr-inria.inria.fr/tlaps/content/Home.html.

38 Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, Lecture Notes in Computer Science, pages 735–743, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-45221-5_49.

39 Alexander Steen. Leo-iii 1.7, July 2022. doi:10.5281/zenodo.7650205.

40 Christian Urban. The Isabelle Cookbook. URL: https://web.cs.wpi.edu/~dd/resources_isabelle/isabelle_programming.urban.pdf.

41 Petar Vukmirović, Alexander Bentkamp, Jasmin Blanchette, Simon Cruanes, Visa Nummelin, and Sophie Tourret. Making Higher-Order Superposition Work. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, Lecture Notes in Computer Science, pages 415–432, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-79876-5_24.

42 Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS Version 3.5. In Renate A. Schmidt, editor, *Automated Deduction – CADE-22*, Lecture Notes in Computer Science, pages 140–145, Berlin, Heidelberg, 2009. Springer. doi:10.1007/978-3-642-02959-2_10.

43 Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle Framework. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, pages 33–38, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-71067-7_7.

44 Benjamin Werner. Sets in types, types in sets. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Martín Abadi, and Takayasu Ito, editors, *Theoretical Aspects of Computer Software*, volume 1281, pages 530–546. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. doi:10.1007/BFb0014566.

45 Freek Wiedijk. Formalizing 100 theorems. https://www.cs.ru.nl/~freek/100/.

46 Bohua Zhan. Formalization of the Fundamental Group in Untyped Set Theory Using Auto2. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving*, Lecture Notes in Computer Science, pages 514–530, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-66107-0_32.

Semantic Foundations of Higher-Order Probabilistic Programs in Isabelle/HOL

Michikazu Hirata
School of Computing, Tokyo Institute of Technology, Japan

Yasuhiko Minamide
School of Computing, Tokyo Institute of Technology, Japan

Tetsuya Sato
School of Computing, Tokyo Institute of Technology, Japan

Abstract

Higher-order probabilistic programs are used to describe statistical models and machine-learning mechanisms. The programming languages for them are equipped with three features: higher-order functions, sampling, and conditioning. In this paper, we propose an Isabelle/HOL library for probabilistic programs supporting all of those three features. We extend our previous quasi-Borel theory library in Isabelle/HOL. As a basis of the theory, we formalize *s-finite kernels*, which is considered as a theoretical foundation of first-order probabilistic programs and a key to support conditioning of probabilistic programs. We also formalize the Borel isomorphism theorem which plays an important role in the quasi-Borel theory. Using them, we develop the *s-finite measure monad* on quasi-Borel spaces. Our extension enables us to describe higher-order probabilistic programs with conditioning directly as an Isabelle/HOL term whose type is that of morphisms between quasi-Borel spaces. We also implement the *qbs prover* for checking well-typedness of an Isabelle/HOL term as a morphism between quasi-Borel spaces. We demonstrate several verification examples of higher-order probabilistic programs with conditioning.

2012 ACM Subject Classification Theory of computation → Denotational semantics; Mathematics of computing → Probabilistic algorithms

Keywords and phrases Higher-order probabilistic program, s-finite kernel, Quasi-Borel spaces, Isabelle/HOL

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.18

Funding *Michikazu Hirata*: supported by JSPS Research Fellowships for Young Scientists and JSPS KAKENHI Grant Number 23KJ0905
Yasuhiko Minamide: supported by JSPS KAKENHI Grant Number 19K11899 and 20H04162
Tetsuya Sato: supported by JSPS KAKENHI Grant Number 20K19775

1 Introduction

Probabilistic programs are used to describe statistical models and machine-learning mechanisms. Programmers can conduct statistical inference just by writing statistical models as programs, without implementing complex inference algorithms by themselves. Higher-order probablistic programming languages, e.g. Anglican [29] and Church [9], integrate fundamental features of probabilistic programming languages such as sampling and conditioning into expressive higher-order funcaional languages and have been an active research topic recently.

Let us see a concrete example by Staton [25, Section 2.2]. The following probabilistic program uses two language features: higher-order functions and conditioning.


© Michikazu Hirata, Yasuhiko Minamide, and Tetsuya Sato;
licensed under Creative Commons License CC-BY 4.0

```
1.  λf. do {
2.    let T = uniform(0,24) in
3.    query T (λt. let r = f t in
4.                 exponential_pdf r 0.0167)
5.    }
```

This program is higher-order since `f` is given as a parameter. The `query` command at line 3 receives a prior distribution and a likelihood, and then returns the posterior distribution. We will explain details of this program in Section 4.2.

Basically, probabilistic programs are interpreted as measurable functions between measurable spaces. Various measure-theoretic structures such as the Giry monad [8] and s-finite kernels [24] are used for such semantic models. However, there is a difficulty to interpret higher-order functions. The result by Aumann [2] implies that there is no suitable measurable space corresponding to the function type `real` ⇒ `real`. In order to overcome this difficulty, Heunen et al. have introduced quasi-Borel spaces and the probability monad on it [10]. The theory provides a suitable denotational semantics for higher-order probabilistic programs. Ścibior et al. have developed the s-finite measure monad[1] on quasi-Borel spaces [22], which enable us to treat infinite measures and to denote higher-order probabilistic programs with conditioning.

In previous work, we have formalized the quasi-Borel spaces and the probability monad in Isabelle/HOL [11]. Using them, we have verified the Monte Carlo approximation algorithm. Our previous work can treat probabilistic programs supporting higher-order functions and sampling but not conditioning. Affeldt et al. have formalized s-finite kernels in Coq [1]. They have embedded a probabilistic program using s-finite kernels. Their work can treat probabilistic programs supporting sampling and conditioning but not higher-order functions.

In this paper, we propose an Isabelle/HOL library for probabilistic programs supporting all of higher-order functions, sampling, and conditioning by extending our previous work. Our contributions are the following.

1. We formalize s-finite kernels and the Borel isomorphism theorem. They are a theoretical basis of quasi-Borel theory, especially a basis of the s-finite measure monad.
2. We develop proof automation for checking *well-typedness* of probabilistic programs and construct the s-finite measure monad.
3. We implement several program examples from previous works and prove their properties.

Our library enables us to interpret an Isabelle/HOL term as a probabilistic program and that makes it easier to write probabilistic programs and reason about them in Isabelle/HOL. Our *qbs prover* for automated *type checking* is also helpful to reason about probabilistic programs. Both of previous formalizations by us [11] and Affeldt et al. [1] use de Bruijn index to describe programs, which makes it harder to read and write programs. Our previous work spent around 450 lines to prove integrability and the weak law of large numbers of the Monte Carlo approximation algorithm, while we have spent around 140 lines to prove them in our new formalization.

In Section 2, we review the standard library for measure theory in Isabelle/HOL. Then we formalize s-finite kernels and the Borel isomorphism theorem. In Section 3, we review our previous formalization of quasi-Borel spaces. Then we discuss our proof automation and formalization of the s-finite measure monad. In Section 4, we show three verification examples of probabilistic programs. In Section 5, we conclude our work.

[1] Details of definition vary among prior studies. In their original paper, they have introduced the σ-finite measure monad. Later Vákár et al. reformulate it as s-finite measure monad [27, 28](see also Section 3.3).

2 Measure Theory

Measure theory is a theoretical basis of probability theory and quasi-Borel theory. We first review the standard definitions of measure theory library in Isabelle/HOL. Then we formalize s-finite kernels which are used to construct the s-finite measure monad in Section 3.3. We also formalize the Borel isomorphism theorem, which plays an important role in quasi-Borel theory.

2.1 Measure Theory in Isabelle/HOL

We use the Isabelle/HOL's libraries: HOL-Analysis and HOL-Probability [3, 7, 12, 13, 16, 18]. The type *'a measure* denotes the type of measures on the type *'a*. A measure $M :: 'a\ measure$ consists of three components:

$$space\ M :: 'a\ set, \qquad sets\ M :: 'a\ set\ set, \qquad emeasure\ M :: 'a\ set \Rightarrow ennreal,$$

where the type *ennreal* denotes the type of extended non-negative real numbers. They correspond to the space, the measurable sets, and the measure, respectively. We often write M for *emeasure M* using a coercion. The triple $(space\ M, sets\ M, emeasure\ M)$ forms a measure space, that is, *sets M* is a σ-algebra on *space M* and *emeasure M* is a countably additive function on *sets M* such that $emeasure\ M\ \emptyset = 0$. We use a measure M as a measurable space when we are not interested in its measure. The library defines the constant $borel :: ('a :: topological\text{-}space)\ measure$ on topological space type class. The *borel* denotes the Borel space, that is, $sets\ borel = \sigma[\{U.\ open\ U\}]$ is the least measurable sets including all open sets. We denote the borel space on real numbers by $\mathbb{R}$, the borel space on extended non-negative real numbers by $\mathbb{R}_{\geq 0}$, the discrete space on natural numbers by $\mathbb{N}$, and the discrete space on boolean by $\mathbb{B}$.

A function f from *space M* to *space N* is called measurable if $f^{-1}\ A \cap space\ M \in sets\ M$ for all $A \in sets\ N$. The set of measurable functions from M to N is denoted by $M \to_M N$. For a measurable function $f \in M \to_M \mathbb{R}$, the Lebesgue integral of f w.r.t. M is denoted by[2] $\int x.\ f\ x\ \partial M$. The Lebesgue integral on a restricted set A is denoted by $\int x{\in}A.\ f\ x\ \partial M$.

For a measure M, the predicate *subprob-space M* means that M is a sub-probability space, that is, $M\ (space\ M) \leq 1$. The predicate *finite-measure M* means that M is a finite measure, that is, $M\ (space\ M) < \infty$. The predicate *sigma-finite-measure M* means that M is a σ-finite measure, that is, there exists a countable disjoint measurable sets $\forall i{::}nat.\ A\ i \in sets\ M$ such that $(\bigcup i.\ A\ i) = space\ M$ and $\forall i.\ M\ (A\ i) < \infty$.

Throughout this paper, we use the following constructions.

The Lebesgue Measure[3]	$sets\ lborel = \sigma[\{U.\ open\ U\}]$
	$emeasure\ lborel\ (a, b] = b - a$
Product Measure	$sets\ (M \otimes_M N) = \sigma[\{A \times B.\ A \in sets\ M \wedge B \in sets\ N\}]$
	$emeasure\ (M \otimes_M N)\ (A \times B) = emeasure\ M\ A * emeasure\ N\ B$
	where *sigma-finite-measure N*, $A \in sets\ M$, and $B \in sets\ N$.
Image Measure	$sets\ (distr\ M\ N\ f) = sets\ N$
	$emeasure\ (distr\ M\ N\ f)\ A = emeasure\ (f^{-1}\ A \cap space\ M)$
	where $f \in M \to_M N$ and $A \in sets\ N$.

[2] In the library, the real-valued integral and the extended non-negative real-valued integral are defined separately. Although we do not distinguish them for simplicity in this paper.

[3] Strictly speaking, *completion lborel* is the Lebesgue measure.

2.2 S-Finite Kernel

Staton introduced a semantic model of first-order probabilistic programs with conditioning using s-finite kernels [24]. S-finite kernels are suitable for program semantics: they support a bind-like operation satisfying desired equations, which are a basis of the s-finite measure monad on quasi-Borel spaces. We formalize s-finite kernels and related notions. For the terminology of s-finite measures/kernels, we refer to the work by Staton [24].

S-finite Measures. A measure M is called a *s-finite measure* if M is represented as a countable sum of finite measures. All σ-finite measures, such as the Lebesgue measure, are also s-finite measures. We formalize s-finite measures with the **locale** command which introduces a context.

locale *s-finite-measure* =
 fixes M :: *'a measure*
 assumes $\exists$ *Mi* :: *nat* $\Rightarrow$ *'a measure*.
 ($\forall$ *i*. *sets* (*Mi i*) = *sets M*) $\wedge$ ($\forall$ *i*. *finite-measure* (*Mi i*))
 $\wedge$ ($\forall$ *A*$\in$*sets M*. *M A* = ($\sum$ *i*. *Mi i A*))

sublocale *sigma-finite-measure* $\subseteq$ *s-finite-measure*

The symbol $\sum i.$ sums over all natural numbers (i.e. $\sum_{i=0}^{\infty}$ in usual mathematics). We remark that s-finite measures may not be σ-finite in general. For instance, the measure $M\ \{0\} = \infty$ on the singleton space $\{0\}$ is not σ-finite but s-finite, because it is equal to the countable infinite sum of the Dirac measure δ_0.

We have formalized basic lemmas related to s-finite measures. One of the important lemma is a restricted Fubini-Tonelli theorem for reordering iterated integrations. The general Fubini-Tonelli theorem does not hold for s-finite measures because product measures are not determined uniquely. However, the (binary) product measures in Isabelle/HOL work well with the Fubini-Tonelli theorem. In mathematics, the product measure is usually defined as the unique measure satisfying $(M \otimes_M N)\ (A \times B) = M\ A * N\ B$, while Isabelle/HOL's library defines the product measure as $(M \bigotimes_M N)\ A = (\int x.\ (\int y.\ \mathit{indicator}\ A\ (x,y)\ \partial N)\ \partial M)$. Using Isabelle/HOL's definition, we can prove Fubini-Tonelli theorem by almost similar ways as the proofs for σ-finite measures.

S-finite Kernels. Roughly speaking, s-finite kernels are generalization of probabilistic processes that return s-finite measures. They are defined as countable sums of finite kernels. In general, classes of kernels are not closed under compositions, but it is convenient that s-finite kernels are so. We first formalize *measure kernels* with the **locale** command.

locale *measure-kernel* =
 fixes M :: *'a measure*
 and N :: *'b measure*
 and κ :: *'a* $\Rightarrow$ *'b measure*
 assumes $\bigwedge x.\ x \in \mathit{space}\ M \Longrightarrow \mathit{sets}\ (\kappa\ x) = \mathit{sets}\ N$
 and $\bigwedge B.\ B \in \mathit{sets}\ N \Longrightarrow (\lambda x.\ \kappa\ x\ B) \in M \rightarrow_M \mathbb{R}_{\geq 0}$
 and $\mathit{space}\ M \neq \emptyset \Longrightarrow \mathit{space}\ N \neq \emptyset$

The third assumption $\mathit{space}\ M \neq \emptyset \Longrightarrow \mathit{space}\ N \neq \emptyset$ in *measure-kernel* is required in order to define the operator $\gg\!=_k$ in a convenient way, later. We formalize *finite kernels*, *sub-probability kernels*, and *s-finite kernels* as sublocales of measure kernels.

locale *finite-kernel* = *measure-kernel* +
 assumes $\exists r{<}\infty.\ \forall x{\in}\ \mathit{space}\ M.\ \kappa\ x\ (\mathit{space}\ N) < r$

locale *subprob-kernel* = *measure-kernel* +
  assumes $\bigwedge x.\ x \in \mathit{space}\ M \Longrightarrow \mathit{subprob\text{-}space}\ (\kappa\ x)$

locale *s-finite-kernel* = *measure-kernel* +
  assumes $\exists\, ki.\ (\forall\, i.\ \mathit{finite\text{-}kernel}\ M\ N\ (ki\ i)) \wedge (\forall\, x{\in}\mathit{space}\ M.\ \forall\, A{\in}\mathit{sets}\ N.\ \kappa\ x\ A = (\sum i.\ ki\ i\ x\ A)))$

We define the operation $M \gg\!=_k \kappa$ for an s-finite measure M and *measure-kernel* $M\ N\ \kappa$, which satisfies the following properties when M is not an empty space.

$$\mathit{sets}\ (M \gg\!=_k \kappa) = \mathit{sets}\ N, \quad (M \gg\!=_k \kappa)\ B = (\int x.\ (\kappa\ x\ B)\ \partial M)$$

If M is an empty space, we cannot obtain the measurable structure of N from M and κ (recall the definition of *measure-kernel*). Hence, $M \gg\!=_k \kappa$ is set to return the discrete empty space as a *default value*. Due to this definition, we need the assumption *space* $M \neq \emptyset \Longrightarrow$ *space* $N \neq \emptyset$ in *measure-kernel*. Without this assumption, we will get stuck to prove *compositionality* of s-finite kernels later.

The operation *bind*, which has been already defined in the Isabelle/HOL's library, satisfies the same equations as the above equation for $\gg\!=_k$ when κ is a sub-probability kernel. Unfortunately, *bind* is defined through the *join* operator of the Giry monad and thus we do not have the above equations for general measure kernels. Hence we need to introduce the operator $\gg\!=_k$ and prove lemmas similar to ones of *bind*.

The following are important properties for constructing the s-finite measure monad in Section 3.3 (called compositionality, associativity, and commutativity, respectively).

lemma
  assumes *s-finite-kernel* $M\ N\ \kappa$ **and** *s-finite-kernel* $(M \otimes_M N)\ L\ (\lambda(x, y).\ \kappa'\ x\ y)$
  shows *s-finite-kernel* $M\ L\ (\lambda x.\ \kappa\ x \gg\!=_k \kappa'\ x)$

lemma
  assumes *sets* μ = *sets* M
    and *s-finite-kernel* $M\ N\ \kappa$ **and** *s-finite-kernel* $N\ L\ \kappa'$
  shows $\mu \gg\!=_k (\lambda x.\ \kappa\ x \gg\!=_k \kappa') = \mu \gg\!=_k \kappa \gg\!=_k \kappa'$

lemma
  assumes *sets* μ = *sets* M **and** *sets* ν = *sets* N
    and *s-finite-measure* μ **and** *s-finite-measure* ν **and** *s-finite-kernel* $(M \bigotimes_M N)\ L\ (\lambda(x,y).\ f\ x\ y)$
  shows $\mu \gg\!=_k (\lambda x.\ \nu \gg\!=_k (\lambda y.\ f\ x\ y)) = \nu \gg\!=_k (\lambda y.\ \mu \gg\!=_k (\lambda x.\ f\ x\ y))$

The Dirac measure on M is denoted by *return* M in Isabelle/HOL. It forms a unit of $\gg\!=_k$.

lemma
  assumes *sets* M = *sets* N
  shows $M \gg\!=_k$ *return* $N = M$

lemma
  assumes *measure-kernel* $M\ N\ \kappa$
    and $x \in$ *space* M
  shows *return* $M\ x \gg\!=_k \kappa = \kappa\ x$

2.3 The Borel Isomorphism Theorem

We prove the Borel isomorphism theorem. The theorem is a key to construct the *s-finite measure monad* and represent s-finite measures as *measures* on quasi-Borel spaces in Section 3.

A separable complete metrizable topological space is called a Polish space. A measurable space generated from a Polish space is called a standard Borel space. For example, $\mathbb{N}$ and $\mathbb{R}$ are standard Borel spaces. We have the following theorems related to standard Borel spaces.

▶ **Theorem 1** (The Borel isomorphism theorem). *A standard Borel space is either countable discrete space or isomorphic to* $\mathbb{R}$.

▶ **Corollary 2.** *For a non-empty standard Borel space* M*, the following statement holds.* ($\star$) *There exist measurable functions* $\textit{to-real}_M$ *and* $\textit{from-real}_M$ *such that*

$$\textit{to-real}_M \in M \to_M \mathbb{R}\ , \quad \textit{from-real}_M \in \mathbb{R} \to_M M\ ,$$
$$\forall x \in \textit{space}\ M.\ \textit{from-real}_M\ (\textit{to-real}_M\ x) = x\ .$$

We have proved the Borel isomorphism (Theorem 1) mainly referring to the textbook by Srivastava [23] and the lecture note by Biskup [6], which was available online. Corollary 2 follows immediately from the Borel isomorphism theorem[4]. We will use measurable functions in ($\star$) in two situations. One is when we construct the s-finite measure monad on quasi-Borel spaces. The other is when we represent s-finite measures as *measures* on quasi-Borel spaces. In our previous work [11], we defined the standard Borel spaces as measurable spaces which satisfy the condition ($\star$). An advantage of our new formalization is that we can obtain many instances easily with the type classes using the following lemma.

lemma *standard-borel* (*borel* :: ('*a* :: *polish-space*) *measure*)

Here, *standard-borel* M means that M is a standard Borel space. A binary or countable product space of standard Borel spaces is again a standard Borel space.

lemma
  assumes *standard-borel* M **and** *standard-borel* N
  shows *standard-borel* ($M \otimes_M N$)
lemma
  assumes *countable* I **and** $\bigwedge i.\ i \in I \Longrightarrow$ *standard-borel* ($M\ i$)
    shows *standard-borel* ($\Pi_M\ i \in I.\ M\ i$)

In the proof of the Borel isomorphism theorem we use metric spaces and topological spaces. The Isabelle/HOL's libraries include formalization of metric spaces by type classes, and topological spaces by type classes and abstract data types. Type class based formalization is not suitable in our situation because we want to change their metrics or topologies during the proof and work with sub-spaces. Thus we have formalized set-based metric spaces and used the existing library of abstract topology with some extensions. Recent work on *types-to-sets* [15, 17, 19] might be used to simplify our formalization.

3 Quasi-Borel Spaces

The theory of quasi-Borel spaces is introduced by Heunen et al. [10] to give a semantic model of programming language supporting both continuous random samplings and higher-order functions. The theory provides a suitable semantics of higher-order probabilistic programs because quasi-Borel spaces always have function spaces with desired properties while measurable spaces do not in general. Furthermore, s-finite measures on standard Borel spaces are represented as measures on quasi-Borel spaces and integration is also performed in quasi-Borel theory.

We formalized the quasi-Borel spaces and the probability monad in our previous work [11]. In this section, we first review our previous formalization, then discuss our extensions: proof automation for quasi-Borel spaces and formalization of the s-finite measure monad.

[4] In fact, the converse also holds: a measurable space satisfying ($\star$) is a standard Borel space. Hence the condition ($\star$) is another characterization of standard Borel spaces. This fact is called as Kuratowski's theorem by Heunen et al. [10]. We have not proved it yet.

3.1 Quasi-Borel Spaces in Isabelle/HOL

The type *'a quasi-borel* denotes quasi-Borel spaces on the type *'a*. A quasi-Borel space $X :: 'a$ *quasi-borel* has two components:

$$\textit{qbs-space}\ X :: 'a\ \textit{set}, \qquad \textit{qbs-Mx}\ X :: (\textit{real} \Rightarrow 'a)\ \textit{set}.$$

They satisfy the following four conditions.

- If $\alpha \in \textit{qbs-Mx}\ X$ and r is a real number, then $\alpha\ r \in \textit{qbs-space}\ X$.
- If $\alpha \in \textit{qbs-Mx}\ X$ and $f \in \mathbb{R} \to_M \mathbb{R}$, then $\alpha \circ f \in \textit{qbs-Mx}\ X$.
- If $x \in \textit{qbs-space}\ X$, then $(\lambda r.\ x) \in \textit{qbs-Mx}\ X$.
- If $(\forall i.\ \alpha\ i \in \textit{qbs-Mx}\ X)$ and $P \in \mathbb{R} \to_M \mathbb{N}$, then $(\lambda r.\ \alpha\ (P\ r)\ r) \in \textit{qbs-Mx}\ X$.

Intuitively, an element of *qbs-Mx X* is a *random variable* whose sample space is the set of real numbers. We sometimes write $x \in X$ instead of $x \in \textit{qbs-space}\ X$ by declaring a coercion.

The set of morphisms (structure-preserving functions) from $X :: 'a$ *quasi-borel* to $Y :: 'b$ *quasi-borel* is defined as follows.

$$\begin{aligned} X \to_Q Y &:: ('a \Rightarrow 'b)\ \textit{set} \\ X \to_Q Y &= \{f.\ \forall \alpha \in \textit{qbs-Mx}\ X.\ f \circ \alpha \in \textit{qbs-Mx}\ Y\} \end{aligned}$$

Quasi-Borel spaces and morphisms form a Cartesian closed category with countable coproducts. Hence, there always exist product spaces $X \otimes_Q Y$, list spaces *list-qbs* X[5], and function spaces $X \Rightarrow_Q Y$ such that $\textit{qbs-space}\ (X \Rightarrow_Q Y) = X \to_Q Y$. Throughout this paper, we assume that all functions are morphisms. In our extension of quasi-Borel theory library, we define the set of morphisms $X \to_Q Y$ as an abbreviation of $\textit{qbs-space}\ (X \Rightarrow_Q Y)$ for the proof automation presented in Section 3.2.

Connection between Measurable Spaces and Quasi-Borel Spaces

There are conversions between measurable spaces and quasi-Borel spaces. Using the conversions, we can easily derive from theorems in the measure theory library that basic functions, such as $+$ and $-$, are morphisms. The conversions L and R return the following structures.

$$\begin{aligned} L &:: 'a\ \textit{quasi-borel} \Rightarrow 'a\ \textit{measure} \\ \textit{space}\ (L\ X) &= \textit{qbs-space}\ X \\ \textit{sets}\ (L\ X) &= \{U \cap \textit{qbs-space}\ X \mid U.\ \forall \alpha \in \textit{qbs-Mx}\ X.\ \alpha -` U \in \textit{sets}\ \mathbb{R}\} \end{aligned} \qquad \begin{aligned} R &:: 'a\ \textit{measure} \Rightarrow 'a\ \textit{quasi-borel} \\ \textit{qbs-space}\ (R\ M) &= \textit{space}\ M \\ \textit{qbs-Mx}\ (R\ M) &= \mathbb{R} \to_M M \end{aligned}$$

We use a measurable space M as a quasi-Borel space $R\ M$. For instance, the quasi-Borel space $\mathbb{R}$ has the following structure: $\textit{qbs-space}\ \mathbb{R} = \textit{UNIV}$ (the universal set of real numbers) and $\textit{qbs-Mx}\ \mathbb{R} = \mathbb{R} \to_M \mathbb{R}$.

The conversions have the following properties.

▶ **Theorem 3** (cf. [10, Propositions 15]). **(i)** $X \to_Q R\ M = L\ X \to_M M$.
(ii) *If M is a standard borel space, then* $\textit{sets}\ (L\ (R\ M)) = \textit{sets}\ M$

Theorem 3 (i) implies that R and L forms an adjunction between the category of measurable spaces and the category of quasi-Borel spaces, and (ii) implies that the adjunction can be restricted to an adjoint equivalence on standard Borel spaces.

[5] The space of lists on X is defined using the isomorphism $\textit{List}[X] \cong \coprod_{n \in \mathbb{N}} \prod_{0 \le i < n} X$.

3.2 Proof Automation

The Isabelle/HOL's measure theory library provides the automated *measurability prover*. In the context of measure theory, one often needs to show measurability: $A \in \mathit{sets}\ M$ or $f \in M \to_M N$. In pen and paper mathematics, measurability proofs are often omitted since they are trivial, while one needs to show measurability each time in the formal proof. The measurability prover automates such proofs of measurability and greatly reduces the cost of proofs. Similar to measure theory, we often need to prove that some function is a morphism, $f \in X \to_Q Y$, in the context of quasi-Borel theory. We have implemented automated *qbs prover*. Unlike measurable spaces, quasi-Borel spaces have function spaces, hence our qbs prover is similar to type checking of a simply-typed functional programming language.

We construct the qbs prover which tries to prove $x \in \mathit{qbs\text{-}space}\ X$ automatically. The qbs prover can also be used to solve morphism statements $f \in X \to_Q Y$ and $\alpha \in \mathit{qbs\text{-}Mx}\ X$ because we have $X \to_Q Y = \mathit{qbs\text{-}space}\ (X \Rightarrow_Q Y)$ and $\mathit{qbs\text{-}Mx}\ X = \mathbb{R} \to_Q X$.

We regard $(\lambda x.\ e) \in X \Rightarrow_Q Y$ as the typing judgment $x : X \vdash e : Y$, and $e \in \mathit{qbs\text{-}space}\ X$ as $\vdash e : X$. Then solving $x \in \mathit{qbs\text{-}space}\ X$ is equivalent to solving the corresponding typing judgment. The qbs prover tries to solve typing judgments with the following method:

Algorithm. We prepare two sets of introduction rules: Rule_1 and Rule_2. Then repeat the following steps.

- Try to apply a rule in Rule_1.
- If none of the rules in Rule_1 is applied, then try to apply a rule in Rule_2.

Rule_1 and Rule_2 consist of (at least) the following inference rules.

- Rule_1

$$\frac{}{x : X \vdash x : X}\ \mathrm{ID} \qquad \frac{\vdash e : Y}{x : X \vdash e : Y}\ \mathrm{CONST}\ (x \text{ does not occur free in } e)$$

After $e \in \mathit{qbs\text{-}space}\ X$ is proved, it may be added as an axiom of Rule_1.

$$\frac{}{\vdash e : X}\ \mathrm{AXIOMS}$$

- Rule_2

$$\frac{\vdash f : X \Rightarrow_Q Y \qquad \vdash x : X}{\vdash f\ x : Y}\ \mathrm{APP}_1 \qquad \frac{x : X \vdash e_1 : Y \Rightarrow_Q Z \qquad x : X \vdash e_2 : Y}{x : X \vdash e_1\ e_2 : Z}\ \mathrm{APP}_2$$

$$\frac{z : X \otimes_Q Y \vdash f[\mathit{fst}\ z/x, \mathit{snd}\ z/y] : Z}{x : X \vdash (\lambda y.\ f) : Y \Rightarrow_Q Z}\ \mathrm{CURRY}$$

For CURRY, we need to have $\mathit{fst} \in X \otimes_Q Y \Rightarrow_Q X$ and $\mathit{snd} \in X \otimes_Q Y \Rightarrow_Q Y$ as axioms of Rule_1. There are mainly two reasons why we divide the rules. First, the rule CONST might overlap with APP_2 or CURRY. Because the rule CONST should be applied first, we add CONST to Rule_1. The other reason is that to prevent terms from being split in certain situation. We sometimes add rules for composition of terms, for example $\mathit{emeasure}\ M\ A \in \mathbb{R}_{\geq 0}$, to Rule_1. If we apply a rule in Rule_2 first, then the composed term will be split by the rule APP_1 or APP_2, that is not what we want the prover to do.

The following code is an example usage of the qbs prover.

```
lemma
  assumes [qbs]: f ∈ ℝ ⇒Q ℝ
  shows (λx. 1 + f x) ∈ ℝ ⇒Q ℝ
  by qbs
```

In the above code, we add $f \in \mathbb{R} \Rightarrow_Q \mathbb{R}$ to the axioms of Rule_1 using the attribute $[qbs]$. Rule_1 is configured by our library so that the axioms contain $r \in \mathbb{R}$ and $(+) \in \mathbb{R} \Rightarrow_Q \mathbb{R} \Rightarrow_Q \mathbb{R}$. Then we call the qbs prover by the tactic *qbs* which immediately solves the goal.

However, it cannot handle assumptions on typing of lambda abstraction well. It fails for the following example.

```
lemma
  assumes [qbs]: (λx. f x c) ∈ X ⇒Q Y
  shows (λx z. f x c) ∈ X ⇒Q Z ⇒Q Y
```

Implementation Note. We have implemented the qbs prover using raw ML code. There are some points to be noted.

- The following theorem corresponds to the rule APP_2 in Isabelle/HOL.
  ```
  lemma
    assumes f ∈ X ⇒Q Y ⇒Q Z and g ∈ X ⇒Q Y
    shows (λx. f x (g x)) ∈ X ⇒Q Z
  ```
  When applying the rule APP_2, we need to instantiate f and g in the lemma so that higher-order unification achieves an intended unification.
- When applying the rule CURRY, we should check by pattern matching that the goal is a lambda abstraction. Otherwise, it may overlap with APP_2 by eta-expanding $e_1\ e_2$ when the term has a function type.

We expect that this *typing* algorithm works in a similar situation where we want to restrict function spaces and constants in Isabelle/HOL. In our situation, function spaces are restricted to the set of morphisms.

3.3 The s-Finite Measure Monad

The s-finite measure monad on quasi-Borel spaces was introduced by Ścibior et al. [22] as the σ-finite measure monad. Then, it was reformulated as a submonad of the continuation monad $[0,\infty]^{[0,\infty]^{(-)}}$ by Vákár et al. [27, 28]. The details of the definition vary among these previous studies[6], and we could not find detailed proofs of monad laws and commutativity in any of them. We thus recover the detailed proofs first, and then we formalize them. We choose the definition given in Yang's lecture slide [30], because it is suitable for formalization in Isabelle/HOL. Its definition is quite similar to the probability monad introduced by Heunen et al. [10]. The probability monad is derived from the monad laws and the commutativity of the Giry monad, while the s-finite measure monad is derived from the properties of s-finite kernels and $\gg\!=_k$.

First, we define measures on quasi-Borel spaces to treat infinite measures such as the Lebesgue measure. Intuitively, a measure is a pair consisting of an s-finite measure μ on $\mathbb{R}$ and a random variable $\alpha \in qbs\text{-}Mx\ X$. We also introduce the equivalence relation $\sim$ of measures on quasi-Borel spaces defined by relating pairs with equal image measures. In our implementation, we use a triple (X,α,μ) rather than a pair (α,μ) because X cannot be inferred from α in simple type system.

[6] Thanks to the Borel-isomorphism theorem and the fact that s-finite measures can be rewritten as pushforward of σ-finite measures, those definitions are essentially equivalent.

▶ **Definition 4.** *A measure on a quasi-Borel space X is an equivalence class $[\![X, \alpha, \mu]\!]_{\mathsf{sfin}}$ where $\alpha \in$ qbs-Mx X, μ is a s-finite measure, and sets $\mu =$ sets $\mathbb{R}$. The equivalence relation is defined by $(X, \alpha, \mu) \sim (Y, \beta, \nu) \iff X = Y \wedge$ distr $(L\ X)\ \mu\ \alpha =$ distr $(L\ Y)\ \nu\ \beta$.*

We call a measure on a quasi-Borel space a *qbs-measure* in order to distinguish it from measures in measure theory. Using **quotient-type** command [14], we define the type *'a qbs-measure* which denotes the type of qbs-measures.

Any qbs-measure can be converted to an s-finite measure by the following function.

$$\begin{aligned} \textit{qbs-l} &:: \textit{'a qbs-measure} \Rightarrow \textit{'a measure} \\ \textit{qbs-l}\ [\![X, \alpha, \mu]\!]_{\mathsf{sfin}} &= \textit{distr}\ (L\ X)\ \mu\ \alpha \end{aligned}$$

The function *qbs-l* is injective by its definition (recall the definition of qbs-measures).

Next, we construct the s-finite measure monad.

▶ **Lemma 5.** *The quasi-Borel space of qbs-measures on X has the following structure.*

$$\begin{aligned} \textit{monadM-qbs} &:: \textit{'a quasi-borel} \Rightarrow \textit{'a qbs-measure quasi-borel} \\ \textit{qbs-space}\ (\textit{monadM-qbs}\ X) &= \{s.\ s \textit{ is a qbs-measure on } X\} \\ \textit{qbs-Mx}\ (\textit{monadM-qbs}\ X) &= \{\lambda r.\ [\![X, \alpha, k\ r]\!]_{\mathsf{sfin}} \mid \alpha\ k.\ \alpha \in \textit{qbs-Mx}\ X \wedge \textit{s-finite-kernel}\ \mathbb{R}\ \mathbb{R}\ k\} \end{aligned}$$

Notice that we use the s-finite kernel in the equation of *qbs-Mx* (*monadM-qbs* X). The proof of being quasi-Borel spaces is almost the same as the one of the probability monad. In the proof, we use that $\mathbb{N} \otimes_M \mathbb{R}$ is standard Borel. It is shown by the facts that $\mathbb{N}$ and $\mathbb{R}$ are standard Borel spaces, and the product measurable space of standard Borel spaces is again a standard Borel space.

The return (unit) operator and bind operator are defined as follows.

$$\begin{aligned} \textit{return}_Q &:: \textit{'a quasi-borel} \Rightarrow \textit{'a} \Rightarrow \textit{'a qbs-measure} \\ \textit{return}_Q\ X\ x &= [\![X, \lambda r.\ x, \nu]\!]_{\mathsf{sfin}} \\ \gg\!= &:: \textit{'a qbs-measure} \Rightarrow (\textit{'a} \Rightarrow \textit{'b qbs-measure}) \Rightarrow \textit{'b qbs-measure} \\ [\![X, \alpha, \mu]\!]_{\mathsf{sfin}} \gg\!= f &= [\![Y, \beta, \mu \gg\!=_k k]\!]_{\mathsf{sfin}} \end{aligned}$$

In the above definition,

- ν is an arbitrary probability measure on $\mathbb{R}$,
- $\beta \in$ *qbs-Mx* Y and *s-finite-kernel* $\mathbb{R}\ \mathbb{R}\ k$,
- $f \circ \alpha = (\lambda r.\ [\![X, \beta, k\ r]\!]_{\mathsf{sfin}})$.

Such β and k always exist since $f \circ \alpha \in$ *qbs-Mx* (*monadM-qbs* X).

The return operator and bind operator are defined in Isabelle/HOL as follows.

definition *return$_Q$* :: *'a quasi-borel* ⇒ *'a* ⇒ *'a qbs-measure* **where**
return$_Q$ X x ≡ [\![*X*, λr. *x*, *SOME μ. real-distribution μ*]\!]$_{\mathsf{sfin}}$

definition *bind-qbs* :: [*'a qbs-measure*, *'a* ⇒ *'b qbs-measure*] ⇒ *'b qbs-measure* **where**
bind-qbs s f ≡ (*let*
(*X*, *α*, *μ*) = *rep-qbs-measure s*;
Y = *qbs-space-of* (*f* (*α undefined*));
(*β*, *k*) = (*SOME* (*β*, *k*). *f* ∘ *α* = (λr. [\![*Y*, *β*, *k r*]\!]$_{sfin}$) ∧ *β* ∈ *qbs-Mx Y* ∧ *s-finite-kernel* ℝ ℝ *k*)
in [\![*Y*, *β*, *μ* ≫=$_k$ *k*]\!]$_{\mathsf{sfin}}$)

Here, *SOME x. P x* denotes some x satisfying P (Hilbert's ε), *real-distribution* μ means that μ is a probability measure on $\mathbb{R}$, *rep-qbs-measure* s returns a representative of the qbs-measure s, and *qbs-space-of* s returns the underlying space of s.

▶ **Theorem 6.** *The triple* $(\mathit{monadM\text{-}qbs}, \mathit{return}_Q, \gg\!=)$ *forms a commutative strong monad.*

The monad inherits properties of s-finite kernels and $\gg\!=_k$ which we have shown in Section 2.2. Proof of the laws for commutative strong monad is similar to the one of the probability monad. In the proof, we use that $\mathbb{R} \otimes_M \mathbb{R}$ is standard Borel.

The Probability Monad

We obtain the probability monad on quasi-Borel spaces by restricting *monadM-qbs X* as follows.

definition *monadP-qbs X* ≡ *sub-qbs* (*monadM-qbs X*) {*s*. *prob-space* (*qbs-l s*)}

The *sub-qbs X A* returns the sub space of a quasi-Borel space.

$$\begin{aligned} \mathit{qbs\text{-}space}\ (\mathit{sub\text{-}qbs}\ X\ A) &= \mathit{qbs\text{-}space}\ X \cap A \\ \mathit{qbs\text{-}Mx}\ (\mathit{sub\text{-}qbs}\ X\ A) &= \{\alpha.\ \alpha \in \mathit{qbs\text{-}Mx}\ X \wedge (\forall r.\ \alpha\ r \in A)\} \end{aligned}$$

The triple $(\mathit{monadP\text{-}qbs}, \mathit{return}_Q, \gg\!=)$ also forms a commutative strong monad. This monad has the exactly same structure with the probability monad in our previous work.

Integration

Integration with qbs-measure is defined through the Lebesgue integration. For $f \in X \to_Q \mathbb{R}$ and $s \in \mathit{qbs\text{-}space}\ (\mathit{monadM\text{-}qbs}\ X)$, the integration $(\int_Q x.\ f\ x\ \partial s)$ is defined by $(\int_Q x.\ f\ x\ \partial s) = (\int x.\ f\ x\ \partial(\mathit{qbs\text{-}l}\ s))$. The notions of *integrable* and *almost everywhere* are defined in a similar way.

For an s-finite measure M on a standard Borel space, integration w.r.t. M in measure theory is represented as integration in quasi-Borel theory. We define the inverse function of *qbs-l*, by $\mathit{qbs\text{-}l}^{-1}\ M = [\![M, \mathit{from\text{-}real}_M, \mathit{distr}\ \mathbb{R}\ M\ \mathit{to\text{-}real}_M]\!]_{\mathsf{sfin}}$. Using these conversions *qbs-l* and $\mathit{qbs\text{-}l}^{-1}$, we obtain $(\int x.\ f\ x\ \partial M) = (\int_Q x.\ f\ x\ \partial(\mathit{qbs\text{-}l}^{-1}\ M))$. We thus may regard an s-finite measure M on a standard Borel space as a qbs-measure $\mathit{qbs\text{-}l}^{-1}\ M$ on $R\ M$, and regard a qbs-measure s as an s-finite measure *qbs-l s*.

For instance, we can represent the Lebesgue measure as a qbs measure. Recall that the Lebesgue measure is σ-finite, hence it is s-finite.

definition $\mathit{lborel}_Q \equiv \mathit{qbs\text{-}l}^{-1}\ \mathit{lborel}$
lemma $\mathit{qbs\text{-}l}\ \mathit{lborel}_Q = \mathit{lborel}$
corollary $(\int_Q x.\ f\ x\ \partial \mathit{lborel}_Q) = (\int x.\ f\ x\ \partial \mathit{lborel})$

4 Probabilistic Programs

Let us implement a probabilistic programming language supporting higher-order functions, sampling, and conditioning with quasi-Borel spaces and the s-finite measure monad. We discuss three examples in this section.

4.1 The Language

We use Isabelle/HOL terms as probabilistic programs. The language design is inspired by `HPProg` introduced by Sato et al. [21]. We first briefly review the type system and semantics of `HPProg`. The language `HPProg` is a higher-order functional probabilistic programming

language based on simply-typed lambda calculus along with the monadic type for distributions. Types are defined inductively as follows.

$$T ::= \mathsf{nat} \mid \mathsf{bool} \mid \mathsf{real} \mid \mathsf{preal} \mid \mathsf{list}[T] \mid T \times T \mid T \Rightarrow T \mid M[T].$$

The type preal denotes the type of $[0, \infty]$ and $M[T]$ denotes the type of distributions (measures) on T. In the semantics, types are interpreted as quasi-Borel spaces.

$$[\![\mathsf{nat}]\!] = \mathbb{N}, \quad [\![\mathsf{bool}]\!] = \mathbb{B}, \quad [\![\mathsf{real}]\!] = \mathbb{R}, \quad [\![\mathsf{preal}]\!] = \mathbb{R}_{\geq 0}, \quad [\![\mathsf{list}[T]]\!] = \textit{list-qbs}\ [\![T]\!]$$
$$[\![T_1 \times T_2]\!] = [\![T_1]\!] \otimes_Q [\![T_2]\!], \quad [\![T_1 \Rightarrow T_2]\!] = [\![T_1]\!] \Rightarrow_Q [\![T_2]\!], \quad [\![M[T]]\!] = \textit{monadM-qbs}\ [\![T]\!].$$

A typing judgment $\Gamma \vdash t : T$ is interpreted as "$[\![t]\!]$ is a morphism from $[\![\Gamma]\!]$ to $[\![T]\!]$". A typing judgment $\vdash t : T$ is interpreted as "$[\![t]\!] \in [\![T]\!]$".

According to this semantics, an Isabelle/HOL term is interpreted as a probabilistic program. We say that an Isabelle/HOL term t is a program of type T if $t \in \textit{qbs-space}\ T$. Many standard constants in Isabelle/HOL are programs.

$$(+) \in \mathbb{R} \Rightarrow_Q \mathbb{R} \Rightarrow_Q \mathbb{R}\ , \qquad (-) \in \mathbb{R} \Rightarrow_Q \mathbb{R} \Rightarrow_Q \mathbb{R}\ , \qquad (*) \in \mathbb{R} \Rightarrow_Q \mathbb{R} \Rightarrow_Q \mathbb{R}$$

$$[] \in \textit{list-qbs}\ X\ , \qquad \textit{Cons} \in X \Rightarrow_Q \textit{list-qbs}\ X \Rightarrow_Q \textit{list-qbs}\ X$$

$$\textit{rec-list} \in Y \Rightarrow_Q (X \Rightarrow_Q \textit{list-qbs}\ X \Rightarrow_Q Y \Rightarrow_Q Y) \Rightarrow_Q \textit{list-qbs}\ X \Rightarrow_Q Y$$

Operators for distributions are also programs.

$$\textit{return}_Q\ X \in X \Rightarrow_Q \textit{monadM-qbs}\ X$$

$$(\gg\!=) \in \textit{monadM-qbs}\ X \Rightarrow_Q (X \Rightarrow_Q \textit{monadM-qbs}\ Y) \Rightarrow_Q \textit{monadM-qbs}\ Y$$

$$(\otimes_{Qmes}) \in \textit{monadM-qbs}\ X \Rightarrow_Q \textit{monadM-qbs}\ Y \Rightarrow_Q \textit{monadM-qbs}\ (X \otimes_Q Y)$$

$$\textit{Uniform} \in \mathbb{R} \Rightarrow_Q \mathbb{R} \Rightarrow_Q \textit{monadM-qbs}\ \mathbb{R}\ , \qquad \textit{Gauss} \in \mathbb{R} \Rightarrow_Q \mathbb{R} \Rightarrow_Q \textit{monadM-qbs}\ \mathbb{R}$$

The program $(\otimes_{Qmes})$ is defined for $p \in \textit{monadM-qbs}\ X$ and $q \in \textit{monadM-qbs}\ Y$ by

$$p \otimes_{Qmes} q = p \gg\!= (\lambda x.\ q \gg\!= (\lambda y.\ \textit{return}_Q\ (X \otimes_Q Y)\ (x,y)))$$

which denotes their product distribution[7]. The program *Uniform a b* denotes the continuous uniform distribution between a and b. The program *Gauss* μ σ denotes the Gaussian distribution with the average μ and the standard deviation σ.

Let us compare the language implementation with our previous work [11]. Our previous language implementation lift Isabelle/HOL constants to constant functions in order to accommodate contexts. For instance, a real number r is described as $(\lambda env.\ r) \in \Gamma \Rightarrow_Q \mathbb{R}$. The variables are projections from contexts and thus programs are written in de Bruijn index, that is, variables are identified by natural numbers. Although using de Bruijn index makes it almost straightforward to write type checking proofs, it causes low readability and cumbersome renaming of variables during proofs. By contrast, our new implementation uses Isabelle/HOL terms directly. This approach is similar to CryptHOL by Basin et al. [5, 16],

[7] Because the s-finite measure monad is commutative, we have

$$p \otimes_{Qmes} q = q \gg\!= (\lambda y.\ p \gg\!= (\lambda x.\ \textit{return}_Q\ (X \otimes_Q Y)\ (x,y))).$$

where they have embedded a functional probabilistic programming language for discrete distributions in order to verify cryptographic algorithms. The benefit is that it is much more readable and easier to work with terms when writing programs and reasoning about programs. Our qbs prover presented in Section 3.2 almost automates type checking even though programs are written as Isabelle/HOL terms. As we will demonstrate in later sections, program verification can be done directly in Isabelle/HOL.

The *query* Command

We define the *query* command which enables one to write conditional distributions. The *query* has the following type:

$$\mathit{query} \in \mathit{monadM\text{-}qbs}\ X \Rightarrow_Q (X \Rightarrow_Q \mathbb{R}_{\geq 0}) \Rightarrow_Q \mathit{monadM\text{-}qbs}\ X\ .$$

For a prior distribution s and a likelihood f, $\mathit{query}\ s\ f$ returns the posterior distribution. The *query* command is defined through two operators: $\mathit{density}_Q$ (`scale` in `HPProg`) and $\mathit{normalize}_Q$.

definition $\mathit{query} \equiv (\lambda s\ f.\ \mathit{normalize\text{-}qbs}\ (\mathit{density\text{-}qbs}\ s\ f))$

The operator $\mathit{density}_Q$ takes a qbs-measure s and a non-negative function f and rescales s with the density function f. The $\mathit{density}_Q$ satisfies following properties.

$$\mathit{density}_Q \in \mathit{monadM\text{-}qbs}\ X \Rightarrow_Q (X \Rightarrow_Q \mathbb{R}_{\geq 0}) \Rightarrow_Q \mathit{monadM\text{-}qbs}\ X$$

$$(\int_Q x.\ g\ x\ \partial(\mathit{density}_Q\ s\ f)) = (\int_Q x.\ f\ x * g\ x\ \partial s)$$

The operator $\mathit{normalize}_Q$ normalizes a qbs-measure s on X. If $\mathit{qbs\text{-}l}\ s\ X = 0$ or ∞, then $\mathit{normalize}_Q\ s$ returns the null-measure on X.

$$\mathit{normalize}_Q \in \mathit{monadM\text{-}qbs}\ X \Rightarrow_Q \mathit{monadM\text{-}qbs}\ X$$

The *condition* Command

We introduce the *condition* command, which produces a conditional distribution with a predicate. The *condition* command has the following type and defined using the *query* command and the indicator function as follows.

$$\mathit{condition} \in \mathit{monadM\text{-}qbs}\ X \Rightarrow_Q (X \Rightarrow_Q \mathbb{B}) \Rightarrow_Q \mathit{monadM\text{-}qbs}\ X$$

definition $\mathit{condition}\ s\ P \equiv \mathit{query}\ s\ (\lambda x.\ \mathit{if}\ P\ x\ \mathit{then}\ 1\ \mathit{else}\ 0)$

4.2 Example: What time is it?

We formalize the example from Staton [25, Section 2.2], which we have shown in introduction. This example uses two language features: higher-order function and conditioning. Let us consider the following situation.

- We want to know what time it is.
- We know the rate of bikes per hour, which depends on time.
- We observed a 1 minute gap between two bikes.
- What time is it?

We define the program *whattime* as follows.

```
definition whattime :: (real ⇒ real) ⇒ real qbs-measure where
whattime ≡ (λf. do {
                let T = Uniform 0 24 in
                query T (λt. let r = f t in
                          exponential-density r (1 / 60))
              })
```

The program *whattime* receives a function f which determines the rate of bikes per hour. Then the program returns the posterior after observing a 1 minute gap between two bikes. The return value $f\ t$ denotes the rate of bikes per hour at the time t, and the time gap between two bikes follows the exponential distribution $\mathrm{Exp}(f\ t)$. Thus, the likelihood is calculated using the density function *exponential-density* of the exponential distribution. We can prove *whattime* is a program just by unfolding the definition thanks to our qbs prover presented in Section 3.2.

```
lemma whattime ∈ (ℝ ⇒Q ℝ) ⇒Q monadM-qbs ℝ
  by(simp add: whattime-def)
```

As explained by Staton, the posterior is computed as follows.

```
lemma
  assumes f ∈ ℝ ⇒Q ℝ and U ∈ sets ℝ and ⋀t. f t ≥ 0
  defines N ≡ (∫ t∈{0<..<24}. (f t * exp (− 1/ 60 * f t)) ∂lborel)
  assumes N ≠ 0 and N ≠ ∞
  shows 𝒫(t in whattime f. t ∈ U) = (∫ t∈{0<..<24}∩U. (f t * exp (− 1/ 60 * f t)) ∂lborel) / N
```

4.3 Example: Two Dice

As a second example, we formalize the example from Sampson [20, Section 2.3]. This example uses two language features: sampling and conditioning. We consider the following problem.

- We role two dice.
- We observe at least one die is 4.
- What is the sum of the two dice?

We describe the distribution of the sum of the two dice as follows.

```
definition two-dice :: nat qbs-measure where
two-dice ≡ do {
             let die1 = die;
             let die2 = die;
             let twodice = die1 ⊗Qmes die2;
             (x,y) ← condition twodice (λ(x,y). x = 4 ∨ y = 4);
             returnQ ℕ (x + y)
            }
```

Here, *die* ∈ *monadM-qbs* ℕ denotes the distribution of rolling a fair die. The program picks a sample from the conditional distribution, then returns the sum of dice. The program *two-dice* has the following type.

```
lemma two-dice ∈ monadM-qbs ℕ
  by(simp add: two-dice-def)
```

We show the probabilities where the program takes each possible value.

```
lemma
 𝒫(x in two-dice. x = 5)  = 2 / 11   𝒫(x in two-dice. x = 6)  = 2 / 11
 𝒫(x in two-dice. x = 7)  = 2 / 11   𝒫(x in two-dice. x = 8)  = 1 / 11
 𝒫(x in two-dice. x = 9)  = 2 / 11   𝒫(x in two-dice. x = 10) = 2 / 11
```

4.4 Example: Gaussian Mean Learning

As a final example, let us formalize the example from Sato et. al. [21, Section 8.2]. We implement the Gaussian Mean Learning algorithm and prove two properties: convergence and stability under change of priors. In a common situation in statistical modeling or machine learning, we try to infer unknown parameters from a sample list. For instance, let us consider the following situation.

- We want to know the mean of a Gaussian distribution with a known standard deviation.
- We have a sample sequence from the Gaussian distribution.
- What is the posterior of the mean?

The following algorithm does Bayesian learning of the mean of a Gaussian distribution with a known standard deviation σ from a sample list.

```
primrec GaussLearn' :: [real, real qbs-measure, real list] ⇒ real qbs-measure where
  GaussLearn' - p [] = p
| GaussLearn' σ p (y#ls) = query (GaussLearn' σ p ls) (normal-density y σ)
```

Here, *normal-density* $y\ \sigma$ is the density function of the Gaussian distribution *Gauss* $y\ \sigma$ with mean y.

The program *GaussLearn'* receives a standard deviation σ, a prior p and a sample list L. In each iteration, the program picks a sample from L, then updates the prior. Our qbs prover can show that *GaussLearn'* is a program because *GaussLearn'* is a primitive recursive function[8].

```
lemma GaussLearn' ∈ ℝ ⇒Q monadM-qbs ℝ ⇒Q list-qbs ℝ ⇒Q monadM-qbs ℝ
  by (simp add: GaussLearn'-def)
```

From now on, we fix $\sigma > 0$ and abbreviate *GaussLearn'* σ as *GaussLearn*.

The first property, convergence, is described as follows.

```
lemma
  assumes ξ > 0 and n = length L
  shows GaussLearn (Gauss δ ξ) L =
        Gauss ((Total L * ξ² + δ * σ²) / (n * ξ² + σ²)) (sqrt ((ξ² * σ²) / (n * ξ² + σ²)))
```

Here, the program *Total* $\in$ *list-qbs* $\mathbb{R} \Rightarrow_Q \mathbb{R}$ sums up all elements of a list. The above statement says that if the prior of the mean is *Gauss* $\delta\ \xi$, then the posterior is also a Gaussian distribution. Furthermore, its mean and standard deviation are close to the average of the samples and 0, respectively, when n is sufficiently large.

Next, let us see the second property, stability under change of priors. We show that if we run *GaussLearn* from two different priors and give a large sample list whose average is bounded, then the resulting posteriors will be close. We measure the difference between distributions by the Kullback-Leiber (KL) divergence. The KL divergence is provided as *KL-divergence* in the standard Isabelle/HOL library. If p and q are probability distributions on $\mathbb{R}$ which have positive density functions f and g, respectively, then we have the following well-known form of KL divergence:

```
KL-divergence (exp 1) p q = (∫ x. g x * ln (g x / f x) ∂lborel)
```

The second property is stated as follows.

[8] Internally, the **primrec** command defines a primitive recursive function using recursors such as *rec-nat* and *rec-list*.

lemma *GaussLearn-KL-divergence*:
fixes *a b c d ε K* :: *real*
assumes $\varepsilon > 0$ **and** $b > 0$ **and** $d > 0$
shows $\exists N.\ \forall L.\ length\ L > N \longrightarrow |Total\ L\ /\ length\ L| < K \longrightarrow$
KL-divergence (*exp 1*) (*GaussLearn* (*Gauss a b*) *L*) (*GaussLearn* (*Gauss c d*) *L*) $< \varepsilon$

Intuitively, the above property says that if we run *GaussLearn* with two different Gauss distributions, then we can make the distance of posteriors as close as we want with a large sample list whose average is bounded.

5 Conclusion

We have implemented s-finite kernels, the Borel isomorphism theorem, proof automation for quasi-Borel spaces, and the s-finite measure monad. Using our formalization, we can directly treat probabilistic programs presented in previous works and prove their properties. Our work enables us to denote probabilistic programs supporting all of higher-order functions, samplings, and conditioning, while our previous work [11] does not support conditioning and the work by Affeldt et al. [1] does not support higher-order functions.

There are several researches related to probabilistic programs with proof assistants. Eberl et al. have constructed an executable first-order functional probabilistic programming language which computes density functions in Isabelle, and proved its correctness [7]. Basin et al. have implemented CryptHOL for rigorous game-based proofs in Isabelle/HOL [5, 16]. They shallowly embedded a functional programming language, and verified cryptographic algorithms. For machine learning verification, Bagnall and Stewart have embedded MLCert in Coq [4], and Tristan et al. have implemented a simplified measure-theoretic semantics of probabilistic programs based on the reparameterizations to the uniform distibution on the unit interval and partially automated verification in Lean [26]. Zhang and Amin formalized a formal semantics for a core probabilistic programming language and proved that logical relatedness implies contextual equivalence using axiomatized measure theory in Coq [31].

There are future extensions of our work. In our formalization, we have manually constructed quasi-Borel spaces on basic data types defined inductively, such as lists and options. Then we show that constructors and recursors (primitive recursive function operator) are morphisms. We expect that we can automate this process. Furthermore, we think that it is also possible to show that general wellfounded recursive functions, which may not be primitive recursive, are morphisms.

References

1 Reynald Affeldt, Cyril Cohen, and Ayumu Saito. Semantics of probabilistic programs using s-finite kernels in Coq. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2023, pages 3–16, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3573105.3575691.

2 Robert J. Aumann. Borel structures for function spaces. *Illinois Journal of Mathematics*, 5(4):614–630, 1961. doi:10.1215/ijm/1255631584.

3 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.

4 Alexander Bagnall and Gordon Stewart. Certifying the true error: Machine learning in Coq with verified generalization guarantees. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33:2662–2669, 2019. doi:10.1609/aaai.v33i01.33012662.

5 David A. Basin, Andreas Lochbihler, and S. Reza Sefidgar, Sefidgar. CryptHOL: Game-based proofs in higher-order logic. *Journal of Cryptology*, 33:494–566, 2020.

6 Marek Biskup. Lecture note of math245b in UCLA. `https://web.archive.org/web/20210506130459/https://www.math.ucla.edu/~biskup/245b.1.20w/`, 2020. Accessed: January 17. 2023.

7 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`.

8 Michèle Giry. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis*, pages 68–85, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg. `doi:10.1007/BFb0092872`.

9 Noah D. Goodman et al. Church: a language for generative models. In *UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence*, pages 220–229. AUAI Press, 2008.

10 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`.

11 Michikazu Hirata, Yasuhiko Minamide, and Tetsuya Sato. Program logic for higher-order probabilistic programs in Isabelle/HOL. In Michael Hanus and Atsushi Igarashi, editors, *Functional and Logic Programming*, pages 57–74, Cham, 2022. Springer International Publishing.

12 Johannes Hölzl. Markov processes in Isabelle/HOL. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 100–111. Association for Computing Machinery, 2017. `doi:10.1145/3018610.3018628`.

13 Johannes Hölzl, Armin Heller, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk. Three chapters of measure theory in Isabelle/HOL. In *Interactive Theorem Proving*, ITP 2011, pages 135–151. Springer Berlin Heidelberg, 2011.

14 Cezary Kaliszyk and Christian Urban. Quotients revisited for Isabelle/HOL. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC, pages 1639–1644, New York, NY, USA, 2011. Association for Computing Machinery. `doi:10.1145/1982185.1982529`.

15 Ondřej Kunčar and Andrei Popescu. From types to sets by local type definitions in higher-order logic. In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving*, pages 200–218, Cham, 2016. Springer International Publishing.

16 Andreas Lochbihler. Probabilistic functions and cryptographic oracles in higher order logic. In *Programming Languages and Systems*, pages 503–531, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. `doi:10.1007/978-3-662-49498-1_20`.

17 Mihails Milehins. An extension of the framework types-to-sets for Isabelle/HOL. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, pages 180–196, New York, NY, USA, 2022. Association for Computing Machinery. `doi:10.1145/3497775.3503674`.

18 Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

19 Andrei Popescu and Dmitriy Traytel. Admissible types-to-pers relativization in higher-order logic. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. `doi:10.1145/3571235`.

20 Adrian Sampson. Probabilistic programming. `http://adriansampson.net/doc/ppl.html`. Accessed: January 25. 2023.

21 Tetsuya Sato, Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Justin Hsu. Formal verification of higher-order probabilistic programs: reasoning about approximation, convergence, bayesian inference, and optimization. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019. `doi:10.1145/3290351`.

22 Adam Ścibior, Ohad Kammar, Matthijs Vákár, Sam Staton, Hongseok Yang, Yufei Cai, Klaus Ostermann, Sean K. Moss, Chris Heunen, and Zoubin Ghahramani. Denotational

validation of higher-order bayesian inference. *Proc. ACM Program. Lang.*, 2(POPL), 2017. doi:10.1145/3158148.

23 Shashi Mohan Srivastava. *A Course on Borel Sets*. Springer, 1998. doi:10.1007/b98956.

24 Sam Staton. Commutative semantics for probabilistic programming. In Hongseok Yang, editor, *Programming Languages and Systems*, pages 855–879, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

25 Sam Staton. *Probabilistic Programs as Measures*, pages 43–74. Cambridge University Press, 2020. doi:10.1017/9781108770750.003.

26 Jean-Baptiste Tristan, Joseph Tassarotti, Koundinya Vajjha, Michael L. Wick, and Anindya Banerjee. Verification of ML systems via reparameterization, 2020. arXiv:2007.06776.

27 Matthijs Vákár, Ohad Kammar, and Sam Staton. A domain theory for statistical probabilistic programming. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290349.

28 Matthijs Vákár and Luke Ong. On s-finite measures and kernels, 2018. doi:10.48550/ARXIV.1810.01837.

29 Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, pages 1024–1032, 2014.

30 Hongseok Yang. Semantics of higher-order probabilistic programs with continuous distributions. https://alfa.di.uminho.pt/~nevrenato/probprogschool_slides/Hongseok.pdf. Accessed: February 8. 2023.

31 Yizhou Zhang and Nada Amin. Reasoning about "reasoning about reasoning": Semantics and contextual equivalence for probabilistic programs with nested queries and recursion. *Proc. ACM Program. Lang.*, 6(POPL), January 2022. doi:10.1145/3498677.

MizAR 60 for Mizar 50

Jan Jakubův
Czech Technical University in Prague, Czech Republic

Karel Chvalovský
Czech Technical University in Prague, Czech Republic

Zarathustra Goertzel
Czech Technical University in Prague, Czech Republic

Cezary Kaliszyk
Universität Innsbruck, Austria
INDRC, Prague, Czech Republic

Mirek Olšák
Institut des Hautes Études Scientifiques, Paris, France

Bartosz Piotrowski
Czech Technical University in Prague, Czech Republic

Stephan Schulz
DHBW Stuttgart, Germany

Martin Suda
Czech Technical University in Prague, Czech Republic

Josef Urban
Czech Technical University in Prague, Czech Republic

Abstract

As a present to Mizar on its 50th anniversary, we develop an AI/TP system that automatically proves about 60 % of the Mizar theorems in the hammer setting. We also automatically prove 75 % of the Mizar theorems when the automated provers are helped by using only the premises used in the human-written Mizar proofs. We describe the methods and large-scale experiments leading to these results. This includes in particular the E and Vampire provers, their ENIGMA and Deepire learning modifications, a number of learning-based premise selection methods, and the incremental loop that interleaves growing a corpus of millions of ATP proofs with training increasingly strong AI/TP systems on them. We also present a selection of Mizar problems that were proved automatically.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases Mizar, ENIGMA, Automated Reasoning, Machine Learning

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.19

Related Version *Extended Version*: `https://doi.org/10.48550/arXiv.2303.06686`

Supplementary Material *Software*: `https://github.com/ai4reason/ATP_Proofs`

Funding The funding for the multi-year development of the methods and for the experiments was partially provided by the ERC Consolidator grant *AI4REASON* no. 649043 (KC, ZG, JJ, BP, MS and JU), the European Regional Development Fund under the Czech project AI&Reasoning no. CZ.02.1.01/0.0/0.0/15_003/0000466 (KC, ZG, JJ, MO, JU), the ERC Starting Grant *SMART* no. 714034 (JJ, CK, MO), the Czech Science Foundation project 20-06390Y and project RICAIP no. 857306 under the EU-H2020 programme (MS), ERC-CZ project POSTMAN no. LL1902 (KC, JJ, BP), Amazon Research Awards (JU), the EU ICT-48 2020 project TAILOR no. 952215 (JU), and the grant 2018/29/N/ST6/02903 of National Science Center, Poland (BP).

Acknowledgements The development of ENIGMA, premise selection and other methods used here, as well as the large-scale experiments, benefited from many informal discussions which involved (at least) Lasse Blaauwbroek, Chad Brown, Thibault Gauthier, Mikoláš Janota, Jelle Piepenbrock, Stanisław Purgał, Bob Veroff, and Jiří Vyskočil.

© Jan Jakubův, Karel Chvalovský, Zarathustra Goertzel, Cezary Kaliszyk, Mirek Olšák, Bartosz Piotrowski, Stephan Schulz, Martin Suda, and Josef Urban;
licensed under Creative Commons License CC-BY 4.0

1 Introduction: Mizar, MML, Hammers and AITP

In recent years, methods that combine machine learning (ML), artificial intelligence (AI) and automated theorem proving (ATP) [44] have been considerably developed, primarily targeting large libraries of formal mathematics developed by the ITP community. This ranges from *premise selection* methods [2] and *hammer* [7] systems to developing and training learning-based *internal guidance* of ATP systems such as E [47, 49] and Vampire [37] on the thousands to millions of problems extracted from the ITP libraries. Such large ITP corpora have further enabled research topics such as *automated strategy invention* [57] and *tactical guidance* [15], learning-based *conjecturing* [58], *autoformalization* [34, 61], and development of metasystems that combine learning and reasoning in various feedback loops [59].

Starting with the March 2003 release[1] of the MPTP system [54] and the first ML/TP and hammer experiments over it [55], the Mizar Mathematical Library [3, 4, 22] (MML) and its subsets have as of 2023 been used for twenty years for this research, making it perhaps the oldest and most researched AI/TP resource in the last two decades.

1.1 Contributions

The last large *Mizar40* evaluation [32] of the AI/TP methods over MML was done almost ten years ago, on the occasion of 40 years of Mizar. Since then, a number of strong methods have been developed in areas such as premise selection and internal guidance of ATPs. In this work, we therefore evaluate these methods in a way that can be compared to the Mizar40 evaluation, providing an overall picture of how far the field has moved. Our main results are:

1. Over 75 % of the Mizar toplevel lemmas can today be proved by AI/TP systems when the premises for the proof can be selected from the library either by a human or a machine. This should be compared to 56 % in Mizar40 achieved on the same version of the MML. Over 200 examples of the automatically obtained proofs are analyzed on our web page.[2]
2. 58.4 % of the Mizar toplevel lemmas can be proved today without any help from the users, i.e., in the large-theory (hammering) mode. This should be compared to about 40.6 % achieved on the same version of the MML in Mizar40. In both cases, this is done by a large portfolio of AI/TP methods which is limited to 420 s of CPU time.
3. Our strongest single AI/TP method alone now proves in 30 s 40 % of the lemmas in the hammering mode, i.e., reaching the same strength as the full 420 s portfolio in Mizar40.
4. Our strongest *single* AI/TP method now proves in 120 s 60 % of the toplevel lemmas in the human-premises (*bushy*) mode (Section 6.6), i.e., outperforming the union of *all* methods developed in Mizar40 (56 %).
5. We show that our strongest method transfers to a significantly newer version of the MML which contains a lot of new terminology and lemmas. In particular, on the new 13 370 theorems coming from the new 242 articles in MML version 1382, our strongest method outperforms standard E prover by 58.2 %, while this is only 56.1 % on the Mizar40 version of the library where we do the training and experiments. This is thanks to our development and use of *anonymous* [25] logic-aware ML methods that learn only from the structure of mathematical problems. This is unusual in today's machine learning which is dominated by large language models that typically struggle on new terminology.

[1] http://mizar.uwb.edu.pl/forum/archive/0303/msg00004.html
[2] https://github.com/ai4reason/ATP_Proofs

1.2 Overview of the Methods and Experiments

The central methods in this evaluation are internal guidance provided by the ENIGMA (and later also Deepire) system, and premise selection methods. We have also used several additional approaches such as many previously invented strategies and new methods for constructing their portfolios, efficient methods for large-scale training on millions of ATP proofs, methods that interleave multiple runs of ATPs with restarts on ML-based selection of the best inferred clauses (*leapfrogging*), and methods for minimizing the premises needed for the problems by decomposition into many ATP subproblems. These methods are described in Sections 3, 4, and 5, after introducing the MML in Section 2. Section 6 describes the large-scale evaluation and its final results, and Section 7 showcases the obtained proofs.

2 The Mizar Mathematical Library and the Mizar40 Corpus

Proof assistant systems are usually developed together with their respective proof libraries. This allows evaluating and showcasing the available functionality. In the case of Mizar [4], the developers have very early decided to focus on its library, the MML (Mizar Mathematical Library) [3]. This was done by establishing a dedicated library committee responsible for the evaluation of potential Mizar articles to be included, as well as for maintaining the library. As a result, the MML became one of the largest libraries of formalized mathematics today. It includes many results absent from those derived in other systems, such as lattices [5] and random-access Turing machines [36].

All the data gathered and evaluations performed in the paper (with the exception of version-transfer in Section 6.6) use the same Mizar library version as the previous large evaluation [32] and all subsequent evaluation papers. This allows us to rigorously compare the methods and evaluate the improvement. That version of the library, MML 1147, when exported to first-order logic using the MPTP export [56] corresponds to 57 897 theorems including the unnamed toplevel lemmas. For a rigorous evaluation in the hammering scenario, we will further split this dataset into several training and testing parts in Section 6.2.

3 ENIGMA: ATP Guidance and Related Technologies

ENIGMA [11, 18–21, 25, 27–29] stands for "*Efficient Learning Based Inference Guiding Machine*". It is the first learning-guided ATP that in 2019 achieved large improvements over state-of-the-art saturation ATPs [29], and the main ingredient of the work reported here. This section summarizes previously published research on ENIGMA and also the related methods that were used to undertake the large-scale experiments done here (Section 6).

3.1 Saturation Theorem Proving Meets Machine Learning

Saturation Provers. State-of-the-art automated theorem provers, like E Prover [45] and Vampire [37], perform the search for a contradiction, first translating the input first-order logic problem into a refutationally equivalent set of clauses. Then the prover operates the proof search using the *given clause algorithm*. In this algorithm, the proof state is split into two subsets, the set P of *processed clauses*, and the set U of unprocessed clauses. Clauses in U are ordered by a heuristic evaluation function. In each iteration of the main loop, the (heuristically) best clause in U is picked. This *given clause* g is then simplified with respect to all clauses in P. If it is not redundant, it is used in turn to simplify all clauses in P. After that, all generating inferences between g and the remaining clauses in P are performed. Both

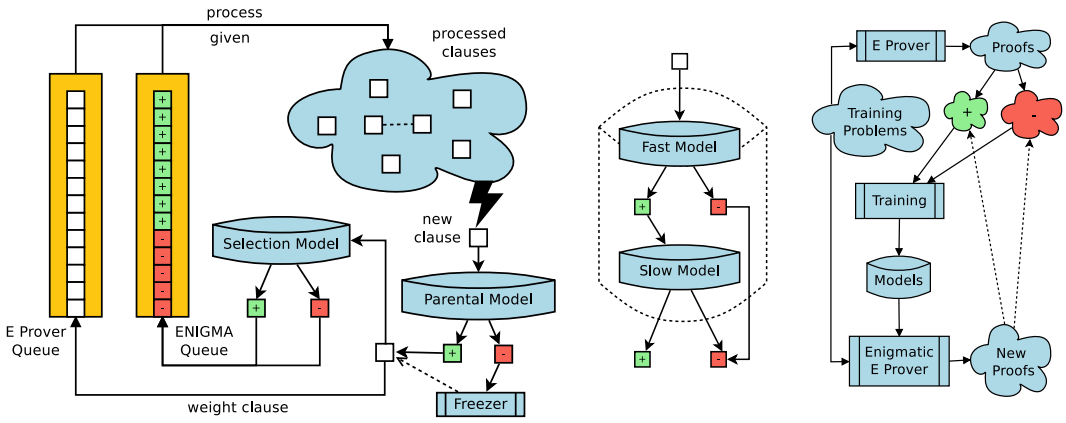


Figure 1 Schema of E Prover with ENIGMA (left), of a two-phase selection model (middle), and of the prove-learn feedback loop (right).

the newly generated clauses and the simplified clauses from P are then completely simplified with respect to P, heuristically evaluated, and added to U. This process continues until the empty clause emerges (or until the system runs out of resources).

Training Data. As of E 1.8 [48], E maintains an internal proof object [50] which allows it to inspect all proof clauses and designate all clauses that have been selected for processing and are part of the proof, as *positive training examples*. All clauses that have been selected for processing, but not contributed to the proof, are designated as *negative training examples*. Clauses that have not been processed at all are neither positive nor negative, reducing the total number of training examples to typically thousands of processed clauses, as opposed to millions of clauses generated. E allows the user to request the actual proof object, or to provide any combination of positive and negative training examples. Examples are provided in separate batches and are also annotated as positive or negative for easy processing.

ML-Based Selection. Selection of the right given clause is critical in E, and an obvious point for the use of machine learning (ML). The positive and negative examples are extracted from previous successful proof searches, and a machine learning model is trained to score the generated clauses or to classify them as useful (⊞) or useless (⊟). E Prover selects the given clause from a priority queue, where the unprocessed clauses are sorted by various heuristics. ENIGMA extends E Prover with an additional queue where clauses positively classified by the ML model are prioritized. The ENIGMA queue is used together with the standard E selection mechanisms, typically in a cooperative way where roughly half of the clauses are selected by ENIGMA. This approach proved to be the most efficient in practice.

Parental Guidance. Later ENIGMA [20] introduced learning-based *parental guidance*, which addresses the quadratic factor when doing *all* possible inferences among the processed clauses in classical saturation-based provers. Instead, an ML model is trained to prevent inferences between the parent clauses that are unlikely to meaningfully interact. When such an inference is recognized by the model as useless with a high degree of confidence, the child clause is not inserted into the set of unprocessed clauses U but its processing is postponed. To maintain completeness, the clause can not be directly discarded since the ML model might be mistaken. Instead, the clause is put into a "freezer" from which it can be retrieved in the case the

prover runs out of unprocessed clauses. As opposed to the above clause selection models, this method affects the standard E selection mechanism because the clause is not inserted into any queue. ENIGMA clause selection models and parental (*generation*) models can be successfully combined. This is schematically illustrated in Figure 1 (left).

Multi-Phase ENIGMA. ML-based multi-phase clause selection was introduced in [20] to deal with computationally expensive (*slow*) ML models, like graph neural networks (GNNs). In a *two-phase* selection model, a *faster* model is used for preliminary clause filtering, and only the clauses that pass are evaluated by the slower model. The fast model is expected to over-approximate on positive classes so that only clauses classified with high confidence as negatives are rejected. When parental guidance is added to the mix, this leads to a *three-phase* ENIGMA. This is schematically illustrated in Figure 1 (middle). *Aggressive forward subsumption* is an additional logic-complete pruning method based on efficient subsumption indexing in E [46]. We use it to eliminate many redundant generated clauses before calling more expensive ML methods (GNN) for clause evaluation. For the effect of such methods, see some of the 3-phase ENIGMA examples in Section 7.

Training. Strong ENIGMAs are typically developed in many prove-learn feedback loops [59] that proceed as follows. (1) The training data $\mathcal{T}$ are curated from (previous) successful proof searches. (2) A model $\mathcal{M}$ is trained on data $\mathcal{T}$ to distinguish positive from negative clauses. (3) The model $\mathcal{M}$ is run with the ATP (E), usually in *cooperation* with the strategy used to obtain the training data. Then we go to step (1) with the new data obtained in step (3). The loop, illustrated in Figure 1 (right), can be repeated as long as new problems are proved. We run this loop for several months in this work.

3.2 Gradient Boosted Decision Tree Classifiers and Features

ENIGMA supports classifiers based on Gradient Boosted Decision Trees (GBDTs). In particular, we experiment with XGBoost [8] and LightGBM [35]. Both frameworks are efficient and can handle large data well both in training and evaluation. For learning, we represent first-order clauses by numeric *feature vectors*. A decision tree is a binary tree with nodes labeled by conditions on the values of the feature vectors. Given a clause, the tree is navigated to the leaf where the clause evaluation is stored. Both frameworks work with a sequence (*ensemble*) of several trees, constructed in a progressive way (*boosting*). The frameworks differ in the underlying algorithm for the construction of decision trees. XGBoost constructs trees level-wise, while LightGBM leaf-wise. This implies that XGBoost trees are well-balanced. On the other hand, LightGBM can produce much deeper trees, and the tree depth limit is indeed an important learning meta-parameter that can be optimized.

ENIGMA extracts various syntactic information from a first-order clause and stores them in the feature vector of the clause. Given a finite set of features, each feature is assigned an index in the feature vector, and the corresponding feature value is stored at this index. For example, a typical clause feature is the clause length. ENIGMA supports the following. **Vertical Features** are constructed by traversing the clause syntax tree and collecting all top-down oriented symbol paths of length 3. Additionally, to abstract from variable names and to deal with possible collisions of Skolem symbols, all variables are replaced by a special name $\odot$ and all Skolem symbols by $\circledast$. **Horizontal Features** introduce for every term $f(t_1, \ldots, t_n)$, a new feature $f(s_1, \ldots, s_n)$, where s_i is the top-level symbol of t_i. **Count Features** include the clause length, literal counts, and similar statistics. **Conjecture Features** embed the conjecture to be proved in the feature vector. Thusly, ENIGMA is able to provide goal specific predictions. **Parent Features** represent a clause by features

(concatenated or summed) of its parents. **Feature Hashing** is an important step towards large data in ENIGMA [11]. It significantly reduces the feature vector size and thusly allows handling of larger data. Each feature is represented by a unique string identifier. This string is passed through the hashing function and the hash modulo the selected *hash base* is used as the feature index. **Symbol Anonymization** allows to abstract from specific symbol names [25]. During the extraction of clause features, all symbol names are replaced by symbol arities, keeping only the information whether the symbol is a function or a predicate. In this way, a decision tree classifier does not depend on symbol names, at the price of symbol collisions, which are however empirically mitigated by collecting longer paths as features.

3.3 Graph Neural Network (GNN) Classifiers

Anonymizing *graph neural networks* provide an alternative approach for abstracting from specific terminology. ENIGMA uses [25] a symbol-independent GNN architecture initially developed for guiding tableaux search [39] implemented in TensorFlow [1]. A set of clauses is directly represented by a hypergraph with three kinds of nodes for clauses, subterms/literals, and symbols. Relationships among the objects are represented by various graph edges, which allow the network to distinguish different symbols while abstracting from their names.

The GNN layers perform message passing across the edges, so the information at every node can get to its neighbors. This allows the network to see how the symbols are used without knowing their names. We always classify the new clauses together with the initial clauses which provide the context for the meaning of the anonymized symbols. During the ATP evaluation, predictions of hundreds of generated clauses are computed at once in larger batches, with the context given both by the initial and the processed clauses. The context can be either *fixed*, containing an initial segment of the initial and processed clauses, or it can be a *shifting context* using a window of clauses with the best GNN evaluation.

3.4 Additional Related Techniques

GPU Server Mode allows using GPUs for real-time evaluation [20]. To reduce the GPU overhead of model loading, we developed a Python GPU server, with preloaded models that can distribute the evaluation over several GPUs. E Prover clients communicate with the server via a network socket. We fully utilize our physical server[3] when we run 160 instances of E prover in parallel. Running both the server and clients on the same machine reduces the network communication overhead.

Leapfrogging addresses the problem of evolving context when new given clauses are selected [10]. We run ENIGMA with a given abstract limit and generate a larger set of clauses. Then we run a premise selection on these generated clauses (e.g., only processed clauses), take the good clauses, and use them as input for a new ENIGMA run. A related *split/merge* method involves repeatedly splitting the generated clauses into components that are run separately and then merged with premise selection. This is inspired by the idea that harder problems consist of components that benefit from such divide-and-conquer approaches.

Deepire is an extension [51, 52] of Vampire [37] by machine-learned clause selection guidance, generally following the ENIGMA-style methodology. It is distinguished by its use of recursive neural networks for classifying the generated clauses based solely on their derivation history. Thus Deepire does not attempt to read "what a clause says", but only

[3] 36 hyperthreading Intel(R) Xeon(R) Gold 6140 CPU @ 2.30 GHz cores, 755 GB of memory, and 4 NVIDIA GeForce GTX 1080 Ti GPUs.

bases its decisions on "where a clause is coming from". This allows the clause evaluation to be particularly fast, while still being able to recognize and promote useful clauses, especially in domains with distinguished axioms which reappear in many problems.

4 Learning Premise Selection From the MML

When an ATP is used over a large ITP library, typically only a small fraction of the facts are relevant for proving a new conjecture. Since giving too many redundant premises to the ATP significantly decreases the chances of proving the conjecture, premise selection is a critical task. The most efficient premise selection methods use *data-driven* or *machine-learning* approaches. If T is a set of theorems with their proofs and C is a set of conjectures without proofs, the task is to learn a (statistical) model from T, which for each conjecture $c \in C$ will rank (or select a subset of) its available premises according to their relevance for producing an ATP proof of c. Two main machine learning settings can be used. In **Multilabel classification**, premises used in the proofs are treated as opaque labels and a machine learning model is trained to label conjectures based on their features. **Binary classification** aims to recognize pairwise-relevance of the (*conjecture, premise*) pairs, i.e. to estimate the chance of a premise being relevant for proving the conjecture based on the features of both the conjecture and the premise.

The first setting is suitable for simpler, fast ML methods, like k-NN or Naive Bayes – these are described in Section 4.1. The second setting (Section 4.2) allows using more powerful ML architectures, like GBDTs and GNNs (Sections 3.3 and 3.2). However, this setting also requires selecting *negative examples* for training [41], which increases its complexity.

4.1 Multilabel Premise Selection ($\mathcal{K}$, $\mathcal{N}$, $\mathcal{R}$)

Naive Bayes and k-nearest neighbors were the strongest selection methods in the Mizar40 evaluation [32]. In this work, we improve them and apply them together with newer methods.

***k*-NN ($\mathcal{K}$).** The k-nearest neighbours algorithm, when applied to premise selection, chooses k facts closest to the conjecture in the feature space and selects their dependencies. Already known modifications of the standard k-NN include considering the number of dependencies (proofs with more dependencies are longer and thus less important) and TF-IDF (rare features are more important) [30]. Additionally, we realize that we do not need to fix the k. Instead, we consider a small k and if the number of scored dependencies is too low, we increase the k and update the dependencies. This is repeated until the requested number of predictions is obtained. The k-NN-based predictions with fixed k will be denoted, e.g., by $\mathcal{K}_{512}$, while with variable k this will be $\mathcal{K}^{\text{fea}}_{\text{var}}$, where *fea* specifies the features used.

Naive Bayes ($\mathcal{N}$). The sparse Naive Bayes algorithm estimates the relevance of a fact F by the conditional probability of F being useful (estimated from past proof statistics) under the condition of the features being present in the conjecture (again estimated from statistics). We also consider *extended features* of F, i.e., features of F and features of facts proved using F. Together with premise selection-specific weights this improves on the basic Naive Bayes and has already been used in HolyHammer and Sledgehammer. A complete derivation of the algorithm is given in [6]. The Naive Bayes predictions will be denoted $\mathcal{N}_{\text{fea}}$.

These algorithms can be parametrized by more complex *features*. We considered: `cp` for constants and paths (Section 3.2) in the term graph, `sub` for subterms, `au` for anti-unification features [33], `eni` for online ENIGMA features discussed in Section 3.2 and `uni` for the union

of all above. Finally, these algorithms also support the *chronological* mode, which in the learning phase discards proofs that use facts introduced after the current conjecture in the Mizar canonical order (`MML.LAR`). This slightly weakens the algorithms, but is compatible with the previous Mizar40 premise selection evaluation [32]. These will be marked by ^{chrono}.

Dependent Selection with RNNs ($\mathcal{R}$). Premise selection methods were originally mainly based on *ranking* the facts *independently* with respect to the conjecture. The highest ranked facts are then used as axioms and given to the ATP systems together with the conjecture. Such approaches (used also with GBDTs), although useful and successful, do not take into account that the premises are *not* independent of each other. Some premises complement each other better when proving a particular conjecture, while some highly-ranked premises might be just minor variants of one another. Recurrent neural network (RNN) encoder-decoder models [9] and transformers [60] (language models) turn out to be suitable ML architectures for modeling such implicit dependencies. Such models have been traditionally developed for natural language processing, however, recently they are also increasingly used in symbolic reasoning tasks [12, 16, 38, 43, 61], including premise selection [42].

4.2 Premise Selection as Binary Classification ($\mathcal{L}$, $\mathcal{G}$)

Gradient Boosted Decision Trees ($\mathcal{L}$). We use GBDTs (LightGBM) also for premise selection in the binary mode. They are faster to train than the deep learning methods, perform well with unbalanced training sets, and handle well sparse features. We fix the LightGBM hyperparameters here based on our previous experiments with applying GBDTs to premise selection [41]. In the binary setting, the GBDT scores the pairwise relevance of the conjecture and a candidate premise. Because the number of possible candidates is large (all preceding facts in the large ITP library), we first use the cheaper k-nearest neighbors algorithm to pre-filter the available premises. The predictions from LightGBM will be denoted as $\mathcal{L}$ below.

Dependent Selection with GNNs ($\mathcal{G}$). The message-passing GNN architecture described in Section 3.3 can also be applied to premise selection. Like RNNs, it can also take into account the dependencies between premises. As the GNN is relatively slow, we will use it in combination with a simpler premise selection method, such as k-NN, preselecting 512 facts. We will denote GNN predictions by $\mathcal{G}$ below. Both $\mathcal{L}$ and $\mathcal{G}$, can be indexed with the threshold on the score (like $\mathcal{L}_{0.1}$ or $\mathcal{G}_{-1}$), used to differentiate useful and useless clauses.

4.3 Ensemble Methods for Premise Selection ($\mathcal{E}$)

There are several ways how we can combine the premise selection methods discussed in previous subsections. Naturally, using different methods for different strategies works well, however, we also found that combining the predictions obtained from several methods and using them for a single prover run gives good and complementary results. Since prediction scores resulting from different algorithms are often incomparable [40], we only use the rankings produced by the various methods and based on this we create a combined ranking. We have compared several ways to combine rankings in previous work [30] and found that several averages work well: arithmetic mean, minimum, and geometric mean, with the harmonic mean giving experimentally the best results. Additionally, we add weights to the different combined methods. The weights give more priority to a stronger prediction method, but allow it to benefit from the simpler ones overall (by picking up some lost facts). Given

predictions from n different methods and method weights $w_1, \ldots, w_n$, assume that a fact has been ranked as r_1-th by the first method until and r_n-th by the last one. Then, the ensemble method would give that fact a score of $1/\sum_{i=1}^{n} \frac{w_i}{r_i}$. The scores of the facts obtained in this way are sorted, to get a ranking of all facts. The ensemble predictions will be denoted by $\mathcal{E}$, with methods and their weights in the super and subscript, for example $\mathcal{E}^{\mathcal{K},\mathcal{N},\mathcal{G}}_{0.25,0.25,0.5}$.

4.4 Subproblem Based Premise Minimization ($\mathcal{M}$)

The proof dependencies obtained by successful ATP runs typically perform better as data for premise selection than the dependencies from the human-written ITP proofs [7,31]. However, some Mizar proofs are hundreds of lines long and it is so far unrealistic to raise the 75 % ATP performance obtained here in the bushy setting to a number close to 100 %. This means that if we used only ATP-based premise data, we would currently miss in the premise selection training 25 % of the proof dependency information available in the MML.

To remedy that, we newly use here *subproblem based premises*. The idea behind this is that a theorem with a longer Mizar proof consists of a series of natural deduction steps that typically have to be justified. Once ATP proofs of all such steps (we call them subproblems) for a given toplevel theorem are available, they can be used to prune the (overapproximated) set of human-written premises of the theorem. Such minimization also increases the chance of proving the theorem directly. In more detail, we consider the following approaches: (1) Use the premises from only ATP-proved subproblems, ignoring unproved subproblems. (2) Add to (1) all explicit Mizar premises of the theorem (possibly ignoring some background facts). (3) Add to (2) also the (semi-explicit) definitional expansions detected by the natural deduction module. (4) Add to (3) also some of the background premises, typically those ranked high by the trained premise selectors. When using (1) and (3), we were able to prove more than 1000 hard theorems (see Table 1 in Section 6.1). We also use (3) as additional proof dependencies for ATP-unproved theorems when training premise selectors (Section 6.2).

5 Strategies and Portfolios

Strategies. E, ENIGMA, Vampire and Deepire are parameterized by ATP strategies and their combinations. While ENIGMA-style guidance typically involves the application of a larger (neural, tree-based, etc.) and possibly slower statistical model to the clauses, standard ATP strategies typically consist of much faster clause evaluation functions and programs written in a DSL provided by the prover. Such programs can again be invented and learned in various ways for particular classes of problems. For the experiments here we have used many ATP strategies invented automatically by the BliStr/Tune systems [26,27,57]. They implement feedback loops that interleave targeted *parameter search* on problem clusters using engines like ParamILS [23], with a large-scale evaluation of the invented strategies used for evolving the problem clustering. Starting from few strategies, BliStr/Tune typically evolve each strategy on the problems where the strategy performs best. During our experiments with the systems we have developed several thousand E Prover strategies, many of them targeted to Mizar problems. Some of these are mentioned in the experiments in Section 6.

Robust Portfolios. Larger AI/TP systems and metasystems rely on portfolios [53] of complementary strategies that attack the problems serially or in parallel using a global time limit. In the presence of premise selection and multiple ATPs, such portfolios may consist of tens to hundreds of different methods. The larger the space of methods, the larger is the risk of overfitting the portfolio during its construction on a particular set of problems. For

example, naive construction of "optimal" portfolios by using SAT solvers for the set-cover problem (where each strategy covers some part of the solution space) often leads to portfolios that are highly specialized to the particular set of problems. This is mitigated in more robust methods such as the *greedy cover*, however, the overfitting there can still be significant. E.g., a 14-strategy greedy cover built in the Mizar40 experiments [32] solved 44.1 % of the random subset used for its construction, while it solved only 40.6 % of the whole MML, i.e., 8 % less.

To improve on this, we propose a more robust way of portfolio construction here, based again on the machine-learning ideas of controlling overfitting. Instead of simply constructing one greedy cover C (with a certain time budget) on the whole development set D and evaluating it in the holdout set H, we first split D randomly into two equal size halves D_1 and D_2. Then we construct a greedy cover C_1 only on D_1, and evaluate its performance also on D_2 and the full set D. This is repeated n times (we use $n = 1000$), which for large enough n typically guarantees that the greedy cover C_1^i will for some of the random splits D_1^i, D_2^i overfit very little (or even underfit). This can be further improved by evaluating the best (strongest and least overfitting) covers on many other random splits and selecting the most robust ones. We use this in Section 6 to build a portfolio that performs only 3.5 % worse on the (unseen) holdout set than on the development set used for its construction.

6 Experiments and Results

6.1 Bushy Experiments and Timeline

The final list of all 43 717 Mizar problems proved by ATPs in our evaluation is available on our web page.[4] The approximate timeline of the methods and the added solutions is shown in Table 1. This was continuously recorded on our web page,[5] which also gives an idea of how the experiments progressed and how increasingly hard problems were proved.

The large evaluation started in April 2020, as a follow-up to our work on ENIGMA Anonymous [25]. By combining the methods developed there and running with higher time limits, the number of problems proved by ENIGMA in the bushy setting reached 65.65 % in June 2020. This was continued by iterating the learning and proving in a large Malarea-style feedback loop. The growing body of proofs was continuously used for training the graph neural networks and gradient boosted guidance, which were used for further proof attempts, combined with different search parameters and later used also for training premise selection.

This included many grid searches on a small random subset of the problems over the thousands of differently trained GNNs and GBDTs corresponding to the training epochs, and then evaluating the strongest and most complementary ENIGMAs using the differently trained GNNs and GBDTs on all, or just *hard* (the so far ATP-unproved), problems. The total number of the saved snapshots of the GNNs corresponding to the training epochs and usable for the grid searches and full evaluations reached 15 920 by the end of the experiments in September 2021.[6] The longest GNN training we did involved 964 epochs and 12 days on a high-end NVIDIA V100 GPU card.[7] The GNN training occasionally (but rarely) diverged after hundreds of epochs, which we handled by restarts.

[4] `http://grid01.ciirc.cvut.cz/~mptp/00proved_20210902`
[5] `https://github.com/ai4reason/ATP_Proofs`
[6] For the grid searches, this was compounded by further parameters of the ENIGMA and E strategies.
[7] We generally use the same GNN hyper-parameters as in [25, 39] with the exception of the number of *layers* that varied here between 5 and 12, providing tradeoffs between the GNN's speed and precision.

Table 1 Timeline of the experiments. $\mathcal{B}$ are standard bushy premises, $\mathcal{M}$ are subproblem-minimized premises, $\mathcal{G}$, $\mathcal{L}$, and $\mathcal{K}$ are GNN/LightGDB/kNN-based premises, and $\mathcal{E}$ their ensembles.

solved	[%]	*date*	*premises*	*methods/notes*
38k	65.65	Jun 2020	$\mathcal{B}$	ENIGMA, reported on July 2nd at IJCAR'20[8]
40 268	69.57	Oct 2020	$\mathcal{B}$	ENIGMA
40 994	70.83	Nov 12	$\mathcal{M}$	ENIGMA, heuristic premise minimization
41 169	71.13	Nov 12	$\mathcal{M}$	Vampire with 300 s limit adds 175
41 792	72.20	Nov 27	$\mathcal{M}$	E/ENIGMA/Vampire with more premise minimization
42 206	72.92	Dec 7	$\mathcal{M}$	E/ENIGMA/Vampire with more premise minimization
42 471	73.38	Jan 6	$\mathcal{G}$, $\mathcal{E}$	E with BliStr/Tune strategies on $\mathcal{G}$, $\mathcal{E}$ premises
42 519	73.46	Jan 10	many	ENIGMA runs on all training predictions
42 826	73.99	May 14	$\mathcal{G}$,$\mathcal{L}$,$\mathcal{K}$	Vampire/Deepire runs – FroCoS'21 [52]
43 414	75.01	Jul 26	$\mathcal{M}$,$\mathcal{B}$	2,3-phase ENIGMA, leapfrogging
43 524	75.20	Aug 21	$\mathcal{M}$	3-phase ENIGMA, shifting context, leapfrog., fwd subsump.
43 599	75.33	Aug 26	$\mathcal{L}$	3-phase ENIGMA, leapfrogging, fwd. subsumption
43 717	75.53	Sep 2	$\mathcal{M}$	mainly Vampire/Deepire

The total number of proofs that we trained the ENIGMA guidance on eventually reached more than three million, which in a pickled and compressed form take over 200 GB. Since the full data do not fit into the main memory of even large servers equipped for efficient GPU-based neural training, we have programmed custom pipelines that continuously load, mix and unload smaller chunks of data used for the ENIGMA training. For many problems, we obtained hundreds of different proofs, while for some problems we may have only a single proof. This motivated further experiments on how and with what frequency the different proofs should be represented in the training data. This was a part of the larger task of *training data normalization*, which included, e.g., removing or pruning very large proof searches in the training data that would cause memory-based GPU crashes.

The 75 % milestone was reached on July 26 2021[9] by using the freshly developed 2 and 3-phase ENIGMAs, together with differently parameterized leapfrogging (Section 3.4) runs. The strongest single 3-phase ENIGMA strategy has reached 56.4 % performance in 30 s on the bushy problems when trained and evaluated in a rigorous train/dev/holdout setting [20]. This best ENIGMA uses a parental threshold of 0.01, 2-phase threshold of 0.1, and context and query sizes of 768 and 256. Its (server-based) GNN has 10 layers trained on at most three proofs for each problem in the training set. See also Section 6.6 for its evaluation on a set of completely new 13 370 problems in 242 new articles of a later version of MML.

6.2 Training Data for Premise Selection

After several months of running the learning/proving loop in various ways on the problems, we used the collected data for training premise selection methods. In particular, at that point, there were 41 504 ATP-proved problems for which we typically had many alternative proofs and sets of premises, yielding 621 642 unique ATP proof dependencies. Since in the hammering scenarios we can also analyze the human-written proofs and learn from them, we have added for each ATP-unproved problem P its premises obtained by taking the union of

[8] https://youtu.be/XojOEpZfH4Y?t=673
[9] https://github.com/ai4reason/ATP_Proofs/blob/master/75percent_announce.md

the ATP dependencies of all subproblems of P. In other words, we use subproblem-based premise minimization (Section 4.4) for the remaining hard problems. This adds 16 651 examples to the premise selection dataset. This dataset of 638 293 unique proof dependencies is then used in various ways for training and evaluating the premise selection methods on MML. In comparison with the Mizar40 experiments this is about six times more proof data. As usual in machine learning experiments, we also split the whole set of Mizar problems into the *train*ing, *devel*opment, and *holdout* subsets, using a $90 : 5 : 5$ ratio. This yields 52 125 problems in the training set, 2896 in devel, and 2896 in the holdout set.

6.3 Training the Premise Selectors

We first train kNN and naive Bayes in multiple ways on the training subset using the different features (Section 4.1) and their combinations. For training the GNN and LightGBM, we first use kNN-based pre-selection to choose 512 most relevant premises for each problem. When training, we add for each example its positives (the real dependencies) and subtract them from the 512 premises pre-selected by kNN, thus forming the set of the negatives for the example. The GNN and LightGBM are thus trained to correct the mistakes done by kNN (a form of *boosting*). When predicting, this is done in the same way, i.e., first we use the trained kNN to preselect 512 premises which are then ranked by the GNN/LightGBM. We use both score thresholds (e.g., including all premises with score better than 0, -1 or -3), and fixed-sized slices as in other premise selection methods. With the same best version of ENIGMA, the strongest GNN-based predictor ($\mathcal{G}_{-1}$) solves 1089 problems compared to 870 solved when using the baseline kNN, which is a large (25.2 %) improvement. The GNN also outperforms LightGBM, which overfits more easily on the training data. Table 2 shows the performance on the devel and holdout sets of the main methods used in the evaluation.

6.4 ENIGMA Experiments on the Premise Selection Data

First, to train ENIGMA on the premise selection problems, we perform several prove/learn iterations with ENIGMA/GBDT on our premise slices. In loop (1), we start with three selected slices $\mathcal{G}_{-1}$, $\mathcal{L}_{0.1}$, and $\mathcal{K}_{64}$, which were found experimentally to be complementary. We evaluate strategy $\mathcal{S}_1$ (bls0f17) on the three slices obtaining 20 604 proved training problems. We train several decision tree (GBDT) models with various learning hyperparameters (tree leaves count, tree depth, ENIGMA features used). We use all the training proofs available. In loop (2), we evaluate several ENIGMA models trained on $\mathcal{B}$ (bushy problems) to obtain additional training data. After few training/evaluation iterations, the training data might start accumulating many proofs for some (easier) problems solved by many strategies. From loop (2) on, we, therefore, use only a limited number of proofs per problem. We either select randomly up to 6 proofs for each problem, or we select only specific proofs (e.g., the shortest, longest, and one medium-length proof). In loop (3), additional training data are added by ENIGMA/GNN runs on the premise slices, with GNN trained on the GBDT runs. In loop (4), we consider training data from 7 additional slices (variants of $\mathcal{G}$, $\mathcal{L}$, $\mathcal{K}$), obtained by running ENIGMA models trained of bushy problems. In loop (5), we extend the training data with bushy proofs of unsolved training problems obtained by our various previous efforts.

Starting from 1215 solved development problems, we ended up with 1735 problems solved after the fifth iteration. While we train GBDT models only on few selected slices, we evaluate the models on many more, up to 56, development slices covering all families $\mathcal{G}$, $\mathcal{L}$, $\mathcal{K}$, $\mathcal{E}$, and $\mathcal{N}$. We report the increasing number of training problems (*trains*) and the total of number of solved development problems by all the evaluated strategy/slice pairs (*devel* union). Since every strategy/slice pair is evaluated in 10 seconds, we construct the greedy cover of best 42

Table 2 Machine learning evaluation of the premise selection models on the **D**evelopment and **H**oldout datasets. Note that the evaluation of GNN is presented here only for completeness, in practice we use it with a score-based threshold and fewer premises.

Model	100-Cover		100-Prec		Recall		AUC		Avg. Rank	
	D	H	D	H	D	H	D	H	D	H
$\mathcal{K}^{\text{cp}}_{\text{var}}$	83.3	82.3	8.837	8.713	386.8	401.9	92.03	91.27	90.17	97.98
$\mathcal{K}^{\text{au}}_{\text{var}}$	83.5	82.7	8.855	8.754	383.32	401.54	92.13	91.36	89.21	97.19
$\mathcal{K}^{\text{mi}}_{\text{var}}$	82.6	81.8	8.700	8.596	401.89	418.40	91.32	90.59	97.30	104.88
$\mathcal{K}^{\text{s0}}_{\text{var}}$	83.6	82.9	8.851	8.785	382.31	399.10	92.19	91.39	88.53	96.84
$\mathcal{N}_{\text{cp}}$	87.8	87.0	9.739	9.665	300.49	310.72	94.77	94.32	62.64	67.51
$\mathcal{N}_{\text{au}}$	88.0	87.5	9.748	9.714	298.66	307.82	94.84	94.44	61.99	66.31
$\mathcal{N}_{\text{mi}}$	83.3	83.5	9.358	9.367	382.87	379.24	92.53	92.41	85.39	86.88
$\mathcal{N}_{\text{s0}}$	88.3	87.5	9.776	9.720	299.10	308.67	94.85	94.41	61.85	66.60
$\mathcal{N}_{\text{mi,chrono}}$	83.9	82.7	9.151	9.010	384.68	393.37	92.33	91.75	87.23	93.27
$\mathcal{L}$	82.9	83.1	9.077	9.090	410.06	408.06	91.53	91.26	95.11	97.74
$\mathcal{G}$	87.4	86.3	9.408	9.282	241.22	249.32	88.45	87.42	66.69	71.87
$\mathcal{E}^{\mathcal{N},\mathcal{K}}_{.5,.5,\&\text{avg}}$	87.8	87.1	9.606	9.522	291.27	304.43	95.06	94.53	59.81	65.47
$\mathcal{E}^{\mathcal{N},\mathcal{K}}_{.5,.5,\&\text{geo}}$	89.4	88.7	9.806	9.733	277.75	288.53	95.53	95.04	55.13	60.27
$\mathcal{E}^{\mathcal{N},\mathcal{K}}_{.5,.5,\&\text{har}}$	89.4	88.9	9.822	9.780	276.34	286.23	95.53	95.08	55.06	59.94
$\mathcal{E}^{\mathcal{N},\mathcal{K}}_{.5,.5,\&\text{min}}$	89.0	88.4	9.753	9.707	279.88	289.41	95.37	94.95	56.70	61.19
$\mathcal{E}^{\mathcal{N},\mathcal{G},\mathcal{K}}_{.5,.25,.25}$	92.1	91.1	10.237	10.160	228.79	248.16	96.64	96.20	44.06	48.76
$\mathcal{E}^{\mathcal{N},\mathcal{K},\mathcal{L},\mathcal{G}}_{.5,.2,.2,.1}$	92.5	91.5	10.297	10.219	210.31	227.74	96.93	96.56	41.20	45.17
$\mathcal{E}^{\mathcal{N},\mathcal{G},\mathcal{K}}_{.33,.33,.33}$	91.3	90.4	10.091	10.014	261.10	272.85	96.20	95.71	48.51	53.64

strategy/slice pairs, to approximate the best possible result obtainable in 420 s (see columns *devel cover*). Since the development set has not been used in any way to train the GBDT models, we can see this as an approximation of the best possible result on the holdout set.

We reach 55.59 % of problems solvable in 420 s, only with ENIGMA/GBDT models. To compare this result to other methods, we construct compatible greedy covers for E Prover in *auto-schedule mode*, and for Vampire in *CASC mode*, that is, in their respective strongest default settings. We evaluate both provers on all 56 development slices, with 30 s limit per problem. For each prover, we construct a greedy cover of best 14 slices, again approximating the best possible result obtainable in 420 s. BliStr/Tune is our previously invented portfolio of 15 E strategies for Mizar bushy problems. We evaluate all 15 strategies on all 56 slices with 2 seconds per problem. Similarly, the greedy cover of length 210 is constructed. The column *pairs* specifies the greedy cover length considered in each case. The time limit for each strategy/slice pair is 420/*pairs*.

The training data obtained in five loops were finally used to train new ENIGMA/GNN models for premise selection slices. Various GNN models were trained (various numbers of layers, networks from various epochs) and evaluated with the limit of 5 s. As before, we construct the greedy cover of length 84 to simulate the best possible run in 420 s. ENIGMA/GNN performs even better then ENIGMA/GBDT, solving 57.66 % problems. The two ENIGMA/* portfolios cover together 1701 development problems (in 840 s), suggesting a decent complementarity of the methods. Note that only the ENIGMA/GBDT strategies can cover up to 1735 (see column *devel* on the left), which is 59.9 % of the development set.

Table 3 Training of ENIGMA/GBDT models (left), and best covers of development set (right).

loop	*trains*	*devel* (union)	*devel cover* (in 420 s)	[%]
init	20 604	1215	-	-
(1)	25 240	1601	1516	52.33
(2)	25 725	1669	1555	53.69
(3)	25 887	1679	1560	53.88
(4)	29 266	1716	1591	54.94
(5)	37 053	1735	1610	55.59

prover (420 s)	cover	*pairs*	[%]
E 2.6 (auto-schedule)	1430	14	49.38
Vampire 4.0 (CASC)	1536	14	53.03
BliStr/Tune	1582	210	54.62
ENIGMA/GBDT	1610	42	55.59
ENIGMA/GNN	1670	84	57.66

Most of our ENIGMA models are combined with the baseline strategy bls0f17. This together with bls05fc are two strategies invented by BliStr/Tune [24] which perform well on premise selection data. We additionally use another two older BliStr [57] strategies mzr02 and mzr03 which perform well on bushy problems. We usually combine training data only from strategies with compatible term ordering and literal selection setting. However, data from strategies with incompatible orderings, were found useful when used in a reasonably small amounts. Few other BliStr and Vampire strategies, together with E in the *auto* mode, are used to gather additional solved development problems. With all our methods (ENIGMA & BliStr/Tune) and with additional Vampire runs of selected strategies, we have solved more than 62.7 % development problems. These results provide training data for the construction of the final holdout portfolio, as described in the next section.

6.5 Final Hammer Portfolio

With the large database of the development results of the systems run on the premise slices, we finally construct our ultimate hammering portfolio. For that, we use the *robust portfolio construction* method described in Section 5. In particular, we randomly split the development set into two equal-sized parts, and compute the 420 s greedy cover using our whole database of results on the first part. This greedy cover is evaluated on the second part, thus measuring the overfitting. This randomized procedure is repeated one thousand times. Then we (manually) select the 20 strongest and least overfitting portfolios and evaluate each of them on 80 more random splits, thus measuring how balanced they are on average. Typically, they reach up to 60.5 % performance on the whole devel set, so we choose a threshold of 59.5 % on the 160 random halves to measure the imbalance. The most balanced portfolio wins with 135 of the 160 random halves passing the threshold.

This final 420-second portfolio has 95 slices that solve 1749 (60.4 %) of the devel problems and 1690 (58.36 %) of the holdout problems. Table 4 shows the initial segment of 13 slices of this portfolio with the numbers of problems solved. The full portfolio is presented in Table 5. The first number t is the number of seconds to run the slice. The *base* column specifies the ATP strategy used, and *ENIGMA* describes what kinds of ENIGMA models are used (if any). We can see that GNN models dominate the schedule with fast runs. The schedule is closed by longer runs, notably also GBDT models, which while evaluated in a single-CPU setting, need several seconds to load the model. This means that we are favoring the GNN ENIGMAs thanks to the use of the preloaded GNN server, and a further improvement is likely if we also preload the GBDT models. Our single strongest GNN-based strategy solves 1178 of the holdout problems in 30 s using the $\mathcal{G}_{-1}$ predictions. This is 39.5 %, which is only 1.1 % less than the 40.6 % solved by the full 420 s portfolio constructed in the Mizar40 experiments.

Table 4 The 13-slice prefix of the final portfolio of the 95 slices. Each column presents the premise selection method, the ATP method, and the number of problems solved up this slice cumulatively on the development and holdout sets. "V" stands for Vampire and "GNN" is ENIGMA/GNN model based on bls0f17. Moreover, $\mathcal{E}_{5221} = \mathcal{E}^{\mathcal{N},\mathcal{K},\mathcal{L},\mathcal{G}}_{.5,.2,.2,.1}$ and $\mathcal{E}_{55} = \mathcal{E}^{\mathcal{N},\mathcal{K}}_{.5,.5,\text{avg}}$ and $\mathcal{E}_{533} = \mathcal{E}^{\mathcal{N},\mathcal{G},\mathcal{K}}_{.5,.25,.25}$.

$\mathcal{G}$	$\mathcal{E}_{5221}$	$\mathcal{L}$	$\mathcal{E}_{5221}$	$\mathcal{N}_{\text{uni}}$	$\mathcal{E}_{5221}$	$\mathcal{N}_{\text{eni}}$	$\mathcal{N}_{\text{eni}}$	$\mathcal{E}_{5221}$	$\mathcal{G}$	$\mathcal{E}_{55}$	$\mathcal{E}_{533}$	$\mathcal{E}_{533}$
GNN	GNN	GNN	GNN	V	V	GNN	GNN	GNN	GNN	V	V	GNN
984	1142	1215	1263	1297	1325	1346	1370	1381	1393	1405	1419	1444
1013	1157	1240	1275	1305	1321	1346	1364	1378	1386	1398	1407	1436

6.6 Transfer to MML 1382

In the final experiment, we run for 120 s the best trained ENIGMA (3-phase, see Section 6.1) on the bushy problems from a new version of Mizar (1382) that has 242 new articles and 13 370 theorems in them. ENIGMA not only never trained on any of these articles, but also never saw the new terminology introduced there. We also run the standard E auto-schedule for 120 s on the new version. ENIGMA proves 37 094 (52.7 %) of the 70 396 problems in the new library, while the E auto-schedule proves 24 158 (34.32 %) of them. ENIGMA thus improves over E by 53.55 % on the new library. We compare this with the old MML, where the trained ENIGMA solves 34 528 (59.65 %) of the 57 880 problems, and E solves 22 119 (38.22 %), i.e., the relative improvement there is 56.10 %.

Surprisingly, just on the new 13 370 theorems – more than half of which contain new terminology – the ratio of ENIGMA-proved to E-proved problems is 5934 to 3751, i.e., ENIGMA is here better than E by 58.20 %. These numbers show that the performance of our *anonymous* [25] logic-aware ML methods, which learn only from the structure of mathematical problems, is practically untouched by the transfer to the new setting with many new concepts and lemmas. This is quite unusual in today's machine learning which seems dominated by large language models that typically struggle on new terminology.

7 Proofs

As the main experiments progressed from spring 2020 to summer 2021, we have collected interesting examples of automatically found proofs and published their summary descriptions on our web page.[10] As of September 2021 there were over 200 of such example proofs, initially with ATP length in tens of clause steps, and gradually reaching hundreds of clause steps. Initially these were proofs found in the bushy setting, with proofs done in the chainy (premise-selection) setting added later, typically to show the effect of alternative premises.

One of the earliest proofs that we put on the web page is `NEWTON:72` proving that for every natural number there exists a larger prime:

```
for l being Nat ex p being Prime st p is prime & p > l
```

The ENIGMA proof starts from 328 preselected Mizar facts which translate to 549 initial clauses. The search is guided by a particular version of the GNN running at that time (April 2020) on the CPU. Since this is relatively costly, the proof search generated only 2856 nontrivial clauses in 6 s, doing 734 nontrivial given clause loops. The final proof takes 83 clausal steps, and uses 38 of the 328 initially provided steps. Many of them replay the

[10] `https://github.com/ai4reason/ATP_Proofs`

Table 5 The final Mizar hammer portfolio for 420 s.

t	*base*	*ENIGMA*	*slice*
2	bls0f17	GNN	$\mathcal{G}_{-1}$
2	bls0f17	GNN	$\mathcal{E}^{\mathcal{N},\mathcal{K},\mathcal{L},\mathcal{G}}_{.5,.2,.2,.1}$
2	bls0f17	GNN	$\mathcal{L}_{0.1}$
2	bls0f17	GNN	$\mathcal{E}^{\mathcal{N},\mathcal{K},\mathcal{L},\mathcal{G}}_{.5,.2,.2,.1}$
2	vampire	-	$\mathcal{N}^{\text{uni}}$
2	vampire	-	$\mathcal{E}^{\mathcal{N},\mathcal{K},\mathcal{L},\mathcal{G}}_{.5,.2,.2,.1}$
2	bls0f17	GNN	$\mathcal{N}^{\text{eni}}$
2	bls0f17	GNN	$\mathcal{N}^{\text{eni}}$
2	bls0f17	GNN	$\mathcal{E}^{\mathcal{N},\mathcal{K},\mathcal{L},\mathcal{G}}_{.5,.2,.2,.1}$
2	bls0f17	GNN	$\mathcal{G}_{-3}$
2	vampire	-	$\mathcal{E}^{\mathcal{N},\mathcal{K}}_{.5,.5}$
2	vampire	-	$\mathcal{E}^{\mathcal{N},\mathcal{G},\mathcal{K}}_{.5,.25,.25}$
5	bls0f17	GNN	$\mathcal{E}^{\mathcal{N},\mathcal{G},\mathcal{K}}_{.5,.25,.25}$
2	vampire	-	$\mathcal{E}^{\mathcal{N},\mathcal{K}}_{.5,.5}$
2	vampire	-	$\mathcal{K}^{\text{ali}}_{\text{var}}$
2	mzr02	-	$\mathcal{K}^{\text{cp}}_{\text{var}}$
2	bls0f17	GNN	$\mathcal{G}_{0}$
2	vampire-16	-	$\mathcal{G}_{-5}$
2	vampire	-	$\mathcal{N}^{\text{uni}}$
2	bls0f17	GNN	$\mathcal{N}^{\text{eni}}$
5	bls0f17	GNN	$\mathcal{N}^{\text{au}}$
2	vampire	-	$\mathcal{E}^{\mathcal{N},\mathcal{K}}_{.5,.5}$
2	vampire	-	$\mathcal{E}^{\mathcal{N},\mathcal{K}}_{.5,.5}$
5	bls0f17	GNN	$\mathcal{E}^{\mathcal{N},\mathcal{K}}_{.5,.5}$
2	bls0f17	GNN	$\mathcal{E}^{\mathcal{N},\mathcal{G},\mathcal{K}}_{.5,.25,.25}$
2	vampire-16	-	$\mathcal{E}^{\mathcal{N},\mathcal{K},\mathcal{L},\mathcal{G}}_{.5,.2,.2,.1}$
5	bls0f17	GNN	$\mathcal{L}_{0.05}$
2	vampire-18	-	$\mathcal{G}$
2	mzr22	-	$\mathcal{E}^{\mathcal{N},\mathcal{K},\mathcal{L}}_{.5,.2,.3}$
2	vampire	-	$\mathcal{E}^{\mathcal{N},\mathcal{K}}_{.5,.5,geo}$
2	vampire	-	$\mathcal{E}^{\mathcal{N},\mathcal{K}}_{.5,.5}$
2	vampire	-	$\mathcal{K}^{\text{cp}}_{\text{var}}$
2	BliStr-edc9	-	$\mathcal{E}^{\mathcal{N},\mathcal{K},\mathcal{L}}_{.5,.2,.3}$
2	vampire	-	$\mathcal{E}^{\mathcal{N},\mathcal{K}}_{.5,.5,chrono}$
2	vampire	-	$\mathcal{N}^{\text{uni}}$
5	bls0f17	GNN	$\mathcal{E}^{\mathcal{N},\mathcal{G},\mathcal{K}}_{.5,.25,.25}$
10	mzr02	GNN	$\mathcal{L}_{0}$
5	bls0f17	GNN	$\mathcal{E}^{\mathcal{N},\mathcal{K},\mathcal{L},\mathcal{G}}_{.5,.2,.2,.1}$
5	bls0f17	GNN	$\mathcal{N}^{\text{au}}$
2	bls0f17	GNN	$\mathcal{N}^{\text{eni}}$
2	vampire-2	-	$\mathcal{G}_{16}$
2	vampire-16	-	$\mathcal{N}^{\text{au}}$
2	vampire	-	$\mathcal{N}^{\text{au}}$
2	E-auto	-	$\mathcal{L}_{96}$
10	bls05fc	GBDT	$\mathcal{L}_{0.25}$
2	vampire	-	$\mathcal{E}^{\mathcal{N},\mathcal{K}}_{.5,.5,min}$
2	vampire-16	-	$\mathcal{N}^{\text{sub}}$
2	vampire-16	-	$\mathcal{E}^{\mathcal{N},\mathcal{G},\mathcal{K}}_{.5,.25,.25}$

t	*base*	*ENIGMA*	*slice*
2	vampire	-	$\mathcal{E}^{\mathcal{N},\mathcal{G},\mathcal{K}}_{.5,.25,.25}$
2	Blistr-5fce	-	$\mathcal{E}^{\mathcal{N},\mathcal{K},\mathcal{L}}_{.5,.2,.3}$
2	vampire	-	$\mathcal{E}^{\mathcal{N},\mathcal{K}}_{.5,.5}$
2	E-auto	-	$\mathcal{G}_{-2}$
2	vampire	-	$\mathcal{K}^{\text{uni}}_{\text{var}}$
2	vampire-16	-	$\mathcal{E}^{\mathcal{N},\mathcal{K}}_{.5,.5}$
2	vampire	-	$\mathcal{E}^{\mathcal{N},\mathcal{K}}_{.5,.5,geo}$
2	vampire	-	$\mathcal{K}^{\text{uni}}_{\text{var}}$
2	vampire-21	-	$\mathcal{G}$
2	vampire	-	$\mathcal{N}^{\text{sub}}$
2	vampire	-	$\mathcal{N}^{\text{uni}}$
2	E-auto	-	$\mathcal{E}^{\mathcal{N},\mathcal{G},\mathcal{K}}_{.5,.25,.25}$
2	vampire	-	$\mathcal{E}^{\mathcal{N},\mathcal{K}}_{.5,.5,min}$
2	vampire	-	$\mathcal{N}^{\text{cp}}$
2	bls0f17	GNN	$\mathcal{K}^{\text{eni}}_{\text{var}}$
5	bls0f17	GNN	$\mathcal{L}_{0.01}$
10	bls05fc	GBDT	$\mathcal{E}^{\mathcal{N},\mathcal{G},\mathcal{K}}_{.33,.33,.33}$
5	bls0f17	GNN	$\mathcal{E}^{\mathcal{N},\mathcal{K}}_{.5,.5}$
5	bls0f17	GNN	$\mathcal{N}^{\text{eni}}$
5	bls0f17	GNN	$\mathcal{E}^{\mathcal{N},\mathcal{G},\mathcal{K}}_{.5,.25,.25}$
10	bls0f17	GNN	$\mathcal{E}^{\mathcal{N},\mathcal{K},\mathcal{L},\mathcal{G}}_{.5,.2,.2,.1}$
10	mzr03	GBDT	$\mathcal{G}_{64}$
10	bls05fc	GBDT	$\mathcal{E}^{\mathcal{N},\mathcal{G},\mathcal{K}}_{.33,.33,.33}$
10	mzr02	GBDT	$\mathcal{K}^{\text{short}}$
10	bls0f17	GNN	$\mathcal{E}^{\mathcal{N},\mathcal{G},\mathcal{K}}_{.5,.25,.25}$
10	bls0f17	GNN	$\mathcal{E}^{\mathcal{N},\mathcal{K},\mathcal{L},\mathcal{G}}_{.5,.2,.2,.1}$
10	bls0f17	GBDT	$\mathcal{L}_{0.01}$
5	bls0f17	GNN	$\mathcal{N}^{\text{au}}$
5	bls0f17	GNN	$\mathcal{L}_{0.005}$
10	bls0f17	GNN	$\mathcal{E}^{\mathcal{N},\mathcal{K},\mathcal{L},\mathcal{G}}_{.5,.2,.2,.1}$
5	bls0f17	GNN	$\mathcal{L}_{0.01}$
5	bls0f17	GNN	$\mathcal{N}^{\text{au}}$
5	bls0f17	GNN	$\mathcal{E}^{\mathcal{N},\mathcal{K}}_{.5,.5}$
5	mzr03	-	$\mathcal{E}^{\mathcal{N},\mathcal{K},\mathcal{L},\mathcal{G}}_{.25,.25,.25,.25}$
10	bls0f17	GNN	$\mathcal{G}_{-1}$
5	bls0f17	GNN	$\mathcal{G}_{0}$
10	bls05fc	GBDT	$\mathcal{K}^{\text{short}}$
5	bls0f17	GNN	$\mathcal{N}^{\text{au}}$
5	bls0f17	GNN	$\mathcal{E}^{\mathcal{N},\mathcal{K}}_{.5,.5}$
10	bls0f17	GNN	$\mathcal{G}_{0.5}$
10	bls0f17	GNN	$\mathcal{N}^{\text{eni}}$
10	bls0f17	GBDT	$\mathcal{L}_{0.01}$
10	bls0f17	GBDT	$\mathcal{E}^{\mathcal{N},\mathcal{K}}_{.5,.5,\&\text{min}}$
10	bls0f17	GNN	$\mathcal{N}^{\text{au}}$
10	bls0f17	GNN	$\mathcal{E}^{\mathcal{N},\mathcal{K}}_{.5,.5}$
10	bls0f17	GNN	$\mathcal{N}^{\text{au}}$
10	mzr03	GBDT	$\mathcal{E}^{\mathcal{N},\mathcal{G},\mathcal{K}}_{.33,.33,.33}$

arithmetical arguments done in Mizar. An interesting point is that the guided prover is here capable of synthesizing a nontrivial witness $(n! + 1)$ by using the supplied facts, after which the proof likely becomes reasonably straightforward given the knowledge in the library (see the Appendix for a more detailed discussion of this example). In general, using the supplied facts together with the trained learner for guided synthesis of nontrivial witnesses seems to be one of the main improvements brought by the ENIGMA guidance that contributed to the new proofs in comparison with the Mizar40 evaluation. This led us to start research of neural synthesis of witnesses and conjectures for AI/TP settings [13, 16, 17, 58].

Arithmetical reasoning, and other kinds of "routine computation" in general, have turned out to be areas where ENIGMA often gradually improved by solving increasingly hard Mizar problems and learning from them. Such problems include reasoning about trigonometric functions, integrals, derivatives, matrix manipulation, etc. From the more advanced results done by 3-phase ENIGMA, this is, e.g., a 619-long proof of `SINCOS10:86` found in 60 s, doing a lot of computation about the domain and range of arcsec , and a 326-long proof of `FDIFF_8:14` , found in 31 s, about the derivative of $\tan(\ln x)$.

```
for x being set st x in [.(- (sqrt 2)),(- 1).] holds arcsec2 . x in [.((3 / 4) * PI),PI.]

for Z being open Subset of REAL st Z c= dom (tan * ln) holds tan * ln
is_differentiable_on Z
& for x being Real st x in Z holds ((tan * ln) `| Z) . x = 1 / (x * (cos . (ln . x))^2)
```

The first proof uses 83 Mizar facts, starting with 1025 preselected ones. Its proof search took 5344 nontrivial given clauses and generated over 100k nontrivial clauses in total, making the 3-phase filtering and the use of the GPU server essential for finding the proof efficiently. The second proof uses 55 Mizar facts, 3136 given clause loops and it generated 26.6k nontrivial clauses. The reader can see on our web page that there are many solved problems of such "mostly computational" kind, suggesting that such learning approaches may be suitable for automatically gaining competence in routine computational tasks, without the need to manually program them as done, e.g., in SMT solvers. This has motivated our research in learning reasoning components [10]. Two less "computational" but still very long ATP proofs found by 3-phase ENIGMA are `BORSUK_5:31` saying that the closure of rationals on (a,b) is [a,b] , and `IDEAL_1:22` saying that commutative rings are fields iff ideals are trivial :

```
for A being Subset of R^1 for a, b being real number
st a < b & A = RAT (a,b) holds Cl A = [.a,b.]

for R being non degenerated comRing holds R is Field iff
for I being Ideal of R holds I = {(0. R)} or I = the carrier of R
```

The Mizar proof of `BORSUK_5:31` takes 80 lines. ENIGMA finds a proof from 38 Mizar facts that uses 359 clausal steps in 4883 given clause loops. On the 400k generated clauses, the multi-phase ENIGMA mechanisms work as follows. 133 869 clauses are frozen by parental guidance, 83 871 are then filtered by aggressive subsumption, and 64 364 by the first-stage LightGBM model. 125 489 remaining "good" clauses are gradually evaluated (in 176 batched calls) by the GNN server, using a context of 1536 processed clauses. The ENIGMA proof of `IDEAL_1:22` uses 48 Mizar facts and takes 493 clausal steps in 4481 given clause loops.

One example of an ATP proof made possible thanks to the premise selector noticing alternative lemmas in the library is `FIB_NUM2:69` . This theorem, called in the MML "Carmichael's Theorem on Prime Divisors", states that if m divides the n-th Fibonacci number (Fib n), then m does not divide any smaller Fibonacci number, provided m, n are prime numbers . The Mizar proof has 122 lines, uses induction and we cannot so far replay it with ATPs. The premise selector, however, finds a prior library lemma `FIB_NUM:5` saying that `(Fib m) gcd (Fib n) = Fib (m gcd n)`, from which the proof follows, using 159 clausal steps, 4214 given clause loops and 32 Mizar facts. Finally, an example of a long Deepire proof using a high time limit is `ORDINAL5:36` , i.e., the $\epsilon_0 = \omega^{\omega^{\omega^{\cdots}}}$ formula for the zeroth epsilon ordinal :

```
first_epsilon_greater_than 0 = omega |^|^ omega
```

The search took 38 065 given clause loops and 504 s. The proof has 1193 clausal steps, using 49 Mizar facts. Deepire's very efficient neural guidance took only 18 s of the total time here.

8 Conclusion: AI/TP Bet Completed

In 2014, after the 40 % numbers were obtained by Kaliszyk and Urban both on the Flyspeck and Mizar corpora, the last author publicly announced three AI/TP bets[11] in a talk at Institut Henri Poincare and offered to bet up to 10 000 EUR on them. Part of the second bet said that by 2024, 60 % of the MML and Flyspeck toplevel theorems will be provable automatically when using the same setting as in 2014. In the HOL setting, this was done as early as 2017/18 by the TacticToe system, which achieved 66.4 % on the HOL library in 60 s and 69 % in 120 s [14, 15]. One could however argue that TacticToe introduced a new kind of ML-guided tactical prover that considerably benefits from targeted, expert-written procedures tailored to the corpora. This in particular showed in the large boost on HOL problems that required induction, on which standard higher-order ATPs traditionally struggled.

In this work, we largely completed this part of the second AI/TP bet also for the Mizar library. The main caveat is our use of more modern hardware, in particular many ENIGMAs using the GPU server for clause evaluation. It is however clear (both from the LightGBM experiments and from the very efficient and CPU-based Deepire experiments) that this is not a major issue. While it is today typically easier to use dedicated hardware in ML-based experiments, there is also growing research in the extraction of faster predictors from those trained on GPUs that can run more efficiently on standard hardware.

References

1 Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. URL: `https://www.tensorflow.org/`.

2 Jesse Alama, Tom Heskes, Daniel Kühlwein, Evgeni Tsivtsivadze, and Josef Urban. Premise selection for mathematics by corpus analysis and kernel methods. *J. Autom. Reasoning*, 52(2):191–213, 2014. `doi:10.1007/s10817-013-9286-5`.

3 Grzegorz Bancerek, Czeslaw Bylinski, Adam Grabowski, Artur Kornilowicz, Roman Matuszewski, Adam Naumowicz, and Karol Pak. The role of the Mizar Mathematical Library for interactive proof development in Mizar. *J. Autom. Reason.*, 61(1-4):9–32, 2018. `doi:10.1007/s10817-017-9440-6`.

4 Grzegorz Bancerek, Czeslaw Bylinski, Adam Grabowski, Artur Kornilowicz, Roman Matuszewski, Adam Naumowicz, Karol Pak, 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, Washington, DC, USA, July 13-17, 2015, Proceedings*, volume 9150 of *Lecture Notes in Computer Science*, pages 261–279. Springer, 2015. `doi:10.1007/978-3-319-20615-8_17`.

5 Grzegorz Bancerek and Piotr Rudnicki. A Compendium of Continuous Lattices in MIZAR. *J. Autom. Reasoning*, 29(3-4):189–224, 2002. `doi:10.1023/A:1021966832558`.

[11] `http://ai4reason.org/aichallenges.html`

6 Jasmin Christian Blanchette, David Greenaway, Cezary Kaliszyk, Daniel Kühlwein, and Josef Urban. A learning-based fact selector for Isabelle/HOL. *J. Autom. Reasoning*, 57(3):219–244, 2016. doi:10.1007/s10817-016-9362-8.

7 Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *J. Formalized Reasoning*, 9(1):101–148, 2016. doi:10.6092/issn.1972-5787/4593.

8 Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM. doi:10.1145/2939672.2939785.

9 Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1724–1734. ACL, 2014. doi:10.3115/v1/d14-1179.

10 Karel Chvalovský, Jan Jakubuv, Miroslav Olsák, and Josef Urban. Learning theorem proving components. In Anupam Das and Sara Negri, editors, *Automated Reasoning with Analytic Tableaux and Related Methods - 30th International Conference, TABLEAUX 2021, Birmingham, UK, September 6-9, 2021, Proceedings*, volume 12842 of *Lecture Notes in Computer Science*, pages 266–278. Springer, 2021. doi:10.1007/978-3-030-86059-2_16.

11 Karel Chvalovský, Jan Jakubuv, Martin Suda, and Josef Urban. ENIGMA-NG: Efficient neural and gradient-boosted inference guidance for E. 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 197–215. Springer, 2019. doi:10.1007/978-3-030-29436-6_12.

12 Richard Evans, David Saxton, David Amos, Pushmeet Kohli, and Edward Grefenstette. Can neural networks understand logical entailment? In *International Conference on Learning Representations*, 2018. URL: https://openreview.net/forum?id=SkZxCk-0Z.

13 Thibault Gauthier. Deep reinforcement learning for synthesizing functions in higher-order logic. In *LPAR*, volume 73 of *EPiC Series in Computing*, pages 230–248. EasyChair, 2020.

14 Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. TacticToe: Learning to reason with HOL4 tactics. In Thomas Eiter and David Sands, editors, *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, volume 46 of *EPiC*, pages 125–143. EasyChair, 2017. doi:10.29007/ntlb.

15 Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. Tactictoe: Learning to prove with tactics. *J. Autom. Reason.*, 65(2):257–286, 2021. doi:10.1007/s10817-020-09580-x.

16 Thibault Gauthier, Miroslav Olsák, and Josef Urban. Alien coding. *CoRR*, abs/2301.11479, 2023.

17 Thibault Gauthier and Josef Urban. Learning program synthesis for integer sequences from scratch. *CoRR*, abs/2202.11908, 2022.

18 Zarathustra Goertzel, Jan Jakubův, and Josef Urban. ENIGMAWatch: ProofWatch meets ENIGMA. In Serenella Cerrito and Andrei Popescu, editors, *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 374–388, Cham, 2019. Springer International Publishing.

19 Zarathustra Amadeus Goertzel. Make E smart again (short paper). In *IJCAR (2)*, volume 12167 of *Lecture Notes in Computer Science*, pages 408–415. Springer, 2020.

20 Zarathustra Amadeus Goertzel, Karel Chvalovský, Jan Jakubuv, Miroslav Olsák, and Josef Urban. Fast and slow Enigmas and parental guidance. In *FroCoS*, volume 12941 of *Lecture Notes in Computer Science*, pages 173–191. Springer, 2021.

21 Zarathustra Amadeus Goertzel, Jan Jakubuv, Cezary Kaliszyk, Miroslav Olsák, Jelle Piepenbrock, and Josef Urban. The Isabelle ENIGMA. In *ITP*, volume 237 of *LIPIcs*, pages 16:1–16:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

22 Adam Grabowski, Artur Korniłowicz, and Adam Naumowicz. Mizar in a nutshell. *J. Formalized Reasoning*, 3(2):153–245, 2010.

23 Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: an automatic algorithm configuration framework. *J. Artificial Intelligence Research*, 36:267–306, October 2009.

24 Jan Jakubův and Josef Urban. Hierarchical invention of theorem proving strategies. *AI Commun.*, 31(3):237–250, 2018. doi:10.3233/AIC-180761.

25 Jan Jakubuv, Karel Chvalovský, Miroslav Olsák, Bartosz Piotrowski, Martin Suda, and Josef Urban. ENIGMA anonymous: Symbol-independent inference guiding machine (system description). In *IJCAR (2)*, volume 12167 of *Lecture Notes in Computer Science*, pages 448–463. Springer, 2020.

26 Jan Jakubuv, Martin Suda, and Josef Urban. Automated invention of strategies and term orderings for vampire. In *GCAI*, volume 50 of *EPiC Series in Computing*, pages 121–133. EasyChair, 2017.

27 Jan Jakubuv and Josef Urban. BliStrTune: hierarchical invention of theorem proving strategies. 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 43–52. ACM, 2017. doi:10.1145/3018610.3018619.

28 Jan Jakubuv and Josef Urban. Enhancing ENIGMA given clause guidance. In Florian Rabe, William M. Farmer, Grant O. Passmore, and Abdou Youssef, editors, *Intelligent Computer Mathematics - 11th International Conference, CICM 2018, Hagenberg, Austria, August 13-17, 2018, Proceedings*, volume 11006 of *Lecture Notes in Computer Science*, pages 118–124. Springer, 2018. doi:10.1007/978-3-319-96812-4_11.

29 Jan Jakubuv and Josef Urban. Hammering Mizar by learning clause guidance. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*, pages 34:1–34:8. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ITP.2019.34.

30 Cezary Kaliszyk and Josef Urban. Stronger automation for Flyspeck by feature weighting and strategy evolution. In Jasmin Christian Blanchette and Josef Urban, editors, *PxTP 2013*, volume 14 of *EPiC Series*, pages 87–95. EasyChair, 2013.

31 Cezary Kaliszyk and Josef Urban. Learning-assisted automated reasoning with Flyspeck. *J. Autom. Reasoning*, 53(2):173–213, 2014. doi:10.1007/s10817-014-9303-3.

32 Cezary Kaliszyk and Josef Urban. MizAR 40 for Mizar 40. *J. Autom. Reasoning*, 55(3):245–256, 2015. doi:10.1007/s10817-015-9330-8.

33 Cezary Kaliszyk, Josef Urban, and Jirí Vyskocil. Efficient semantic features for automated reasoning over large theories. In *IJCAI*, pages 3084–3090. AAAI Press, 2015.

34 Cezary Kaliszyk, Josef Urban, and Jirí Vyskocil. Automating formalization by statistical and semantic parsing of mathematics. In *ITP*, volume 10499 of *Lecture Notes in Computer Science*, pages 12–27. Springer, 2017.

35 Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *NIPS*, pages 3146–3154, 2017.

36 Artur Korniłowicz and Christoph Schwarzweller. Computers and algorithms in Mizar. *Mechanized Mathematics and Its Applications*, 4(1):43–50, 2005.

37 Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *CAV*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013. doi:10.1007/978-3-642-39799-8_1.

38 Guillaume Lample and François Charton. Deep learning for symbolic mathematics. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL: https://openreview.net/forum?id=S1eZYeHFDS.

39 Miroslav Olsák, Cezary Kaliszyk, and Josef Urban. Property invariant embedding for automated reasoning. In Giuseppe De Giacomo, Alejandro Catalá, Bistra Dilkina, Michela Milano, Senén Barro, Alberto Bugarín, and Jérôme Lang, editors, *ECAI 2020 - 24th European Conference on Artificial Intelligence*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 1395–1402. IOS Press, 2020. doi:10.3233/FAIA200244.

40 David W. Opitz and Richard Maclin. Popular ensemble methods: An empirical study. *J. Artif. Intell. Res.*, 11:169–198, 1999. doi:10.1613/jair.614.

41 Bartosz Piotrowski and Josef Urban. ATPboost: Learning premise selection in binary setting with ATP feedback. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*, volume 10900 of *Lecture Notes in Computer Science*, pages 566–574. Springer, 2018. doi:10.1007/978-3-319-94205-6_37.

42 Bartosz Piotrowski and Josef Urban. Stateful premise selection by recurrent neural networks. In *LPAR*, volume 73 of *EPiC Series in Computing*, pages 409–422. EasyChair, 2020.

43 Bartosz Piotrowski, Josef Urban, Chad E. Brown, and Cezary Kaliszyk. Can neural networks learn symbolic rewriting? *CoRR*, abs/1911.04873, 2019.

44 John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001. URL: https://www.sciencedirect.com/book/9780444508133/handbook-of-automated-reasoning.

45 Stephan Schulz. E – A Brainiac Theorem Prover. *AI Commun.*, 15(2-3):111–126, 2002. URL: http://iospress.metapress.com/content/n908n94nmvk59v3c/.

46 Stephan Schulz. Simple and efficient clause subsumption with feature vector indexing. In *Automated Reasoning and Mathematics*, volume 7788 of *Lecture Notes in Computer Science*, pages 45–67. Springer, 2013.

47 Stephan Schulz. System description: E 1.8. In Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *LPAR*, volume 8312 of *LNCS*, pages 735–743. Springer, 2013. doi:10.1007/978-3-642-45221-5_49.

48 Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*, pages 735–743. Springer, 2013.

49 Stephan Schulz, Simon Cruanes, and Petar Vukmirović. Faster, higher, stronger: E 2.3. In Pascal Fontaine, editor, *Proc. of the 27th CADE, Natal, Brasil*, number 11716 in LNAI, pages 495–507. Springer, 2019.

50 Stephan Schulz and Geoff Sutcliffe. Proof generation for saturating first-order theorem provers. In David Delahaye and Bruno Woltzenlogel Paleo, editors, *All about Proofs, Proofs for All*, volume 55 of *Mathematical Logic and Foundations*, pages 45–61. College Publications, London, UK, January 2015.

51 Martin Suda. Improving ENIGMA-style clause selection while learning from history. 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 543–561. Springer, 2021. doi:10.1007/978-3-030-79876-5_31.

52 Martin Suda. Vampire with a brain is a good ITP hammer. In Boris Konev and Giles Reger, editors, *Frontiers of Combining Systems - 13th International Symposium, FroCoS 2021, Birmingham, UK, September 8-10, 2021, Proceedings*, volume 12941 of *Lecture Notes in Computer Science*, pages 192–209. Springer, 2021. doi:10.1007/978-3-030-86205-3_11.

53 Tanel Tammet. Towards efficient subsumption. In *CADE*, volume 1421 of *Lecture Notes in Computer Science*, pages 427–441. Springer, 1998.

54 J. Urban. Translating Mizar for First Order Theorem Provers. In A. Asperti, B. Buchberger, and J.H. Davenport, editors, *Proceedings of the 2nd International Conference on Mathematical Knowledge Management*, number 2594 in LNCS, pages 203–215. Springer, 2003.

55 Josef Urban. MPTP – Motivation, Implementation, First Experiments. *J. Autom. Reasoning*, 33(3-4):319–339, 2004. doi:10.1007/s10817-004-6245-1.

56 Josef Urban. MPTP 0.2: Design, implementation, and initial experiments. *J. Autom. Reasoning*, 37(1-2):21–43, 2006. doi:10.1007/s10817-006-9032-3.

57 Josef Urban. BliStr: The Blind Strategymaker. In Georg Gottlob, Geoff Sutcliffe, and Andrei Voronkov, editors, *Global Conference on Artificial Intelligence, GCAI 2015, Tbilisi, Georgia, October 16-19, 2015*, volume 36 of *EPiC Series in Computing*, pages 312–319. EasyChair, 2015. URL: http://www.easychair.org/publications/paper/BliStr_The_Blind_Strategymaker, doi:10.29007/8n7m.

58 Josef Urban and Jan Jakubuv. First neural conjecturing datasets and experiments. In Christoph Benzmüller and Bruce R. Miller, editors, *Intelligent Computer Mathematics - 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings*, volume 12236 of *Lecture Notes in Computer Science*, pages 315–323. Springer, 2020. doi:10.1007/978-3-030-53518-6_24.

59 Josef Urban, Geoff Sutcliffe, Petr Pudlák, and Jiří Vyskočil. MaLARea SG1 – Machine Learner for Automated Reasoning with Semantic Guidance. In *IJCAR*, pages 441–456, 2008. doi:10.1007/978-3-540-71070-7_37.

60 Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. arXiv:1706.03762.

61 Qingxiang Wang, Cezary Kaliszyk, and Josef Urban. First experiments with neural translation of informal to formal mathematics. In Florian Rabe, William M. Farmer, Grant O. Passmore, and Abdou Youssef, editors, *11th International Conference on Intelligent Computer Mathematics (CICM 2018)*, volume 11006 of *LNCS*, pages 255–270. Springer, 2018. doi:10.1007/978-3-319-96812-4_22.

Constructive Final Semantics of Finite Bags

Philipp Joram
Department of Software Science, Tallinn University of Technology, Estonia

Niccolò Veltri
Department of Software Science, Tallinn University of Technology, Estonia

Abstract

Finitely-branching and unlabelled dynamical systems are typically modelled as coalgebras for the finite powerset functor. If states are reachable in multiple ways, coalgebras for the finite bag functor provide a more faithful representation. The final coalgebra of this functor is employed as a denotational domain for the evaluation of such systems. Elements of the final coalgebra are non-wellfounded trees with finite unordered branching, representing the evolution of systems starting from a given initial state.

This paper is dedicated to the construction of the final coalgebra of the finite bag functor in homotopy type theory (HoTT). We first compare various equivalent definitions of finite bags employing higher inductive types, both as sets and as groupoids (in the sense of HoTT). We then analyze a few well-known, classical set-theoretic constructions of final coalgebras in our constructive setting. We show that, in the case of set-based definitions of finite bags, some constructions are intrinsically classical, in the sense that they are equivalent to some weak form of excluded middle. Nevertheless, a type satisfying the universal property of the final coalgebra can be constructed in HoTT employing the groupoid-based definition of finite bags. We conclude by discussing generalizations of our constructions to the wider class of analytic functors.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Constructive mathematics

Keywords and phrases finite bags, final coalgebra, homotopy type theory, Cubical Agda

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.20

Supplementary Material *Software (Agda Code)*: `github.com/phijor/agda-cubical-multiset` archived at `swh:1:snp:3c33a341583333a888a148d4a91c08b94e404482`

Funding This work was supported by the Estonian Research Council grant PSG749 and the ESF funded Estonian IT Academy research measure (project 2014-2020.4.05.19-0001).

1 Introduction

Coalgebras are functions of the form $c : S \to FS$, where S is a set of states and F is a functor specifying a certain class of collections of states [23, 14]. For example, FS could be lists over S, bags (*i.e.* multisubsets) or subsets of S (possibly with some cardinality restrictions), wellfounded trees with leaves or nodes in S, or probability distributions over S. The coalgebra c describes the dynamics of a transition system or an automaton: to each state $s : S$, the function c associates the collection of states $c\,s : FS$ that are reachable from s in one step. The choice of collection functor F is dictated by the specific flavor of non-determinism that is specified by the transition relation. Does the order or multiplicity of reachable states matter? Is the choice of a new state probabilistic? Does the transition relation additionally depend on a set of labels, weights or actions?

The denotational semantics of a transition system $c : S \to FS$ is typically given in terms of the *final coalgebra* L_F of the functor F, which consists of non-wellfounded trees with branching specified by F. When F is the list functor, each tree has a finite and ordered collection of subtrees. If F is the finite bag functor, the order of subtrees does not matter,


© Philipp Joram and Niccolò Veltri;
licensed under Creative Commons License CC-BY 4.0

and if F is the finite powerset functor, multiplicity of subtrees does not matter either. The interpretation of a state $s : S$ in L_F is the possibly-infinite tree obtained by "running" the coalgebra c with s as initial state. As such, it gives a complete description of the evolution of the system c starting from state s.

The theory of dynamical systems as coalgebras [23, 14], and in particular the formal description of final coalgebras [4, 2, 32], with the associated notion of bisimilarity and behavioural equivalence of states, is traditionally developed in a set-theoretic framework with reasoning based on classical logic. In this work, we propose to study the theory of coalgebras in a framework based on constructive logic, more specifically in homotopy type theory (HoTT) [26]. The use of a constructive metatheory is beneficial for the development of *formal denotational semantics* of dynamical systems and programming languages, often centered on the notions of final coalgebra and bisimilarity [27], in proof assistants based on variants of Martin-Löf type theory, such as Agda, Coq, Idris and Lean. The specific choice of HoTT is motivated by its expressiveness and innovative features, higher inductive types (HITs) and the univalence principle, which are crucial ingredients for faithfully representing a variety of collection functors F and reasoning up to equivalent presentations of F.

Specific constructions of final coalgebras for a selection of functors, performed *internally* in HoTT, already exist in the literature. Ahrens *et al.* [3] presented a construction of *M-types*, *i.e.* final coalgebras of polynomial functors. They show that, for a polynomial functor F, the traditional set-theoretic construction of its M-type as the ω-limit of the chain

$$1 \xleftarrow{!} F1 \xleftarrow{F(!)} F^2 1 \xleftarrow{F^2(!)} F^3 1 \xleftarrow{F^3(!)} \cdots \tag{1}$$

(with 1 being the unit type and ! the unique map into 1) can be ported without major complications to the setting of HoTT. Veltri [29] examined various constructions of the final coalgebra of the finite powerset functor, which is known to not be definable as an ω-limit [2]. Worrell proposed a set-theoretic construction as a $(\omega + \omega)$-limit [32], but Veltri showed that this cannot be ported to the constructive setting of HoTT: Worrell's construction defines the final coalgebra of the finite powerset functor if and only if the *lesser limited principle of omniscience* (LLPO) holds, which is a constructive taboo [6].

We extend this line of work by studying the final coalgebra of the *finite bag functor*. This is an intermediate situation between finitary polynomial functors, such as the one delivering lists, and general finitary functors, such as the one delivering finite subsets. It also serves as a representative starting point for a constructive analysis of Joyal's *analytic functors* [15] and their final semantics. In type theory, analytic functors arise from *quotient containers* [1] and encompass many datatypes with symmetries, such as finite bags, unordered pairs and cyclic lists [34, 33].

Following the recent work of Choudhury and Fiore [8], we define and compare various implementations of the type of finite bags in HoTT. Choudhury and Fiore give two equivalent presentations of finite bags as HITs: as free commutative monoids and as lists modulo swapping of adjacent entries. We add an equivalent presentation of finite bags as an analytic functor FMSet: an element of $\mathsf{FMSet}\, X$ is a pair of a natural number n (its size) and an equivalence class of functions typed $\mathsf{Fin}\, n \to X$ picking an element of X for each $k < n$. Two functions $v, w : \mathsf{Fin}\, n \to X$ belong to the same equivalence class if there merely exists an equivalence $\sigma : \mathsf{Fin}\, n \to \mathsf{Fin}\, n$ such that $v = w \circ \sigma$. The type $\mathsf{FMSet}\, X$ is always a set (in the sense of HoTT, *i.e.* a type with at most one identification between any two terms), since it employs set-quotienting. Similarly, the HITs of Choudhury and Fiore are sets. Following [16], finite bags can alternatively be defined as a polynomial functor Bag returning a groupoid (in the sense of HoTT, *i.e.* a type whose equality types are sets) instead of a set. In this case,

an element of $\mathsf{Bag}\,X$ is a pair consisting of a (Bishop-)finite type Y and a function from Y to X. The set-based and the groupoid-based definitions of bags are appropriately related by the set-truncation construction: $\|\mathsf{Bag}\,X\|_2 \simeq \mathsf{FMSet}\,X$.

We investigate 3 constructions of the final coalgebra of the finite bag functor:

1. Working with the set-based functor FMSet, we try to replicate the classical set-theoretic construction as an ω-limit of (1) in our constructive setting. We show that this cannot be directly performed in HoTT without introducing some form of classical logic, an issue already spotted in the case of the finite powerset functor [29]. Formally, we show that FMSet weakly preserves the ω-limit of (1), but strong preservation of this limit implies LLPO.
2. The list functor admits a final coalgebra $\mathsf{L}_{\mathsf{List}}$ in HoTT [3] and classically an appropriate quotient of the latter, by a relation Bisim identifying non-wellfounded trees which differ in the order of their subtrees, delivers the final coalgebra of the finite bag functor. This construction is also inherently classical: attempting to define a FMSet-coalgebra structure on $\mathsf{L}_{\mathsf{List}} /_2 \mathsf{Bisim}$, *i.e.* a function of type $\mathsf{L}_{\mathsf{List}} /_2 \mathsf{Bisim} \to \mathsf{FMSet}(\mathsf{L}_{\mathsf{List}} /_2 \mathsf{Bisim})$, by directly lifting the List-coalgebra structure of $\mathsf{L}_{\mathsf{List}}$, implies LLPO. We point out that this issue already appears in the category of *setoids* [5], before effectively forming the set-quotient $\mathsf{L}_{\mathsf{List}} /_2 \mathsf{Bisim}$. We were able to prove that $\mathsf{L}_{\mathsf{List}} /_2 \mathsf{Bisim}$ is the final FMSet-coalgebra only under the assumption of the axiom of choice.
3. The groupoid-based polynomial functor Bag admits a final coalgebra $\mathsf{L}_{\mathsf{Bag}}$ as the ω-limit of (1), a result arising directly from the work of Ahrens *et al.* [3]. $\mathsf{L}_{\mathsf{Bag}}$ is a groupoid, not a set. One might wonder if the set-truncation $\|\mathsf{L}_{\mathsf{Bag}}\|_2$ is a good candidate for the final FMSet-coalgebra. We show that it is a fixpoint of FMSet, but we were able to prove that it is the final coalgebra only under the assumption of two variants of the axiom of choice.

We do not yet know whether the uses of choice in the last two constructions are also necessary. Nevertheless, the set-truncation $\|\mathsf{L}_{\mathsf{Bag}}\|_2$ can be practically employed as denotational domain for transition systems with a *(Bishop-)finite* set of states: given a coalgebra $c : S \to \mathsf{FMSet}\,S$ where S is finite, there exists a unique coalgebra morphism from S to $\|\mathsf{L}_{\mathsf{Bag}}\|_2$, and no additional choice principle needs to be assumed in this case.

We conclude by discussing generalizations of our constructions to other analytic functors.

The material presented in the paper (apart from Section 7) has been formalized in the Cubical Agda proof assistant, building on top of the `agda/cubical` library [25]. The code is freely available at `https://github.com/phijor/agda-cubical-multiset`. For any result in the paper decorated with an `Identifier`, the repository contains instructions on how to find the corresponding formalization.

2 Type Theory and Cubical Agda

We work in homotopy type theory [26] and practically our formalization takes place in Cubical Agda [30]. We recall some basic notions that are employed in our development.

Given a type A and a type family B on A, the associated dependent function type is $(x : A) \to B\,x$, written also $\forall x.\, B\,x$ when the type A is clear from context. Implicit arguments of dependent functions are enclosed in curly brackets. Basic inductive types include: unit 1, empty $\bot$, naturals $\mathbb{N}$, finite prefixes of naturals $\mathsf{Fin}\,n$, lists $\mathsf{List}\,A$, dependent pair $\sum(x : A).\, B\,x$, binary sum $A + B$. We use standard names for their constructors. The unique function from a type A into the unit type is $! : A \to 1$. Given an inductive type T, we write elim_T and rec_T for its dependent and non-dependent elimination principles, respectively (we employ the same notation also for higher inductive types). The action on

maps of a functor $F : \mathsf{Type} \to \mathsf{Type}$ is map_F; to avoid ambiguities, we write $\mathsf{map}_F\, f$ over the conventional $F(f)$. Most of our constructions are universe-polymorphic, but for the sake of readability in the paper we use only the two lowest universe of types, Type and Type_1.

Given $x, y : A$, their definitional equality is denoted $x =_{\mathsf{df}} y$ while propositional equality is $x = y$. Following "cubical terminology", the latter is called the *path type* between x and y. In Cubical Agda, the path type $x = y$ behaves similarly to a function type $\mathbb{I} \to A$, where $\mathbb{I}$ is a primitive interval type with endpoints i_0 and i_1. An element $p : x = y$ is eliminated by application to an interval name $r : \mathbb{I}$, returning $p\, r : A$. But unlike function types, this application can compute even when p is unknown by using the endpoints x and y: $p\, i_0$ reduces to x and $p\, i_1$ reduces to y. Path introduction is lambda abstraction $(\lambda i : \mathbb{I}.\, t) : x = y$, but it causes the extra requirement to match the endpoints: $t[i_0/i]$ is judgementally equal to x and $t[i_1/i]$ is judgementally equal to y. We write $\mathsf{refl}\ x$ for the constant path (*i.e.* proof of reflexivity) in $x = x$ and $(\bullet)$ for sequential composition of paths.

A function $f : A \to B$ is an *equivalence* if it has contractible fibers, *i.e.* if the preimage of any element in B under f is a singleton type. Any function underlying a type isomorphism defines an equivalence. Writing $A \simeq B$ for the type of equivalences between A and B, Voevodsky's *univalence principle* states that the canonical function of type $A = B \to A \simeq B$ is an equivalence. This is a theorem in Cubical Agda. In particular, there is a function $\mathsf{ua} : A \simeq B \to A = B$ turning equivalences into path equalities. Univalence implies *function extensionality*: pointwise equal functions are equal. We recall the first instances of the hierarchy of *homotopy levels*,[1] and say that a type A is:

($n = 1$) a *proposition*, if $\mathsf{isProp}\, A =_{\mathsf{df}} (a\, b : A) \to a = b$ is inhabited,

($n = 2$) a *set*, if $\mathsf{isSet}\, A =_{\mathsf{df}} (a\, b : A) \to \mathsf{isProp}\ (a = b)$ is inhabited,

($n = 3$) a *groupoid*, if $\mathsf{isGroupoid}\, A =_{\mathsf{df}} (a\, b : A) \to \mathsf{isSet}\ (a = b)$ is inhabited.

When mentioning "sets" or "groupoids", we always refer to the definitions above.

A *higher inductive type* (HIT) is like an inductive type, but its constructors can build both its elements and its (higher) paths. HITs are primitively supported in Cubical Agda. We recall the definition of three basic HITs: propositional truncation, set-truncation and set-quotient.

The *propositional truncation* $\|A\|_1$ is the proposition associated to the type A, *i.e.* it identifies all the elements and (higher) paths of A. It is the HIT with constructors

$$\frac{a : A}{|a|_1 : \|A\|_1} \qquad \frac{x, y : \|A\|_1}{\mathsf{squash}_1\ x\ y : x = y}$$

We define the *existential quantifier* $\exists(x : A).\, B\, x =_{\mathsf{df}} \|\sum(x : A).\, B\, x\|_1$, which records the mere existence of an element x satisfying B.

The *set-truncation* $\|A\|_2$ is the set associated to the type A, *i.e.* it identifies all (higher) paths of A. It is the HIT with constructors

$$\frac{a : A}{|a|_2 : \|A\|_2} \qquad \frac{x, y : \|A\|_2 \qquad p, q : x = y}{\mathsf{squash}_2\ p\ q : p = q}$$

The *set-quotient* $A /_2 R$ of a type A by a (possibly proof-relevant) relation $R : A \to A \to \mathsf{Type}$ is the HIT with constructors

$$\frac{a : A}{[a]_2 : A /_2 R} \qquad \frac{a, b : A \qquad r : R\ a\ b}{\mathsf{eq}/_2\ r : [a]_2 = [b]_2} \qquad \frac{x, y : A /_2 R \qquad p, q : x = y}{\mathsf{squash}/_2\ p\ q : p = q}$$

[1] To stay close to the formalization, we follow Voevodsky's [31] 0-based numbering of h-levels.

The term $[a]_2$ is the R-equivalence class of a, while the path constructor $\mathsf{eq}/_2$ states that R-related elements have path equal equivalence classes. The higher path constructor $\mathsf{squash}/_2$ forces $A /_2 R$ to be a set.

Other HITs are presented in the next section, where we also take a closer look at their elimination principles.

3 The Finite Bag Functor in Sets

The action of the finite bag functor on a type X can be encoded as a higher inductive type in various ways, three of which are presented here. The first is the algebraic presentation of the free commutative monoid, the second as lists modulo permutations, the third as an analytic functor. These are all set-based definitions, in the sense that the type of finite bags is a set. In Section 3.4 we prove these are naturally equivalent as types, therefore being equivalent as functors. Groupoid-based definitions are discussed in Section 5.

3.1 As the Free Commutative Monoid

Given a type X, the *free commutative monoid* on X [8] is the HIT induced by the following rules:

$$\frac{}{\varepsilon : \mathsf{FCM}\, X} \qquad \frac{x : X}{\eta\, x : \mathsf{FCM}\, X} \qquad \frac{xs, ys : \mathsf{FCM}\, X}{xs \oplus ys : \mathsf{FCM}\, X}$$

$$\frac{xs : \mathsf{FCM}\, X}{\mathsf{unit} : \varepsilon \oplus xs = xs} \qquad \frac{xs, ys, zs : \mathsf{FCM}\, X}{\mathsf{assoc} : xs \oplus (ys \oplus zs) = (xs \oplus ys) \oplus zs}$$

$$\frac{xs, ys : \mathsf{FCM}\, X}{\mathsf{comm} : xs \oplus ys = ys \oplus xs} \qquad \frac{xs, ys : \mathsf{FCM}\, X \qquad p, q : xs = ys}{\mathsf{squash}_{\mathsf{FCM}}\, p\, q : p = q}$$

The constructor η embeds X into $\mathsf{FCM}\, X$, while ε and $\oplus$ are the unit and multiplication of the monoid. The path constructors express unitality of ε with respect to $\oplus$, associativity and commutativity of $\oplus$, and the final higher path constructor forces $\mathsf{FCM}\, X$ to be a set.

In Cubical Agda, functions out of HITs like $\mathsf{FCM}\, X$ can be defined directly by pattern matching. But it is often useful to have elimination principles at hand that give more control on the shape of the proof obligations. For example, the non-dependent elimination principle of $\mathsf{FCM}\, X$ states that a function of type $\mathsf{FCM}\, X \to A$ is definable, provided that A is a commutative monoid and there exists a function $\eta^* : X \to A$.

$$\begin{aligned}
&\mathsf{rec}_{\mathsf{FCM}\, X} : \{A : \mathsf{Type}\} \to \mathsf{isSet}\, A \\
&\qquad \to (\varepsilon^* : A)\, (\eta^* : X \to A)\, ((+) : A \to A \to A) \\
&\qquad \to (\forall a.\ \varepsilon^* + a = a) \\
&\qquad \to (\forall a\, b\, c.\ a + (b + c) = (a + b) + c) \\
&\qquad \to (\forall a\, b.\ a + b = b + a) \\
&\qquad \to \mathsf{FCM}\, X \to A
\end{aligned}$$

FCM is a functor, with action on maps given by

$$\begin{aligned}
&\mathsf{map}_{\mathsf{FCM}} : (f : X \to Y) \to \mathsf{FCM}\, X \to \mathsf{FCM}\, Y \\
&\mathsf{map}_{\mathsf{FCM}}\ f =_{\mathsf{df}} \mathsf{rec}_{\mathsf{FCM}\, X}\ \ \mathsf{squash}_{\mathsf{FCM}}\ \varepsilon\ (\eta \circ f)\ (\oplus)\ \mathsf{unit}\ \mathsf{assoc}\ \mathsf{comm}
\end{aligned}$$

3.2 As a Quotient of Lists

Another standard definition of the type of finite bags is as lists modulo permutations. The relation specifying the existence of a permutation between two lists can be given in multiple ways, here we mention two possibilities.

Given $xs, ys : \mathsf{List}\, X$, the relation $\mathsf{Perm}\, xs\, ys$ is generated by the rules:

$$\frac{}{\mathsf{Perm}\, xs\, xs} \qquad \frac{\mathsf{Perm}\, (xs +\!\!+ x :: y :: ys)\, zs}{\mathsf{Perm}\, (xs +\!\!+ y :: x :: ys)\, zs}$$

In other words, Perm is the reflexive-transitive closure of the relation generated by pairs of lists of the form $xs +\!\!+ x :: y :: ys$ and $xs +\!\!+ y :: x :: ys$. This is a very "intensional" way of representing permutations of lists: a proof of $\mathsf{Perm}\, xs\, ys$ not only records where each entry in xs is moved to in ys, but also how it is moved there. As such, $\mathsf{Perm}\, xs\, ys$ is generally not a proposition.

Another way of specifying permutations is via a *relation lifting*, often called a *relator* [19]. Given a relation R on a type X, we inductively define a relation $\mathsf{DRelator}\, R$ on $\mathsf{List}\, X$, which intuitively states that each occurrence of an element x in the first list is R-related to the occurrence of an element y in the second list. The type of occurrences $x \in xs$ is generated by

$$\frac{x : X \qquad xs : \mathsf{List}\, X}{x \in x :: xs} \qquad \frac{x\, y : X \qquad xs : \mathsf{List}\, X \qquad m : x \in xs}{x \in y :: xs}$$

Removal $xs \setminus m : \mathsf{List}\, X$ of an occurrence $m : x \in xs$ is defined by induction on m. The *directed relation lifting* of R is the relation generated by rules

$$\frac{}{\mathsf{DRelator}\, R\, [\,]\, ys} \qquad \frac{\exists(y : Y).\ \sum(m : y \in ys).\ R\, x\, y \times \mathsf{DRelator}\ R\, xs\, (ys \setminus m)}{\mathsf{DRelator}\, R\, (x :: xs)\ ys}$$

and we take the relation lifting of R to be the *symmetrization* of the relation $\mathsf{DRelator}\, R$, *i.e.* $\mathsf{Relator}\, R\ xs\ ys =_{\mathsf{df}} \mathsf{DRelator}\, R\ xs\ ys \times \mathsf{DRelator}\, R\ ys\ xs$. Because of the presence of a propositional truncation in the premise of the 2nd rule, both $\mathsf{DRelator}\, R$ and $\mathsf{Relator}\, R$ are propositionally-valued. If R is reflexive and transitive, then $\mathsf{Relator}\, R$ is an equivalence relation.

When R is path equality on X, the type $\mathsf{Relator}\, (=)\ xs\ ys$ expresses the mere existence of a permutation connecting xs and ys. In fact, $\|\mathsf{Perm}\ xs\ ys\|_1$ and $\mathsf{Relator}\, (=)\ xs\ ys$ are equivalent types.

3.3 As an Analytic Functor

We additionally introduce the type of finite bags over X as an *analytic functor* (in the formulation of Hasegawa [12]). For any type X, we define a type of *finite multisets* [20, 8]

$$\mathsf{FMSet}\, X =_{\mathsf{df}} \sum(n : \mathbb{N}).\ (\mathsf{Fin}\, n \to X)\,/_2\, \mathsf{SymAct}\, n$$

where $\mathsf{SymAct}\, n$ is the propositionally-valued relation

$$\mathsf{SymAct}\, n\ v\ w =_{\mathsf{df}} \exists(\sigma : \mathsf{Fin}\, n \simeq \mathsf{Fin}\, n).\ v = w \circ \sigma$$

In other words, an element of $\mathsf{FMSet}\, X$ is a pair of a natural number n (the size of the set) and an equivalence class of functions $v : \mathsf{Fin}\, n \to X$ picking an element in X for each index $k < n$. The relation $\mathsf{SymAct}\, n$ is the action of the symmetric group $\mathsf{Fin}\, n \simeq \mathsf{Fin}\, n$ on n-tuples of elements of X. We write $\mathsf{SymAct}_\infty\, n$ for the non-propositionally-truncated variant of $\mathsf{SymAct}\, n$. We write $v \sim w$ instead of $\mathsf{SymAct}\, n\ v\ w$ when n is clear from context, and analogously $(\mathsf{Fin}\, n \to X)\,/_2 \sim$ in place of $(\mathsf{Fin}\, n \to X)\,/_2\, \mathsf{SymAct}\, n$.

The proof of Theorem 23 employs the fact that FMSet is invariant under set-truncation. The latter fact factors through the following lemma, stating that set-truncation distributes over finite families of types.

▶ **Lemma 1** (finChoiceEquiv). *For any* $n : \mathbb{N}$ *and type family* $Y : \mathsf{Fin}\, n \to \mathsf{Type}$*, there is an equivalence* $\mathsf{box} : ((k : \mathsf{Fin}\, n) \to \|Y\, k\|_2) \simeq \|(k : \mathsf{Fin}\, n) \to Y\, k\|_2$.

Proof. We sketch a proof for a constant type family $Y = (\lambda\_.\ X)$. The dependent case is analogous. The function underlying the equivalence is defined by induction on n. For $n = 0$ we have $\mathsf{Fin}\, 0 \simeq \bot$, so $\mathsf{box} =_{\mathsf{df}} (\lambda\_.\ |\mathsf{elim}_{\bot}|_2)$. In the inductive step, we lift the derivable "cons" operation $(::) : X \to (\mathsf{Fin}\, n \to X) \to (\mathsf{Fin}\,(1 + n) \to X)$ to the set-truncation. A two-sided inverse $\mathsf{unbox} : \|\mathsf{Fin}\, n \to X\|_2 \to \mathsf{Fin}\, n \to \|X\|_2$ of box is given by $\mathsf{unbox}\, \bar{v}\, k =_{\mathsf{df}} \mathsf{map}_{\|\_\|_2}(\lambda v.\ v\, k)\, \bar{v}$. ◀

The equivalence of Lemma 1 allows to define a variant of the elimination principle $\mathsf{elim}_{\|X\|_2}$ taking $\mathsf{Fin}\, n \to \|X\|_2$ as input instead of $\|X\|_2$ (a sort of "finite choice" principle for set-truncation):

$$\begin{aligned} \mathsf{elim}_{\|X\|_2\mathsf{fin}} : \{n : \mathbb{N}\}\, \{B : (\mathsf{Fin}\, n \to \|X\|_2) \to \mathsf{Type}\} \{s_B : \forall v.\ \mathsf{isSet}(B\, v)\} \\ \to (c : (w : \mathsf{Fin}\, n \to X) \to B\, (|\_|_2 \circ w)) \\ \to (v : \mathsf{Fin}\, n \to \|X\|_2) \to B\, v \end{aligned} \tag{2}$$

This comes with a (propositional) computation rule $\mathsf{elim}^{\beta}_{\|X\|_2\mathsf{fin}} : \mathsf{elim}_{\|X\|_2\mathsf{fin}}\ c\ (|\_|_2 \circ v) = c\, v$.

▶ **Theorem 2** (FMSetTruncInvariance). FMSet *is invariant under set-truncation: for any type* X*, there is an equivalence* $\mathsf{FMSet}\, \|X\|_2 \simeq \mathsf{FMSet}\, X$.

Proof. The equivalence is obtained from an isomorphism. The right-to-left function is $\mathsf{map}_{\mathsf{FMSet}}\ |\_|_2$. For the left-to-right direction, we use $\mathsf{elim}_{\|X\|_2\mathsf{fin}}$ in (2) to define a function typed $(\mathsf{Fin}\, n \to \|X\|_2) \to (\mathsf{Fin}\, n \to X)\,/_2\, \sim$ that turns set-truncation into a set-quotient, which is enough to obtain a function typed $\mathsf{FMSet}\, \|X\|_2 \to \mathsf{FMSet}\, X$. That these maps are mutual inverses follows from $\mathsf{elim}^{\beta}_{\|X\|_2\mathsf{fin}}$. ◀

3.4 Equivalence of Presentations

All encodings of finite multisets used in the preceding section induce equivalent functors:

▶ **Proposition 3** (FMSetEquivs). *For any type* X*, there is a sequence of equivalences*

$$\mathsf{FCM}\, X \overset{\alpha}{\simeq} \mathsf{List}\, X\,/_2\, \mathsf{Perm} \overset{\beta}{\simeq} \mathsf{List}\, X\,/_2\, \mathsf{Relator}\ (=) \overset{\gamma}{\simeq} \mathsf{FMSet}\, X,$$

which are natural in X*: for any* $f : X \to Y$*,* $\alpha \circ \mathsf{map}_{\mathsf{FCM}}\, f = \mathsf{map}_{\mathsf{List}\, X\,/_2\, \mathsf{Relator}\ (=)}\, f \circ \alpha$*, and similarly for* β *and* γ.

Proof. Equivalence α is obtained by observing that both types form a free commutative monoid on X, with addition $(\oplus)$ and $(+\!+)$ respectively. For β, note that Relator $(=)$ is a propositionally-valued relation, while Perm is generally not. Yet it is enough to provide a bi-implication between the relations to conclude that the set-quotients they define are equivalent, as mentioned in Section 3.2. Equivalence γ is obtained similarly, this time proving that the encodings of permutations ("intensionally" via the relator and "extensional" in terms of equivalence of types) are logically equivalent. Naturality is established directly. ◀

In the formalization, we make use of slight variations of the above types where convenient. These mostly concern presentation of lists (*e.g.* bundling lengths via $\mathsf{List}\, A \simeq \sum_{n:\mathbb{N}} \mathsf{Vec}\, A\, n$), and are easily seen to be naturally equivalent.

3.5 Definable Quotients and Sorting

In the absence of the axiom of choice, it is not generally possible to define a section of the equivalence class constructor $[\_]_2 : A \to A /_2 R$. A set-quotient $A /_2 R$ for which such a section exists is called *definable* [20]. Spelled out, there is a representative-picking function $\mathsf{rep} : A /_2 R \to A$ such that $[\mathsf{rep}\, x]_2 = x$ for all $x : A /_2 R$.

In the proof of Theorem 11 we employ the fact that the type of finite bags $\mathsf{FMSet}\, X$, for some specific choice of X, is linearly-ordered and $(\mathsf{Fin}\, n \to X) /_2 \sim$ is a definable set-quotient. A relation $(<)$ is a *linear order* when it is asymmetric, transitive, propositionally-valued and total, in the sense that the trichotomy $(x < y) + (x = y) + (y < x)$ holds for all $x, y : X$. If X is a set with linear order $(<)$, then lists over X can be sorted with respect to $(<)$ via a function $\mathsf{sort} : \mathsf{List}\, X \to \mathsf{List}\, X$ essentially implementing the insertion-sort algorithm, which allows the construction of a permutation typed $\mathsf{Perm}\ xs\ (\mathsf{sort}\, xs)$. Sorting is independent of the positions of each entry in the input list, therefore via $\mathsf{rec}_{\mathsf{List}\, X /_2 \mathsf{Perm}}$ we obtain a function $\mathsf{sortPerm} : \mathsf{List}\, X /_2 \mathsf{Perm} \to \mathsf{List}\, X$. It is not hard to show that $\mathsf{sortPerm}$ is a section of the equivalence class constructor, so $\mathsf{List}\, X /_2 \mathsf{Perm}$ is a definable quotient. Since $\mathsf{List}\, X /_2 \mathsf{Perm} \simeq \mathsf{FMSet}\, X$, we obtain the following result.

▶ **Proposition 4** (`SymActDefinable`). *If X is a linearly-ordered set, then $(\mathsf{Fin}\, n \to X) /_2 \sim$ is a definable quotient for all $n : \mathbb{N}$.*

In the presence of a linear order $(<)$ on X, we can extract from any proof that two lists are merely related by a permutation an actual permutation witnessing this:

▶ **Proposition 5** (`SymActUntruncate`). *If X is a linearly-ordered set, then for all $n : \mathbb{N}$ and $v, w : \mathsf{Fin}\, n \to X$ there exists a function typed $\mathsf{SymAct}\, n\ v\ w \to \mathsf{SymAct}_\infty\, n\ v\ w$,* i.e. *the propositional truncation in $\mathsf{SymAct}\, n\ v\ w$ can be removed.*

Proof. Since $\mathsf{FMSet}\, X \simeq \mathsf{List}\, X /_2 \mathsf{Perm}$, it is enough to define for all $xs\ ys : \mathsf{List}\, X$ a function $\|\mathsf{Perm}\ xs\ ys\|_1 \to \mathsf{Perm}\ xs\ ys$. To escape the truncation, we first implement a function $\mathsf{canonPerm} : \mathsf{Perm}\ xs\ ys \to \mathsf{Perm}\ xs\ ys$ returning a "canonical" way of permuting xs into ys. Given $\sigma : \mathsf{Perm}\ xs\ ys$, sorting yields a path $p_\sigma : \mathsf{sort}\, xs = \mathsf{sort}\, ys$. Composing (along p_σ) the permutations obtained from sorting xs and (un-)sorting ys gives the desired term $\mathsf{canonPerm}\, \sigma$. Since X is a set, p_σ lands in a proposition. Thus, $\mathsf{canonPerm}$ is *weakly constant* and lifts to a function from the truncation [7, Corollary 2]. ◀

To illustrate the computational behavior of `canonPerm`, see `BraidExample` in the code.

The order $(<)$ can be extended to a linear order on $\mathsf{List}\, X$ via the *lexicographic order*

$$\frac{}{\mathsf{Lex}\,(<)\ [\,]\ (y :: ys)} \qquad \frac{x < y}{\mathsf{Lex}\,(<)\ (x :: xs)\ (y :: ys)} \qquad \frac{x = y \qquad \mathsf{Lex}\,(<)\ xs\ ys}{\mathsf{Lex}\,(<)\ (x :: xs)\ (y :: ys)}$$

and further to $\mathsf{List}\, X /_2 \mathsf{Perm}$ by defining $\mathsf{LexPerm}\,(<)\ x\ y =_{\mathsf{df}} \mathsf{Lex}\,(<)\ (\mathsf{sortPerm}\, x)\ (\mathsf{sortPerm}\, y)$.

▶ **Proposition 6** (`linLexFMSet`). *If X is a linearly-ordered set, then $(\mathsf{Fin}\, n \to X) /_2 \sim$ is linearly-ordered for all $n : \mathbb{N}$.*

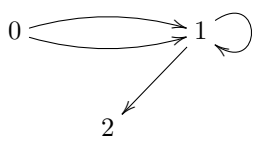
4 The Final Coalgebra in Sets

We now turn to constructing the final coalgebra of the finite bag functor, given by one of the equivalent definitions in Section 3.

Given a functor $F : \mathsf{Type} \to \mathsf{Type}$, the types of *coalgebras* and *coalgebra morphisms* between two coalgebras (A, a) and (B, b) are

$$\mathsf{Coalg}\ F =_{\mathsf{df}} \sum (A : \mathsf{Type}).\ A \to FA$$
$$\mathsf{CoalgMor}\ F\ (A, a)\ (B, b) =_{\mathsf{df}} \sum (f : A \to B).\ b \circ f = (\mathsf{map}_F\ f) \circ a$$

Coalgebras can be used to represent transition systems. For example, the coalgebra on the right encodes the small transition system on the left:



$$c : \mathsf{Fin}\,3 \to \mathsf{FCM}(\mathsf{Fin}\,3)$$
$$c\ 0 =_{\mathsf{df}} \eta\ 1 \oplus \eta\ 1$$
$$c\ 1 =_{\mathsf{df}} \eta\ 1 \oplus \eta\ 2$$
$$c\ 2 =_{\mathsf{df}} \varepsilon$$

A coalgebra is *final* if there exists a unique coalgebra morphism from any other coalgebra. This can be formalized by saying that there is a coalgebra $C : \mathsf{Coalg}\ F$ such that the type $\mathsf{CoalgMor}\ F\ D\ C$ is contractible for any other coalgebra D. These definitions are the same of Ahrens *et al.* [3], which they only consider in the case of F being a polynomial functor.

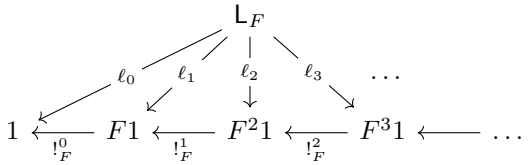
We analyze two constructions of the final coalgebra for the finite bag functor: as an ω-limit and as a set-quotient of the final coalgebra of the List functor.

4.1 As an ω-Limit

Consider the chain in (1), for some $F : \mathsf{Type} \to \mathsf{Type}$. We formally define $F^n 1$ by recursion on n: $F^0 1 =_{\mathsf{df}} 1$ and $F^{1+n} 1 =_{\mathsf{df}} F(F^n 1)$. Similarly we can define the iteration $\mathsf{map}^n_F\ !$, which we denote $!^n_F$. In HoTT, the *(homotopy) limit* of the chain is definable as

$$\lim_n (F^n 1) =_{\mathsf{df}} \sum (x : (n : \mathbb{N}) \to F^n 1).\ \forall n.\ !^n_F\ (x\ (1 + n)) = x\ n$$

Write $\mathsf{L}_F =_{\mathsf{df}} \lim_n (F^n 1)$. An element of the limit consists of an element $x\,n : F^n 1$, for all $n : \mathbb{N}$, and a proof that restricting $x\,(1 + n)$ to $F^n 1$ via $!^n_F$ is equal to $x\,n$. Writing $\ell_n : \mathsf{L}_F \to F^n 1$ for the n-th projection from the limit, we obtain the usual diagram:



(3)

The limit is invariant with respect to shifting the chain by one position, *i.e.* there is an equivalence $\mathsf{shift} : \mathsf{L}_F \simeq \mathsf{L}^{\mathsf{sh}}_F$, where $\mathsf{L}^{\mathsf{sh}}_F =_{\mathsf{df}} \lim_n (F^{1+n} 1)$. We use $\ell_n : \mathsf{L}^{\mathsf{sh}}_F \to F^{1+n} 1$ also for the n-th projection from the shifted limit. If F is set-valued, the limit L_F is also a set. Notice that for naturally equivalent and set-valued F and G, we have $\mathsf{L}_F \simeq \mathsf{L}_G$, since naturally equivalent chains have equivalent limits; see `chainEquivToLimitEquiv` in the formalization for details. This implies that, when proving properties of this limit for finite bags, we can use any of the equivalent presentations in Proposition 3 as convenient.

In classical set theory, $\mathsf{L}_{\mathsf{FMSet}}$ can be proved to be the final coalgebra of FMSet. The proof proceeds by first constructing a function $\mathsf{pres}_{\mathsf{FMSet}} : \mathsf{FMSet}\ \mathsf{L}_{\mathsf{FMSet}} \to \mathsf{L}^{\mathsf{sh}}_{\mathsf{FMSet}}$ via the universal property of the limit: take $\ell_n\ (\mathsf{pres}_{\mathsf{FMSet}}\ s)$ as $\mathsf{map}_{\mathsf{FMSet}}\ \ell_n\ s$. The function $\mathsf{pres}_{\mathsf{FMSet}}$ is then proved to be an equivalence, showing that FMSet preserves the ω-limit. The composition of shift with the inverse $\mathsf{pres}^{-1}_{\mathsf{FMSet}}$ provides a coalgebra structure for $\mathsf{L}_{\mathsf{FMSet}}$. This can be proved to be final, again using the universal property of the limit.

Constructively, there are issues in proving that $\mathsf{pres}_{\mathsf{FMSet}}$ is an equivalence. Its injectivity is equivalent to the *lesser limited principle of omniscience* (LLPO) [6, Ch. 1]. The latter is a weak version of the law of the excluded middle, and it is not provable from intuitionistic axioms alone. It states that, given an infinite stream of Boolean values that yields true in at most one position, one can decide whether all even or all odd positions are false. Both injectivity of $\mathsf{pres}_{\mathsf{FMSet}}$ and LLPO are propositions, so to establish an equivalence, it is sufficient to find a bi-implication between them.

▶ **Theorem 7** (`InjectiveFMSetPresToLLPO`). *If* $\mathsf{pres}_{\mathsf{FMSet}}$ *is injective, then LLPO holds.*

Proof. In this proof we use FCM instead of FMSet. The statement of the theorem holds since $\mathsf{FCM}\,X$ is naturally equivalent to $\mathsf{FMSet}\,X$. It is sufficient to show that the injectivity of $\mathsf{pres}_{\mathsf{FCM}}$ implies that the following type is inhabited:

$$\begin{aligned}&(x, y_1, y_2 : \mathsf{L}_{\mathsf{FCM}}) \to (ys : \mathbb{N} \to \mathsf{L}_{\mathsf{FCM}})\\ &\quad \to (\mathsf{split} : \forall n.\ ys\ n = y_1 + ys\ n = y_2)\ (\mathsf{diag} : \forall n.\ \ell_n\ x = \ell_n\ (ys\ n))\\ &\quad \to \|x = y_1 + x = y_2\|_1\end{aligned} \tag{4}$$

This is a form of *completeness* of two-element subsets of the ω-limit: every converging sequence ys consisting of elements from the subset $\{y_1, y_2\}$ has its limit x also belonging to the subset $\{y_1, y_2\}$. Mandelkern [21] has proved the equivalence of LLPO with completeness of two-element subsets of real numbers. We adapt their proof, for details refer to either [29, Theorem 7] or the formalization (`CompleteToLLPO`).

To prove completeness, assume $x, y_1, y_2, ys, \mathsf{split}$ and diag as in (4). Using split, define the complement of ys as $\overline{ys}\ n =_{\mathsf{df}} y_2$ if $ys\ n = y_1$ and $\overline{ys}\ n =_{\mathsf{df}} y_1$ if $ys\ n = y_2$. The diagonal of $\overline{ys}$ also has the limit-property, *i.e.*

$$\forall n.\ !^n_{\mathsf{FCM}}(\ell_{1+n}\ (\overline{ys}\,(1+n))) = \ell_n\ (\overline{ys}\ n) \tag{5}$$

For this, fix n and check the four cases generated by inspecting $\mathsf{split}\,n$ and $\mathsf{split}\,(1+n)$. In one case, (5) reduces to the limit-property of y_1, in another to that of y_2 and in the remaining cases to Lemma 8 below. Call $\overline{x} : \mathsf{L}_{\mathsf{FCM}}$ the element of the limit such that $\ell_n\,\overline{x} =_{\mathsf{df}} \ell_n\,(\overline{ys}\ n)$.

Write $\{x, y\} =_{\mathsf{df}} \eta\,x \oplus \eta\,y$ for the two-element bag comprising of x and y. For all n, we know that $\{ys\ n, \overline{ys}\ n\} = \{y_1, y_2\}$ either by refl or $\oplus\mathsf{comm}$, depending on $\mathsf{split}\,n$. Using the latter equality, the definition of $\mathsf{pres}_{\mathsf{FCM}}$ and the assumption diag, we can form the following sequence of equalities:

$$\begin{aligned}\ell_n\,(\mathsf{pres}_{\mathsf{FCM}}\{x, \overline{x}\}) &= \{\ell_n\,x, \ell_n\,\overline{x}\} = \{\ell_n\,(ys\ n), \ell_n\,(\overline{ys}\ n)\}\\ &= \ell_n\,(\mathsf{pres}_{\mathsf{FCM}}\{ys\ n, \overline{ys}\ n\}) = \ell_n\,(\mathsf{pres}_{\mathsf{FCM}}\{y_1, y_2\})\end{aligned}$$

which implies $\mathsf{pres}_{\mathsf{FCM}}\{x, \overline{x}\} = \mathsf{pres}_{\mathsf{FCM}}\{y_1, y_2\}$. From the injectivity of $\mathsf{pres}_{\mathsf{FCM}}$ it follows that $\{x, \overline{x}\} = \{y_1, y_2\}$, which also implies that (merely) $x = y_1$ or $x = y_2$. ◀

The above depends on a property of sequences in L_F that are "approximated" by some $x : \mathsf{L}_F$:

▶ **Lemma 8** (`LimitAlternationLemma`). *For any functor* F *where* $x : \mathsf{L}_F$ *and* $ys : \mathbb{N} \to \mathsf{L}_F$ *such that* $P : \forall n.\ \ell_n\,x = \ell_n\,ys_n$*, we have* $\forall n.\ !^n_F(\ell_{n+1}\,ys_n) = \ell_n\,ys_{n+1}$.

Proof. By alternating application of the limit property and assumption P, we obtain

$$!^n_F(\ell_{n+1}\,ys_n) = \ell_n\,ys_n \overset{P_n}{=} \ell_n\,x = !^n_F(\ell_{n+1}\,x) \overset{P_{n+1}}{=} !^n_F(\ell_{n+1}\,ys_{n+1}) = \ell_n\,ys_{n+1}$$ ◀

▶ **Theorem 9** (LLPOToInjectiveFMSetPres). *LLPO implies the injectivity of* $\mathsf{pres}_{\mathsf{FMSet}}$.

For the proof of Theorem 9, which employs the functor $\mathsf{List}(-)\,/_2\,\mathsf{Relator}\,(=)$ instead of FMSet, we refer the reader to our Agda formalization. The proof is similar to the one of a related result [29, Theorem 9]: the injectivity of $\mathsf{pres}_{\mathsf{Pfin}} : \mathsf{Pfin}\ \mathsf{L}_{\mathsf{Pfin}} \to \mathsf{L}^{\mathsf{sh}}_{\mathsf{Pfin}}$, where Pfin is the finite powerset functor, is derivable from LLPO and the axiom of countable choice. It turns out that countable choice is not needed, neither in Theorem 9 nor in Theorem 9 of [29].

Nevertheless, we are able to salvage the fact that $\mathsf{pres}_{\mathsf{FMSet}}$ has a section/right-inverse which targets the shifted limit. This implies that FMSet weakly preserves the ω-limit $\mathsf{L}_{\mathsf{FMSet}}$, but strong limit-preservation is equivalent to LLPO.

▶ **Lemma 10** (linLexIterFMSet). *For all* n, $\mathsf{FMSet}^n\,1$ *is linearly-ordered.*

Proof. Define $(<^n) : \mathsf{FMSet}^n\,1 \to \mathsf{FMSet}^n\,1 \to \mathsf{Type}$ by induction on n: $(<^0)$ is the empty relation and $x <^{1+n} y =_{\mathsf{df}} \mathsf{LexPerm}\,(<^n)\,x\,y$. Since the empty relation is linear, Proposition 6 implies that the order $(<^n)$ is linear for all $n : \mathbb{N}$. ◀

▶ **Theorem 11** (FMSetPresSection). *The function* $\mathsf{pres}_{\mathsf{FMSet}}$ *has a section.*

Proof. Let $s : \mathsf{L}^{\mathsf{sh}}_{\mathsf{FMSet}}$, we build an element $t : \mathsf{FMSet}\ \mathsf{L}_{\mathsf{FMSet}}$ in the fiber of $\mathsf{pres}_{\mathsf{FMSet}}$ over s. The size (*i.e.* the 1st projection) of the bags $\ell_n\,s$ is the same for all n, call it n^*. We set the size of t to be n^*. Given an index $k : \mathsf{Fin}\,n^*$, we now search for an element $u\,k : \mathsf{L}_{\mathsf{FMSet}}$ for defining $t =_{\mathsf{df}} (n^*, u)$.

For each $d : \mathbb{N}$, we know that $\ell_{d+1}\,s$ is path equal to a pair of the form (n^*, v_d). In order to construct u we need access to a representative of the equivalence class $v_d : (\mathsf{Fin}\,n^* \to \mathsf{FMSet}^d\,1)\,/_2\,\sim$ for each d. We know that this can be done using Lemma 10 and Proposition 4. Let $w_d : \mathsf{Fin}\,n^* \to \mathsf{FMSet}^d\,1$ be the canonical representative of v_d. The limit-property of s can be translated to the mere existence of a permutation $\sigma_d : \mathsf{Fin}\,n^* \simeq \mathsf{Fin}\,n^*$ such that $p : !^d_{\mathsf{FMSet}}(w_{1+d}\,k) = w_d\,(\sigma_d\,k)$, for all $d : \mathbb{N}$ and $k : \mathsf{Fin}\,n^*$. The construction of u also requires access to each permutation σ_d, which sits inside a propositional truncation for each d. We can access all these permutations by invoking Lemma 10 and Proposition 5.

We now have all the ingredients for building u. Define a permutation $\sigma^*_d : \mathsf{Fin}\,n^* \simeq \mathsf{Fin}\,n^*$ by induction on d: $\sigma^*_0 =_{\mathsf{df}} \mathsf{id}$, $\sigma^*_{1+d} =_{\mathsf{df}} \sigma_d^{-1} \circ \sigma^*_d$. Then take u such that $\ell_d\,(u\,k) =_{\mathsf{df}} w_d\,(\sigma^*_d\,k)$. One can show that $u\,k : \mathsf{L}_{\mathsf{FMSet}}$ for all $k : \mathsf{Fin}\,n^*$ since $!^d_{\mathsf{FMSet}}(w_{d+1}(\sigma^*_{d+1}\,k)) \overset{p}{=} w_d(\sigma_d(\sigma^*_{d+1}\,k)) = w_d(\sigma_d(\sigma_d^{-1}(\sigma^*_d\,k))) = w_d(\sigma^*_d\,k)$, and that t is indeed in the fiber of $\mathsf{pres}_{\mathsf{FMSet}}$ over s. ◀

By *Lambek's theorem*, every final coalgebra is necessarily an equivalence. Assuming LLPO we have $\mathsf{FMSet}\,\mathsf{L}_{\mathsf{FMSet}} \simeq \mathsf{L}_{\mathsf{FMSet}}$, and proving that the coalgebra underlying this equivalence is final in the category of sets is straightforward using the universal property of the limit.

4.2 As a Quotient of the Final List-Coalgebra

Instead of considering a type of unordered trees quotiented at each step of the construction, we investigate whether it is possible to define a final FMSet-coalgebra by quotienting the type of ordered trees by some suitable relation. It is known that the limit $\mathsf{L}_{\mathsf{List}} =_{\mathsf{df}} \lim_n(\mathsf{List}^n\,1)$ is the final coalgebra of the list functor in HoTT [3]. The limit $\mathsf{L}_{\mathsf{List}}$ is a type of non-wellfounded *ordered* trees, and we denote by $\mathsf{coalg}_{\mathsf{List}}$ its coalgebra structure. By choosing a suitable relation R, one can hope to obtain a type of *unordered* trees $\mathsf{L}_{\mathsf{List}}\,/_2\,R$ endowed with a FMSet-coalgebra structure. We choose R to be a notion of *bisimilarity* Bisim, obtained iteratively

from the *relation lifting* Relator applied to finite approximations of trees in $\mathsf{L}_{\mathsf{List}}$ [13]:

$$\begin{aligned}&\mathsf{Approx}^n : \mathsf{List}^n\ 1 \to \mathsf{List}^n\ 1 \to \mathsf{Type}\\&\mathsf{Approx}^0\ x\ y =_{\mathsf{df}} 1\\&\mathsf{Approx}^{1+n}\ x\ y =_{\mathsf{df}} \mathsf{Relator}\ (\mathsf{Approx}^n)\ x\ y.\end{aligned}$$

From the fact that $(\forall x, y.\ R\,x\,y \to S\,x\,y)$ implies $\forall xs, ys.\ \mathsf{Relator}\ R\ xs\ ys \to \mathsf{Relator}\ S\ xs\ ys$, we obtain, for $s, t : \mathsf{L}_{\mathsf{List}}$, a chain of propositions

$$\mathsf{Approx}^0(\ell_0\ s)\ (\ell_0\ t) \leftarrow \mathsf{Approx}^1(\ell_1\ s)\ (\ell_1\ t) \leftarrow \mathsf{Approx}^2(\ell_2\ s)\ (\ell_2\ t) \leftarrow \cdots \tag{6}$$

The desired relation $\mathsf{Bisim}\,s\,t$ is the limit of the chain in (6).

To find a coalgebra structure on $\mathsf{L}_{\mathsf{List}} /_2 \mathsf{Bisim}$, we investigate whether $\mathsf{coalg}_{\mathsf{List}}$ lifts to a coalgebra of setoids $(\mathsf{L}_{\mathsf{List}}, \mathsf{Bisim}) \to (\mathsf{List}\,\mathsf{L}_{\mathsf{List}}, \mathsf{Relator}\,\mathsf{Bisim})$ and if so, whether this induces a (final) coalgebra on the quotient. For this, one needs to show that it is a *setoid-morphism*, *i.e.* for $s, t : \mathsf{L}_{\mathsf{List}}$, if $\mathsf{Bisim}\ s\ t$ then $\mathsf{Relator}\,\mathsf{Bisim}\ (\mathsf{coalg}_{\mathsf{List}}\ s)\ (\mathsf{coalg}_{\mathsf{List}}\ t)$. Once again, the same issue we found when trying to prove the injectivity of $\mathsf{pres}_{\mathsf{FMSet}}$ in Section 4.1 arises:

▶ **Theorem 12** (`isSetoidMorphismCoalgListToLLPO`). *If* $\mathsf{coalg}_{\mathsf{List}}$ *is a setoid-morphism, then LLPO holds.*

The proof is similar to that of Theorem 7. Similarly to Theorem 9, the converse is also true. Nevertheless, the inverse of $\mathsf{coalg}_{\mathsf{List}}$ is always a setoid-morphism. Therefore $\mathsf{coalg}_{\mathsf{List}}$ is an equivalence of setoids whenever it is a setoid-morphism, *i.e.* LLPO holds. Under this assumption alone it is the final coalgebra of an endofunctor in the category of setoids:

▶ **Theorem 13** (`finalFMSetoidCoalgebra`). *Assuming* $\mathsf{coalg}_{\mathsf{List}}$ *is a setoid-morphism, the setoid* $(\mathsf{L}_{\mathsf{List}}, \mathsf{Bisim})$ *has a coalgebra structure for the functor* $(X, R) \mapsto (\mathsf{List}\,X, \mathsf{Relator}\,R)$*, which is final in the category of setoids.*

Promisingly, we can show that the resulting quotient is a fixpoint for FMSet, and in particular a coalgebra (of sets):

▶ **Theorem 14** (`FMSetFixpointTree/Bisim`). *If* $\mathsf{coalg}_{\mathsf{List}}$ *is a setoid-morphism, it lifts to an equivalence* $\mathsf{coalg}_{\mathsf{FMSet}} : \mathsf{L}_{\mathsf{List}} /_2 \mathsf{Bisim} \xrightarrow{\simeq} \mathsf{FMSet}(\mathsf{L}_{\mathsf{List}} /_2 \mathsf{Bisim})$.

Proof. For the proof, we employ the equivalent functor $\mathsf{List}(-) /_2 \mathsf{Relator}\ (=)$ instead of FMSet. The assumption implies that $\mathsf{coalg}_{\mathsf{List}}$ lifts to a function $\mathsf{L}_{\mathsf{List}} /_2 \mathsf{Bisim} \to \mathsf{FMSet}(\mathsf{L}_{\mathsf{List}} /_2 \mathsf{Bisim})$, definable by recursion on the set-quotient. An inverse $\mathsf{FMSet}(\mathsf{L}_{\mathsf{List}} /_2 \mathsf{Bisim}) \to \mathsf{L}_{\mathsf{List}} /_2 \mathsf{Bisim}$ is definable since $\mathsf{coalg}^{-1}_{\mathsf{List}}$ is always a setoid-morphism and $\mathsf{List}\,X /_2 \mathsf{Relator}\,R$ is an *effective quotient* for any setoid (X, R). ◀

However, like in case of the finite powerset, this fixpoint is not obviously the final FMSet-coalgebra. We were able to prove this assuming the axiom of choice:

▶ **Theorem 15** (`FinalFMSetCoalgebra`). *Assuming the axiom of choice, the fixpoint of Theorem 14 is the final* FMSet*-coalgebra in the category of sets.*

Proof. Define abbreviations $U =_{\mathsf{df}} \mathsf{L}_{\mathsf{List}} /_2 \mathsf{Bisim}$ and $R =_{\mathsf{df}} \mathsf{Relator}\ (=)$, and use the presentation of FMSet used in the proof of Theorem 14. To build a coalgebra morphism $u_c : C \to U$ from a given coalgebra $c : C \to \mathsf{FMSet}\,C$ to $\mathsf{coalg}_{\mathsf{FMSet}} : U \to \mathsf{FMSet}\,U$, one defines a function $u' : (C \to \mathsf{List}\,C) /_2 R^* \to (C \to U)$. Here, R^* is the pointwise lifting of R, and u' is obtained by recursion from the unique List-coalgebra morphism typed $C \to \mathsf{L}_{\mathsf{List}}$. The axiom of choice implies that the canonical map $(C \to \mathsf{List} /_2 R) \to (C \to \mathsf{List}\,C) /_2 R^*$ has a section θ for *arbitrary* C. This is sufficient to prove that $u_c =_{\mathsf{df}} u'(\theta(c))$ is the unique FMSet-coalgebra morphism from (C, c) to $(\mathsf{L}_{\mathsf{List}} /_2 \mathsf{Bisim}, \mathsf{coalg}_{\mathsf{FMSet}})$. ◀

5 The Finite Bag Functor in Groupoids

The results of Section 4 are evidence that the set-based definitions of finite bags from Section 3 are not fit for a fully constructive construction of the final coalgebra. In this section we study a groupoid-based definition and, following the ideas of Kock [16] and Finster *et al.* [10], argue that the correct perspective on finite bags in HoTT is to define them as groupoids instead of sets, particularly for the goal of final semantics. The rationale is that identifications of bags are permutations, and these should inherently be treated as *data*. Instead of viewing bags as quotients of lists, thereby "forgetting" about the permutations, we define a type of lists with "more identifications". Since all constructions based on this type have to be homotopy coherent, they will automatically respect the extra data, making them invariant under permutation for free. We define two equivalent type families Tote and Bag of finite bags valued in groupoids, and substantiate the previous claims by showing that the set-truncation of the former is equivalent to FMSet (Theorem 17), and constructing the final coalgebra of the latter in a straightforward way (Theorem 21 and Corollary 22).

First, recall one way of defining finite sets in HoTT [11]. A type B is called *(Bishop-)finite* if $\mathsf{isFinSet}\, B =_{\mathsf{df}} \sum(n : \mathbb{N}).\ \|B \simeq \mathsf{Fin}\, n\|_1$ holds, and we denote the collection of such types by $\mathsf{FinSet} =_{\mathsf{df}} \sum(B : \mathsf{Type}).\ \mathsf{isFinSet}\, B$. The underlying type of a FinSet is accessed via the first projection $\langle - \rangle : \mathsf{FinSet} \to \mathsf{Type}$.

The type $\mathsf{isFinSet}\, B$ is a proposition and any type B satisfying the predicate is a set. It follows that FinSet forms a groupoid. Note that FinSet is a *large* type, *i.e.* $\mathsf{FinSet} : \mathsf{Type}_1$. From this, we can define a "tote" (in the sense of a "large bag") $\mathsf{Tote} : \mathsf{Type} \to \mathsf{Type}_1$ as

$$\mathsf{Tote}\, X =_{\mathrm{df}} \sum (B : \mathsf{FinSet}).\ \langle B \rangle \to X,$$

Elements of $\mathsf{Tote}\, X$ are pairs consisting of a finite set B and a function from (the type underlying) B to X which picks the elements in the tote. Univalence implies that the path type $(B, v) = (C, w)$ in $\mathsf{Tote}\, X$ is equivalent to the type of dependent pairs consisting of an equivalence $\sigma : \langle B \rangle \simeq \langle C \rangle$ and a path $v = w \circ \sigma$. This indicates that $\mathsf{Tote}\, X$ is not a set, in general it is at least a groupoid.

▶ **Proposition 16** (`isGroupoidTote`). *If X is a groupoid, then $\mathsf{Tote}\, X$ is a groupoid.*

Proof. Since X is a groupoid, the function type $\langle B \rangle \to X$ is a groupoid for any $B : \mathsf{FinSet}$. The type FinSet is also a groupoid, so the entire Σ-type is a groupoid. ◀

Similar to how $\mathsf{FMSet}\, X$ is the free commutative monoid on X, $\mathsf{Tote}\, X$ can be proved equivalent to the *free symmetric monoidal groupoid* on X [22, Corollary 5.103], which serves as an alternative proof of MacLane's coherence for symmetric monoidal categories. It differs from $\mathsf{FMSet}\, X$ in that path equality in the former records the permutations between the (finite sets representing) sizes of the bags, while the second only cares about the mere existence of a permutation. Nonetheless, the two definitions become equivalent when we set-truncate the type of totes.

▶ **Theorem 17** (`FMSetToteTruncEquiv`, `isNatural-FMSetToteTruncEquiv`). *For any type X, $\|\mathsf{Tote}\, X\|_2 \simeq \mathsf{FMSet}\, X$. The equivalence is natural in X.*

Proof. The proof proceeds by constructing an isomorphism $\mathsf{FMSet}\, X \cong \|\mathsf{Tote}\, X\|_2$.

A function $\mathsf{toTote} : \mathsf{FMSet}\, X \to \|\mathsf{Tote}\, X\|_2$ is defined by first giving a function $f : \forall\{n\}.\ (\mathsf{Fin}\, n \to X) \to \|\mathsf{Tote}\, X\|_2$ and then showing that it respects $(\sim)$. Take $f(v) =_{\mathsf{df}} |(\mathsf{Fin}\, n, v)|_2$, since $\mathsf{Fin}\, n$ is a finite set. To prove that $v \sim w$ implies $f(v) = f(w)$, note that

the conclusion is a proposition, thus by invoking the recursion principle of propositional truncation we can assume given a permutation σ such that $r : v = w \circ \sigma$. By univalence, $\mathsf{ua}\,\sigma : \mathsf{Fin}\,n = \mathsf{Fin}\,n$, and transporting r along this path yields $p : (\mathsf{Fin}\,n, v) = (\mathsf{Fin}\,n, w)$. Then $\mathsf{cong}\,|\_|_2\,p : f(v) = f(w)$ as desired.

A function $\mathsf{toFMSet} : \|\mathsf{Tote}\,X\|_2 \to \mathsf{FMSet}\,X$ is defined via $\mathsf{rec}_{\|\mathsf{Tote}\,X\|_2}$, so it is enough to provide $g : \mathsf{Tote}\,X \to \mathsf{FMSet}\,X$. Assume given a finite set B of size n with $e : \|B \simeq \mathsf{Fin}\,n\|_1$ and $v : B \to X$. We would like to return something in $(\mathsf{Fin}\,n \to X)/_2 \sim$ by recursion on e, but this cannot work since the return type is a set. We can however employ a different recursion principle of propositional truncation [7, Corollary 2], which allows to define a function into a set provided that it is *(weakly) constant* (in the sense of [17]). Define $g' : (B \simeq \mathsf{Fin}\,n) \to (\mathsf{Fin}\,n \to X)/_2 \sim$ as $g'\,\alpha =_{\mathsf{df}} [v \circ \alpha]_2$, which can be proved to be constant and therefore well-defined. We can then take $g\,((B, n, e), v) =_{\mathsf{df}} (n, g'\,e)$. Proving $\mathsf{toFMSet} \circ \mathsf{toTote} = \mathsf{id}$ is straightforward. Proving $\mathsf{toTote} \circ \mathsf{toFMSet} = \mathsf{id}$ reduces to showing that $v \circ \alpha \sim v$ for any $v : B \to X$ and $\alpha : \mathsf{Fin}\,n \simeq B$, which is also direct. ◀

Before studying $\mathsf{L}_{\mathsf{Tote}}$, notice that the iteration $\mathsf{Tote}^n\,1$ is not well-typed as Tote targets a large universe. We could in principle define $\mathsf{Tote}' : \mathsf{Type}_1 \to \mathsf{Type}_1$ which does not raise the universe level by first lifting the unit type 1 to the universe Type_1. The resulting limit would be a large groupoid in $\mathsf{L}_{\mathsf{Tote}'} : \mathsf{Type}_1$. Instead, we define an equivalent small variant of Tote with the help of HITs.

Following [10], we first introduce an equivalent but small definition Bij of the type of finite sets FinSet. This is equivalent to the *groupoid-quotient* [24, 28] of the (categorical) groupoid with objects given by natural numbers and morphisms between n and m given by equivalences in $\mathsf{Fin}\,m \simeq \mathsf{Fin}\,n$. It is possible to prove that hom also preserves identities and inverses.

$$\frac{n : \mathbb{N}}{\mathsf{obj}\,n : \mathsf{Bij}} \qquad \frac{m, n : \mathbb{N} \qquad \alpha : \mathsf{Fin}\,m \simeq \mathsf{Fin}\,n}{\mathsf{hom}\;\alpha : \mathsf{obj}\,m = \mathsf{obj}\,n}$$

$$\frac{m, n, o : \mathbb{N} \qquad \alpha : \mathsf{Fin}\,m \simeq \mathsf{Fin}\,n \qquad \beta : \mathsf{Fin}\,n \simeq \mathsf{Fin}\,o}{\mathsf{hom}\,(\beta \circ \alpha) = \mathsf{hom}\;\alpha \bullet \mathsf{hom}\;\beta} \qquad \frac{}{\mathsf{isGroupoid}\;\mathsf{Bij}}$$

▶ **Proposition 18** (`BinFinSetEquiv`). *There is an equivalence* $\mathsf{Bij} \simeq \mathsf{FinSet}$. *In particular, one can extract a type* $\langle x \rangle : \mathsf{Type}$ *from each* $x : \mathsf{Bij}$.

A small type of finite bags is defined by replacing FinSet with Bij.

$$\mathsf{Bag}\,X =_{\mathsf{df}} \sum(x : \mathsf{Bij}).\ \langle x \rangle \to X$$

▶ **Proposition 19** (`BagToteEquiv`). *For any type* X*, the equivalence of Proposition 18 extends to an equivalence* $\mathsf{Bag}\,X \simeq \mathsf{Tote}\,X$ *natural in* X.

Combining the above with Theorem 17 yields the follows convenient characterization:

▶ **Corollary 20** (`TruncBagFMSetEquiv`). *For any* X, $\|\mathsf{Bag}\,X\|_2 \simeq \mathsf{FMSet}\,X$ *naturally in* X.

6 The Final Coalgebra in Groupoids

When defined this way, it is immediate that Bag is the polynomial functor associated to the *container* $(\mathsf{Bij}, \langle \_ \rangle)$ in the sense of [3, Definition 2]. Crucially, [3, Theorem 7] proves that for such functors, the ω-limit is the carrier of the final coalgebra; independently of the homotopy level of the container it is associated to. Therefore $\mathsf{L}_{\mathsf{Bag}}$ carries the structure of a final Bag-coalgebra, even though Bij is not a set:

▶ **Theorem 21** (isLimitPreservingBag). *The map* $\mathsf{pres}_{\mathsf{Bag}} : \mathsf{Bag}\ \mathsf{L}_{\mathsf{Bag}} \to \mathsf{L}^{\mathsf{sh}}_{\mathsf{Bag}}$ *is an equivalence of groupoids.*

▶ **Corollary 22** ([3, Theorem 7]). $\mathsf{pres}^{-1}_{\mathsf{Bag}} \circ \mathsf{shift} : \mathsf{L}_{\mathsf{Bag}} \to \mathsf{Bag}\ \mathsf{L}_{\mathsf{Bag}}$ *is the final* Bag*-coalgebra.*

We refer to the formalization of [3] for a proof of Corollary 22.

Iterated application of Corollary 20 and Theorem 2 shows that $\|\mathsf{Bag}^n\ 1\|_2$ is equivalent to $\mathsf{FMSet}^n\ 1$ for all n (IterTruncBagFMSetEquiv). One might wonder whether similarly the set-truncation of $\mathsf{L}_{\mathsf{Bag}}$ delivers the final coalgebra of FMSet. We are able to show that $\|\mathsf{L}_{\mathsf{Bag}}\|_2$ is a fixpoint of FMSet. But to prove finality, we require the additional assumption of the axiom of choice and a "higher" version $\mathsf{AC}_{3,2}$ of the axiom of choice [26, Exercise 7.8],[2] which states that for a set X and a groupoid-valued type family Y on X, the following type is inhabited: $((x : X) \to \|Y\ x\|_2) \to \|(x : X) \to Y\ x\|_2$.

▶ **Theorem 23** (FMSetFixpointTruncBagLim). *The set-truncation of* $\mathsf{L}_{\mathsf{Bag}}$ *is a fixpoint of* FMSet*,* i.e. *there is an equivalence* $\mathsf{FMSet}\ \|\mathsf{L}_{\mathsf{Bag}}\|_2 \simeq \|\mathsf{L}_{\mathsf{Bag}}\|_2$.

Proof. The equivalence is obtained from the composition

$$\mathsf{FMSet}\ \|\mathsf{L}_{\mathsf{Bag}}\|_2 \overset{\alpha}{\simeq} \mathsf{FMSet}\ \mathsf{L}_{\mathsf{Bag}} \overset{\beta}{\simeq} \|\mathsf{Bag}\ \mathsf{L}_{\mathsf{Bag}}\|_2 \overset{\gamma}{\simeq} \|\mathsf{L}_{\mathsf{Bag}}\|_2$$

where α is invariance of FMSet under set-truncation (Theorem 2), β follows from Corollary 20, and γ follows from Theorem 21. ◀

Let $\mathsf{coalg}_{\mathsf{FMSet}}$ be the coalgebra underlying the equivalence of Theorem 23.

▶ **Theorem 24** (FMSetFinalCoalgebra). *Assuming axiom of choice and* $\mathsf{AC}_{3,2}$*,* $\|\mathsf{L}_{\mathsf{Bag}}\|_2$ *is the final coalgebra of* FMSet *in the category of sets.*

Proof. Let $c : X \to \mathsf{FMSet}\ X$ be a coalgebra, which by Corollary 20 is equivalent to having a function $c' : X \to \|\mathsf{Bag}\ X\|_2$. Applying $\mathsf{AC}_{3,2}$ on c' gives $c'' : \|X \to \mathsf{Bag}\ X\|_2$. Invoking $\mathsf{rec}_{\|X \to \mathsf{Bag}\ X\|_2}$ on c'', we receive $g : X \to \mathsf{Bag}\ X$. From the finality of $\mathsf{L}_{\mathsf{Bag}}$ in Corollary 22, there exists a unique Bag-coalgebra morphism $f^* : X \to \mathsf{L}_{\mathsf{Bag}}$ and we define $f\ x =_{\mathsf{df}} |f^*\ x|_2$. The function f is the desired unique FMSet-coalgebra morphism between (X, c) and $(\|\mathsf{L}_{\mathsf{Bag}}\|_2, \mathsf{coalg}_{\mathsf{FMSet}})$. The proof of uniqueness uses an application of the axiom of choice. We refer to the formalization for details. ◀

In the absence of the axiom of choice and $\mathsf{AC}_{3,2}$, the type $\|\mathsf{L}_{\mathsf{Bag}}\|_2$ can still be used to give semantics to transition systems with *finite* set of states.

▶ **Proposition 25** (uniqueCoalgMorphismFinCarrier). *Given any* $n : \mathbb{N}$ *and a coalgebra* $c : \mathsf{Fin}\ n \to \mathsf{FMSet}(\mathsf{Fin}\ n)$*, there exists a unique coalgebra morphism from* $(\mathsf{Fin}\ n, c)$ *to* $(\|\mathsf{L}_{\mathsf{Bag}}\|_2, \mathsf{coalg}_{\mathsf{FMSet}})$.

This is true since $\mathsf{AC}_{3,2}$ holds when X is equivalent to $\mathsf{Fin}\ n$, it follows from the "finite choice" principle in Lemma 1. The particular instance of the axiom of choice used in the proof of Theorem 24 also holds when $X \simeq \mathsf{Fin}\ n$.

[2] We use 0-based indexing of h-levels, while [26] uses -2-based indexing, so our $\mathsf{AC}_{3,2}$ is their $\mathsf{AC}_{1,0}$.

7 Other Analytic Functors

The formulation FMSet of the finite bag functor exposes this as an analytic functor [15, 12], which differs from a polynomial functor in that the type of tuples $\mathsf{Fin}\ n \to X$ is quotiented by the relation induced by the action of the symmetric group on $\mathsf{Fin}\, n$. Other analytic functors arise by choosing a different subgroup of the symmetric group. For example, picking the subgroup of cyclic permutations delivers the functor of cyclic lists, while taking the trivial subgroup allows us to recover the list functor.

In type theory analytic functors can be seen as instances of the functors associated to the *quotient containers* of Abbott *et al.* [1]. A quotient container is a triple consisting of a type A, a family $B : A \to \mathsf{Type}$ and a propositionally-valued family $P : \forall\{a\}.\ B\, a \simeq B\, a \to \mathsf{Type}$ closed under identity, inverses and composition of equivalences. The associated functor is:

$$\mathsf{F}_{A,B,P}\ X =_{\mathsf{df}} \sum(a : A).\ (B\, a \to X)\ /_2\ \mathsf{Act}\ P\, a$$

where the relation $\mathsf{Act}\ P\, a$ is

$$\mathsf{Act}\ P\, a\ v\ w =_{\mathsf{df}} \exists(\sigma : B\, a \simeq B\, a).\ P\, \sigma \times (v = w \circ \sigma)$$

The type $\mathsf{F}_{A,B,P}\ X$ is a set whenever the type of shapes A is a set. The functor FMSet corresponds to the instance where $A =_{\mathsf{df}} \mathbb{N}$, $B =_{\mathsf{df}} \mathsf{Fin}$ and $P\, \sigma =_{\mathsf{df}} 1$.

We know that the construction of the final coalgebra as an ω-limit in the category of sets for a general analytic functor $\mathsf{F}_{A,B,P}$ is constructively problematic, since it is already problematic for FMSet. Nevertheless, one can ask if a result like Theorem 11 is valid for any $\mathsf{F}_{A,B,P}$. We do not know how to generally define a section for the function $\mathsf{pres}_{\mathsf{F}} : \mathsf{F}_{A,B,P}(\lim_n(\mathsf{F}^n_{A,B,P}\, 1)) \to \lim_n(\mathsf{F}^{1+n}_{A,B,P}\, 1)$. But we believe the surjectivity of $\mathsf{pres}_{\mathsf{F}}$ to be provable under the assumption of the axiom of countable choice. The proof of Theorem 11 relies on Propositions 4 and 5, which are very specific properties of the finite bag functor. The employment of these propositions can be seen as the invocation of two specific instances of the axiom of countable choice, which happen to hold in the case of FMSet.

Each quotient container (A, B, P) also specifies a polynomial functor $\mathsf{G}_{A,B,P}$ valued in groupoids, akin to the functor Bag. First, the small HIT construction of the groupoid of finite types Bij can be generalized:

$$\frac{a : A}{\mathsf{obj}\, a : \mathsf{U}_{A,B,P}} \qquad \frac{a : A \quad \alpha : Ba \simeq Ba \quad p : P\, \alpha}{\mathsf{hom}\ \alpha\, p : \mathsf{obj}\, a = \mathsf{obj}\, a}$$

$$\frac{a : \mathbb{N} \qquad \alpha, \beta : B\, a \simeq B\, a \qquad p : P\, \alpha \qquad q : P\, \beta}{\mathsf{hom}\, (\beta \circ \alpha)\, (\mathsf{Pcomp}\ p\, q) = \mathsf{hom}\ \alpha\, p \bullet \mathsf{hom}\ \beta\, q} \qquad \frac{}{\mathsf{isGroupoid}\ \mathsf{U}_{A,B,P}}$$

Above Pcomp is the closure of P with respect to composition of equivalences. It is possible to prove that hom also preserves identities and inverses. When B is valued in sets, there is a function $\langle - \rangle : \mathsf{U}_{A,B,P} \to \mathsf{Type}$ extracting a set, so that $\langle \mathsf{obj}\, a \rangle =_{\mathsf{df}} B\, a$. The functor $\mathsf{G}_{A,B,P}$ is

$$\mathsf{G}_{A,B,P}\, X =_{\mathsf{df}} \sum(x : \mathsf{U}_{A,B,P}).\ \langle x \rangle \to X$$

Since $\mathsf{G}_{A,B,P}$ is a polynomial functor, its final coalgebra can be constructed as an ω-limit, as in Theorem 21 and Corollary 22. We conjecture that the latter can be related to $\mathsf{F}_{A,B,P}$ similarly to how Bag and FMSet are related via Corollary 20: if B is injective, then $\|\mathsf{G}_{A,B,P}\, X\|_2 \simeq \mathsf{F}_{A,B,P}\, X$. Notice that Fin is an injective type family, which the proof of $\mathsf{Bij} \simeq \mathsf{FinSet}$ (Proposition 18) crucially depends on.

8 Conclusions

We looked at various definitions of the finite bag functor, valued in sets and in groupoids, and constructions of their final coalgebras. When working with set-based definitions, the set-theoretic constructions as ω-limit of the terminal chain in (3) and as quotient of the final List-coalgebra are not directly replicable in HoTT, since they imply the validity of classical principles like LLPO. We are at least able to salvage the weak preservation of the ω-limit. The situation is brighter when working with the groupoid-based definition. The latter is a polynomial functor, thus has the final coalgebra given by the ω-limit of the terminal chain.

Our conclusion is in line with the one of Kock [16] and Finster *et al.* [10]: the bag functor is better behaved when valued in groupoids instead of sets, especially from the perspective of final semantics. This seems to indicate that the denotational semantics of "resource-sensitive" computations is better performed using groupoids instead of sets (switching from categorical to bicategorical semantics). In particular, the syntax of process calculi such as CCS, or term calculi for linear logic, could be defined directly as a groupoid, *i.e.* structural congruences could be treated as data instead of property. We plan to properly investigate this connection to programming language semantics in future work, along the lines of [9]. For this endeavor, it will also be necessary to study the final coalgebra of combinations of the bag functor with other functors *e.g.* formalizing the presence of labels or actions in the transition relation.

Cubical Agda allows the definition of coinductive types with HITs appearing in the codomain of destructors. For example, it is possible to define the following coinductive record:

```
record cLim-FCM : Type where
  coinductive
  field
    unfold : FCM cLim-FCM
```

It is moreover possible to prove that this type is the final coalgebra of the set-based finite bag functor. The proof is similar to the one given for the finite powerset functor [29, Theorem 2]. Definitions such as `cLim-FCM` are an experimental feature of Cubical Agda, since the interaction of coinductive types and HITs has not yet been investigated (only for some M-types [30], which are definable internally in HoTT anyway). We believe that such definitions could be motivated by looking at recent work by Kristensen *et al.* [18], which seems to indicate that the final coalgebra of functors with action on objects given as a HIT, such as FCM, should be definable as the *strict* ω-limit of the chain in (3) in the cubical set model. Strictness means that the limit-property holds on the nose, not only up-to path equality.

References

1 Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Constructing polymorphic programs with quotient types. In Dexter Kozen and Carron Shankland, editors, *Proc. of 7th Int. Conf. on Mathematics of Program Construction, MPC'04*, volume 3125 of *LNCS*, pages 2–15. Springer, 2004. doi:10.1007/978-3-540-27764-4_2.

2 Jirí Adámek and Václav Koubek. On the greatest fixed point of a set functor. *Theoretical Computer Science*, 150(1):57–75, 1995. doi:10.1016/0304-3975(95)00011-K.

3 Benedikt Ahrens, Paolo Capriotti, and Régis Spadotti. Non-wellfounded trees in Homotopy Type Theory. In Thorsten Altenkirch, editor, *Proc. of 13th Int. Conf. on Typed Lambda Calculi and Applications, TLCA'15*, volume 38 of *LIPIcs*, pages 17–30. Schloss Dagstuhl, 2015. doi:10.4230/LIPICS.TLCA.2015.17.

4 Michael Barr. Terminal coalgebras in well-founded set theory. *Theoretical Computer Science*, 114(2):299–315, 1993. doi:10.1016/0304-3975(93)90076-6.

5 Gilles Barthe, Venanzio Capretta, and Olivier Pons. Setoids in type theory. *Journal of Functional Programming*, 13(2):261–293, 2003. doi:10.1017/S0956796802004501.

6 Douglas Bridges and Fred Richman. *Varieties of Constructive Mathematics*. Cambridge University Press, 1987. doi:10.1017/cbo9780511565663.

7 Paolo Capriotti, Nicolai Kraus, and Andrea Vezzosi. Functions out of higher truncations. In Stephan Kreutzer, editor, *Proc. of 24th EACSL Ann. Conf. on Computer Science Logic, CSL'15*, volume 41 of *Leibniz International Proceedings in Informatics*, pages 359–373. Schloss Dagstuhl, 2015. doi:10.4230/LIPICS.CSL.2015.359.

8 Vikraman Choudhury and Marcelo Fiore. Free commutative monoids in homotopy type theory. In *Proc. of 38th Conf. on Mathematical Foundations of Programming Semantics (MFPS XXXVIII)*, volume 1. Centre pour la Communication Scientifique Directe (CCSD), 2023. doi:10.46298/entics.10492.

9 Vikraman Choudhury, Jacek Karwowski, and Amr Sabry. Symmetries in reversible programming: from symmetric rig groupoids to reversible programming languages. *Proc. of the ACM on Programming Languages*, 6(POPL):1–32, 2022. doi:10.1145/3498667.

10 Eric Finster, Samuel Mimram, Maxime Lucas, and Thomas Seiller. A cartesian bicategory of polynomial functors in homotopy type theory. In Ana Sokolova, editor, *Proc. of 37th Conf. on Mathematical Foundations of Programming Semantics, MFPS'21*, volume 351 of *EPTCS*, pages 67–83, 2021. doi:10.4204/EPTCS.351.5.

11 Dan Frumin, Herman Geuvers, Léon Gondelman, and Niels van der Weide. Finite sets in homotopy type theory. In June Andronick and Amy P. Felty, editors, *Proc. of 7th ACM SIGPLAN Int. Conf. on Certified Programs and Proofs, CPP'18*, pages 201–214. ACM, 2018. doi:10.1145/3167085.

12 Ryu Hasegawa. Two applications of analytic functors. *Theoretical Computer Science*, 272(1-2):113–175, 2002. doi:10.1016/S0304-3975(00)00349-2.

13 Ichiro Hasuo, Kenta Cho, Toshiki Kataoka, and Bart Jacobs. Coinductive predicates and final sequences in a fibration. In Dexter Kozen and Michael W. Mislove, editors, *Proc. of the 29th Conf. on the Mathematical Foundations of Programming Semantics, MFPS'13*, volume 298 of *ENTCS*, pages 197–214. Elsevier, 2013. doi:10.1016/j.entcs.2013.09.014.

14 Bart Jacobs. *Introduction to Coalgebra: Towards Mathematics of States and Observation*, volume 59 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2016. doi:10.1017/CBO9781316823187.

15 André Joyal. Foncteurs analytiques et espèces de structures. In *Combinatoire énumérative*, pages 126–159. Springer Berlin Heidelberg, 1986. doi:10.1007/bfb0072514.

16 Joachim Kock. Data Types with Symmetries and Polynomial Functors over Groupoids. In Ulrich Berger and Michael Mislove, editors, *Proc. of 28th Conf. on Mathematical Foundations of Programming Semantics, MFPS'12*, volume 286 of *ENTCS*, pages 351–365. Elsevier, 2012. doi:10.1016/j.entcs.2013.01.001.

17 Nicolai Kraus, Martín Escardó, Thierry Coquand, and Thorsten Altenkirch. Notions of Anonymous Existence in Martin-Löf Type Theory. *Logical Methods in Computer Science*, 13(1), 2017. doi:10.23638/LMCS-13(1:15)2017.

18 Magnus Baunsgaard Kristensen, Rasmus Ejlers Møgelberg, and Andrea Vezzosi. Greatest HITs: Higher inductive types in coinductive definitions via induction under clocks. In Christel Baier and Dana Fisman, editors, *Proc. of 37th Ann. ACM/IEEE Symp. on Logic in Computer Science, LICS'22*, pages 42:1–42:13. ACM, 2022. doi:10.1145/3531130.3533359.

19 Paul Blain Levy. Similarity quotients as final coalgebras. In Martin Hofmann, editor, *Proc. of 14th Int. Conf on Foundations of Software Science and Computational Structures, FoSSaCS'11*, volume 6604 of *LNCS*, pages 27–41. Springer, 2011. doi:10.1007/978-3-642-19805-2_3.

20 Nuo Li. *Quotient types in type theory*. PhD thesis, University of Nottingham, UK, 2015. URL: http://eprints.nottingham.ac.uk/28941/.

21 Mark Mandelkern. Constructively complete finite sets. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 34(2):97–103, 1988. doi:10.1002/malq.19880340202.

22 Stefano Piceghello. *Coherence for Monoidal and Symmetric Monoidal Groupoids in Homotopy Type Theory*. PhD thesis, University of Bergen, Norway, 2021. URL: https://hdl.handle.net/11250/2830640.

23 Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, 2000. doi:10.1016/S0304-3975(00)00056-6.

24 Kristina Sojakova. *Higher Inductive Types as Homotopy-Initial Algebras*. PhD thesis, Carnegie Mellon University, USA, 2016. URL: http://reports-archive.adm.cs.cmu.edu/anon/anon/usr0/ftp/home/ftp/2016/CMU-CS-16-125.pdf.

25 The agda/cubical development team. The agda/cubical library, 2018. URL: https://github.com/agda/cubical/.

26 The Univalent Foundations Program. *Homotopy type theory: Univalent foundations of mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

27 Daniele Turi and Gordon D. Plotkin. Towards a mathematical operational semantics. In *Proc. of 12th Ann. IEEE Symp. on Logic in Computer Science, LICS'97*, pages 280–291. IEEE Computer Society, 1997. doi:10.1109/LICS.1997.614955.

28 Niccolò Veltri and Niels van der Weide. Constructing higher inductive types as groupoid quotients. *Logical Methods in Computer Science*, 17(2), 2021. doi:10.23638/LMCS-17(2:8)2021.

29 Niccolò Veltri. Type-theoretic constructions of the final coalgebra of the finite powerset functor. In Naoki Kobayashi, editor, *Proc. of 6th Int. Conf. on Formal Structures for Computation and Deduction, FSCD'21*, volume 195 of *LIPIcs*, pages 22:1–22:18. Schloss Dagstuhl, 2021. doi:10.4230/LIPICS.FSCD.2021.22.

30 Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: A dependently typed programming language with univalence and higher inductive types. *Proc. of the ACM on Programming Languages*, 3(ICFP):1–29, 2019. doi:10.1145/3341691.

31 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.

32 James Worrell. On the final sequence of a finitary set functor. *Theoretical Computer Science*, 338(1-3):184–199, 2005. doi:10.1016/j.tcs.2004.12.009.

33 Brent Yorgey. *Combinatorial Species and Labelled Structures*. PhD thesis, University of Pennsylvania, 2014. URL: http://ozark.hendrix.edu/~yorgey/pub/thesis.pdf.

34 Brent A. Yorgey. Species and functors and types, oh my! In Jeremy Gibbons, editor, *Proc. of 3rd ACM Symp. on Haskell, Haskell'10*, pages 147–158. ACM, 2010. doi:10.1145/1863523.1863542.

Proof Pearl: Faithful Computation and Extraction of μ-Recursive Algorithms in Coq

Dominique Larchey-Wendling
Université de Lorraine, CNRS, LORIA, Vandœuvre-lès-Nancy, France

Jean-François Monin
Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, France

Abstract

Basing on an original Coq implementation of unbounded linear search for *partially decidable* predicates, we study the computational contents of μ-recursive functions via their syntactic representation, and a correct by construction Coq interpreter for this abstract syntax. When this interpreter is extracted, we claim the resulting OCaml code to be the natural combination of the implementation of the μ-recursive schemes of composition, primitive recursion and unbounded minimization of *partial* (i.e., possibly non-terminating) functions. At the level of the fully specified Coq terms, this implies the representation of higher-order functions of which some of the arguments are themselves partial functions. We handle this issue using some techniques coming from the Braga method. Hence we get a faithful embedding of μ-recursive algorithms into Coq preserving not only their extensional meaning but also their intended computational behavior. We put a strong focus on the quality of the Coq artifact which is both self contained and with a line of code count of less than 1k in total.

2012 ACM Subject Classification Theory of computation → Models of computation; Theory of computation → Type theory; Theory of computation → Functional constructs; Software and its engineering → Formal methods; Software and its engineering → Functional languages; Theory of computation → Higher order logic

Keywords and phrases Unbounded linear search, μ-recursive functions, computational contents, Coq, extraction, OCaml

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.21

Supplementary Material *Software*: `https://github.com/DmxLarchey/Murec_Extraction` archived at `swh:1:dir:4d128568b56a17277c4f69ee1805e3910665f34f`

Funding *Dominique Larchey-Wendling*: partially supported by NARCO (ANR-21-CE48-0011).

1 Introduction

The theory of μ-recursive functions is a well established and widely used model for representing (partial) recursive functions of type $\mathbb{N}^k \rightharpoonup \mathbb{N}$ where $\mathbb{N}^k$ is the type of tuples of natural numbers of arity k. It originates from primitive recursive functions, invented in the 1920s in the Hilbert school (the modern denomination was coined by Rózsa Péter), which is the smallest class of functions containing constant functions, the successor function, projections (of the i-th argument), and closed under the schemes of composition and primitive recursion. Primitive recursive schemes define provably total functional relations but do not cover all the spectrum of computability, the Ackermann function giving the most popular counter-example.

Gödel defined the larger class of "general" recursive functions, developing ideas of Herbrand, and Kleene [7] later proposed to augment the allowed primitive recursive schemes with that of unbounded minimization of partial functions, giving the class of μ-recursive functions, equivalent (extensionally) to that of general recursive functions, and of which primitive recursive functions form a natural, strict sub-class.


© Dominique Larchey-Wendling and Jean-François Monin;
licensed under Creative Commons License CC-BY 4.0

```
(** val ra_compute : recalg → nat list → nat **)
let rec ra_compute s v =
  match s with
  | Ra_zero → O
  | Ra_succ → match v with y :: _ → S y
  | Ra_proj i → vec_prj v i
  | Ra_comp (s, s_vec) → ra_compute s (vec_map_compute (fun t → ra_compute t v) s_vec)
  | Ra_prec (s, s'') → match v with y :: u →
    prim_rec_compute (ra_compute s) (fun w n x → ra_compute s'' (n :: x :: w)) u y
  | Ra_umin s' → umin_compute (fun n → ra_compute s' (n :: v)) O
```

Figure 1 An OCaml interpreter for μ-algorithms.

In the context of mechanization, μ-recursive functions have been implemented and/or used in several projects [13, 9, 2, 10]. Most of these developments are concerned with computability theory (the S-n-m theorem, Rice's theorem, Kleene's normal form theorem, Hilbert's tenth problem) so they mainly focus on their extensional properties. On the contrary, in this proof pearl, we mostly focus on the intentional contents of μ-recursive schemes. We call these μ-algorithms and reserve the term "function" for the extensional notion.

Of course, μ-algorithms do not provide all the algorithmic means to compute values: for instance, they lack course-of-values recursion, nested recursion, higher-order primitive recursion, higher-order functions, and they are grounded on the very rudimentary datatype natural numbers, hence, when e.g. computing with lists or trees, one must pass through encoders and decoders. However, μ-recursive functions are universal in that they capture all computable functions and, in comparison with other models with the same power, they have the usual "referential transparency" advantage of functional languages over imperative ones, which *naturally* makes them better suited to equational and compositional reasoning than, say, Turing or Minsky (counter) machines.

In a contemporary approach, μ-algorithms can be described by an abstract syntax and a simple interpreter providing the computation rules to be followed by each construct. This abstract syntax can be implemented by the following type, where `Ra_comp` encodes composition, `Ra_prec` encodes primitive recursion, `Ra_umin` encodes minimization, and `Ra_zero`, `Ra_succ` and `Ra_proj` are obvious.

```
type nat = O | S of nat
type recalg = Ra_zero | Ra_succ | Ra_proj of nat | Ra_comp of recalg * recalg list |
              Ra_prec of recalg * recalg | Ra_umin of recalg
```

A natural OCaml program for interpreting pieces of code written in the above language is displayed in Figure 1. The arguments (s, s_{vec}) of `Ra_comp` stand respectively for (the source code of) a n-ary function and a n-tuple of functions, whose output is expected to be fed as inputs for (the interpreter of) s. The arguments (s, s'') of `Ra_prec` stand respectively for the result to be returned in the base case (the last argument is zero) and the function to be applied in the step case (the last argument is a successor). For convenience, the tuple $\langle x_1, \ldots, x_k \rangle$ of natural numbers representing the inputs of a μ-recursive algorithm is encoded by a list in the reverse order $[x_k ; \ldots ; x_1]$, so that the driving argument n for primitive recursion is the head of the list.

The functional implementations of μ-recursive schemes are straightforward: `vec_prj` for projections, `vec_map_compute` for compositions, `prim_rec_compute` for primitive recursion and `umin_compute` for μ-minimization.

```
(** val vec_prj : α list → nat → α **)
let rec vec_prj u i = match u with
  | x :: v → match i with O → x | S j → vec_prj v j

(** val vec_map_compute : (α → β) → α list → β list **)
let rec vec_map_compute f = function
| []     → []
| x :: v → f x :: vec_map_compute f v

(** val prim_rec_compute : (α → β) → (α → nat → β → β) → α → nat → β **)
let rec prim_rec_compute f g x = function
| O   → f x
| S n → g x n (prim_rec_compute f g x n)

(** val umin_compute : (nat → nat) → nat → nat **)
let rec umin_compute f n = match f n with
| O   → n
| S _ → umin_compute f (S n)
```

Though the above code is not very complicated, there are some subtle points making it a not-so-trivial case study if we want to prove its correctness w.r.t. a formal specification. In particular, as a purely functional piece of code, it seems reasonable to get it by extraction of a Coq proof. However, unbounded minimization or μ-minimization of a (partial) function $f : \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}$ consists in the (partial) function $\mu f : \mathbb{N} \rightharpoonup \mathbb{N}$ such that $\mu f(x)$ is the number n for which $f(n, x)$ is 0 and $f(i, x)$ is defined and not equal to 0 for every number i less than n. Although uniquely defined, it may not exist. For instance, μ-minimization of a constant (non-zero) function (e.g. $f(\cdot, \cdot) = 1$) is the nowhere defined function of type $\mathbb{N} \rightharpoonup \mathbb{N}$. Indeed, as soon as μ-minimization is available, *all* the other constructs are contaminated and encode possibly non-terminating functions. Altogether, not only `ra_compute` is clearly undefined on some inputs (where computation does not terminate), but it relies on higher-order programs such as `vec_map_compute` and `prim_rec_compute` which have to be themselves considered as non-terminating since they are applied to *possibly non-terminating arguments*, all in a nested recursive manner.

In a previous work [9], the first author showed that it is possible to derive a Coq term which computes the same result as a given μ-recursive function. The latter was first transformed, by bounding it using the folklore "fuel" technique, thus giving a primitive recursive (hence terminating) algorithm, and then applying Constructive Epsilon, i.e. unbounded minimization of inhabited and decidable predicates over $\mathbb{N}$. Kleene used a comparable trick in order to establish that every μ-recursive function can be obtained from the μ-minimization of some primitive recursive function, i.e. the normal form theorem [8]. In his section "resource bounded evaluation," Carneiro [2] uses the same fuel trick in Lean to approximate partial computable functions with primitive recursive ones, combined with a variant of Constructive Epsilon he calls find.

Though this approach is sufficient for theoretical purposes such as studying the expressive power of computational models,[1] it is unsatisfactory from an algorithmic point of view: the underlying calculation boils down to a systematic and heavy trial-and-error process that

[1] Most of the textbook presentations of μ-recursive functions like e.g. [12, 1] focus on their extensional meaning as set-theoretic partial functions, i.e., the relation between inputs and outputs, or so called graph, but not on the calculations performed.

is unfaithful to the intended behavior, unlike the intuitive OCaml code above. Directly reasoning on the latter with extraction in mind is actually more demanding: we need to express the computational counterpart of the desired specific program in a type-theoretic framework where only total functions are allowed. In other words, we need to provide "generic" partial termination certificates for (the Coq counterpart of) the above OCaml functions.

We show here how the Braga method [11], our trick to manage termination issues in recursive programs, can be adapted in the present case study in order to satisfy the above requirements. In a nutshell, the Braga method allows us to define functions of x whose termination is provided by an additional domain argument $d : D\,x$, of sort `Prop`, where D is inductively defined and systematically derived from the shape of the desired program `Pgm` to be extracted. In this way we can express a functional program

$$\texttt{Fixpoint}\ g\ x\ (d : D\,x)\ \{\texttt{struct}\ d\} :=\ \ldots\ g\,x'\,d'\ \ldots$$

where $d' : D\,x'$ is structurally smaller than d – the *guard condition* which ensures termination. Then we can reason on the partial correctness properties of g before (and without) bothering about termination (e.g., without defining a measure which, anyway, could not exist in all cases). The domain D is typically the domain of an inductive input-output graph G, which is nothing but a relational presentation of `Pgm` and can be seen as a complete characterization of `Pgm`. In [11], G is also used in the Σ-type of $g : \forall x,\ D\,x \to \{y \mid G\,x\,y\}$ so that, after erasure, the extracted OCaml program automatically satisfies this characterization as well as the partial correctness properties which are derived from it.

In the case of μ-algorithms we need to go further than [11]. First, partial functions are essential here whereas almost all examples considered in [11] turn out to be total functions. Second, we already have a natural input-output graph: a relational semantics for the abstract syntax of μ-algorithms. However, this semantics is naturally expressed as a combination of partial functions, one for each construct of μ-algorithms. In order to mimic the original formulation, we encode partial functions from $\mathbb{N}^k$ to $\mathbb{N}$ by total functions from $\mathbb{N}^k$ to $\mathbb{N} \to$ `Prop`, yielding a compact and crystal-clear specification. This semantics is detailed in section 4.2 and noted $[\![.]\!]$. Looking back at the intended program above, we need to extend the Braga method to get Coq programs that: first, combine in a similar and hopefully modular way an implementation of μ-minimization with "ordinary looking" structural recursion on data structures such as `nat` or the syntax of μ-algorithms; and second, are driven by the domain of an input-output graph G, such as $[\![.]\!]$, that is not necessarily expressed inductively. A suitable general type scheme for such programs is simply $\forall x,\ (\exists y, G\,x\,y) \to \{y \mid G\,x\,y\}$, which can be implemented either by ordinary structural recursion on the first input x, or by using the Braga method on a termination certificate derived from the second input $(\exists y, G\,x\,y)$.

Additionally, μ-minimization provides another opportunity to change the Braga machinery a little bit: for `umin_compute`, which is basically a tail-recursive presentation of an (unbounded) loop for linear search, the propagation of assertions is not encoded backwards by a Σ-type embedding of the expected input-output relation, but forwards by a parameter containing an inductive invariant. This makes the enriched Coq program already tail-recursive and, more importantly, proofs of propagation become simpler.

Remarks about the Coq code. The artifact we publicly deliver only requires a minimal amount of Coq machinery to implement linear search and the interpreter for μ-recursive algorithms. To illustrate this and also produce self-contained code, we do not use the Coq standard library, except for the modules `Utf8` to allow for better human readable Coq code, and `Extraction` to witness our claims about the faithfulness of the extracted code to the

"natural" OCaml interpreter. The code is intended to be read as an essential part of this pearl, and we invite the reader to consult the associated artifact, starting with the `README.md` file. We tried to make the artifact readable by a human, without the help of the type-checker. This means that proofs written with scripts (`Ltac`) are both very short, and with only light automation not beyond `trivial` or `easy`. When the contents of terms is critical, e.g. when it contributes to extraction, or in order to visualize structural decrease, we write these as λ-terms.

Contributions. We hope our contributions to be somewhat valuable for people using Coq or a similar Type Theory as a programming language, but nothing original is claimed about computability theory. First of all, we provide a short, clean, readable and (hopefully) informative, Coq implementation of the partial linear search algorithm, extending Constructive Epsilon to partially decidable predicates, using a variant of the Braga method that also fully takes into account the tail recursivity of the underlying program. Second, we contribute a Coq interpreter for μ-recursive algorithms, which follows their intended (functional) operational semantics, taking [1] as reference, as witnessed by a neat extraction to OCaml. Third, this interpreter relies on linear search in a way that illustrates a general approach to integrating programs written with the Braga method (or variants of it) and structural recursive programs that depend on each other in a nested or mutual recursive way. With this reasonably sized and documented code, we also hope to popularize further some dependent inversion techniques that sometimes hinder the usage of Coq structural fixpoints. For instance, see our small library for dependent vectors that features improvements on the standard library and is of independent interest.

This proof pearl is constructed as follows. Section 2 provides the prerequisites needed to understand the paper and the associated Coq artifact. Section 3 explains how to generalize the specification of the linear search algorithm to be able to search with partially decidable predicates. This generalization of Constructive Epsilon is the essential ingredient to implement the scheme of μ-minimization. Section 4 presents the Coq implementation of μ-algorithms with their extensional semantics, following [1], and then their intentional contents as a Coq term, an interpreter computing the output, if provided with a proof of termination on the given input. Section 5 presents the result of the extraction of the above mentioned interpreter as an OCaml program. We list the various tweaks that help at completely rendering a readable (and herein presentable) program. We also explain how to get rid of some possibly unwanted OCaml tricks used by the extraction plugin to circumvent the limitations of the OCaml type-system, as compared to that of Coq, the source type theory.

2 Type-theoretic basics and notations

We present this pearl in the language of the type theory of Coq containing the sort `Prop`, the impredicative type of propositions, and the sort `Set`, the predicative type at the ground-level of sorts in `Type`, the predicative type hierarchy above both `Prop` and `Set`.

The basic inductive structures we use are the propositions (`True : Prop := I`) and (`False : Prop :=` .), the data types `unit : Set := tt`, first-order logic connectives and Peano natural numbers `nat : Set := O | S (_ : nat)`, endowed with natural ($\leq$) and strict ($<$) orders. Only a minimal amount of basic arithmetic results are needed, for which we give tiny proofs in `arith_mini.v`. We won't use lists but vectors instead, see Section 4.4.

We will use tuples (n-ary products) and dependent pairs (Σ-types) that come under various forms in Coq, see files `sigma.v`, `relations.v` and `vec.v`. We will write e.g. $\langle a, b, c\rangle$ for a triple of three values that may be proofs of propositions, hence giving of a proof of say $A \wedge B \wedge C$, or else terms giving a value in the product type $A \times B \times C$.

Given a predicate $P : X \to \texttt{Prop}$, we write dependent pairs as $\langle\langle x, p\rangle\rangle$ where p is a proof of $P\,x$ (hence dependent on x). In this paper, depending on the context (`Prop` vs. `Type`), $\langle\langle x, p\rangle\rangle$ denotes either a proof of the proposition $\exists x, P\,x$: `Prop`, also written $\texttt{ex}\,P$, or an inhabitant of the type $\{x \mid P\,x\} : \texttt{Type}$, also written $\texttt{sig}\,P$. Likewise, given $d := \langle\langle x, p\rangle\rangle$, we write $\pi_1\,d := x$ and $\pi_2\,d := p$ for the projections of the dependent pair d.

In the file `relations.v`, we also give basic tools to manipulate 0-ary, unary and binary relations (i.e. predicates), like notations for composition, inclusion, conjunction. A unary relation $P : X \to \texttt{Prop}$ is *functional (or deterministic)* if there is at most one x s.t. $P\,x$.

3 Unbounded linear search in Coq

The linear search algorithm on an unbounded interval of $\mathbb{N}$ (LS for short) is the main engine of μ-minimization. As one of the simplest examples of a program which computes a *partial* (recursive) function, it is particularly interesting. It can be specified as follows. Given a decidable predicate P on $\mathbb{N}$ and a starting number $s : \mathbb{N}$, find an $n : \mathbb{N}$ greater or equal to s such that $P(n)$. Among its many other applications, we can cite Constructive Epsilon, which corresponds to the special case where $s = 0$. More precisely, it realizes the specification $\texttt{ex}\ P \to \texttt{sig}\ P$, a short hand for $(\exists x, P\ x) \to \{n \mid P\ n\}$. As we reuse some ideas coming from `ConstructiveEpsilon.v` of the Coq standard library, the name "Constructive Epsilon" will below refer to this implementation.

Assuming a suitable program `test`, here is an obvious OCaml program for unbounded linear search:

```
let rec loop s = if test s then s else loop (s + 1)
```

Let us informally write $\texttt{Gr\_than}_P(s)$ for the set of natural numbers x such that $x \geq s$ and $P\,x$ holds. If $\texttt{Gr\_than}_P(s)$ is inhabited, the returned value is actually the *least* value m in $\texttt{Gr\_than}_P(s)$; but otherwise, the algorithm loops forever. The underlying function is then clearly *partial*. In the following we discuss the contents of file `linear_search.v`.

3.1 Specification of linear search

Aiming first at a general Coq specification of the unbounded linear search algorithm, we actually don't need to assume that P is decidable on `nat`, but only between s and a large enough number. For additional generality and convenience we also consider two predicates $D_{\texttt{test}}$ (the domain of `test`) and Q such that, whenever $D_{\texttt{test}}$ holds, P or Q can be decided and P and Q cannot hold together.

$$\texttt{test} : \forall n,\ D_{\texttt{test}}\ n \to \{P\ n\} + \{Q\ n\} \qquad \texttt{PQ\_abs} : \forall n,\ D_{\texttt{test}}\ n \to P\ n \to Q\ n \to \texttt{False}$$

We then only assume that $D_{\texttt{test}}$ holds between s and a large enough number. On the side of the post-condition, we see that not only $D_{\texttt{test}}$ and P hold at the returned value m, if any, but also that $s \leq m$ and that $D_{\texttt{test}}$ and Q hold at all k such that $s \leq k < m$. Defining

$$\texttt{Definition btwn}\ (A : \texttt{nat} \to \texttt{Prop})\ n\ m := n \leq m \wedge (\forall k,\ n \leq k < m \to A\ k) \tag{1}$$

and with the notation $A \wedge_1 B := \lambda n,\ A\,n \wedge B\,n$, the strongest post-condition characterizing the output of LS is then:

$$\texttt{Definition}\ \mathbb{P}\texttt{ost}_{\texttt{ls}}\ s\ x := (D_{\texttt{test}}\ x \wedge P\ x) \wedge \texttt{btwn}\ (D_{\texttt{test}} \wedge_1 Q)\ s\ x.$$

On the side of the pre-condition of LS, we assume the existence of some x in $\texttt{Gr\_than}_P(s)$ and the ability to perform `test` from s to x. It can be stated using the following predicate.

```
Definition ℙre_ls s x := (D_test x ∧ P x) ∧ btwn D_test s x.
```

The expected type for linear search (starting at s) is then $(\exists x,\ \mathbb{P}\texttt{re}_{\texttt{ls}}\ s\ x) \to \{m \mid \mathbb{P}\texttt{ost}_{\texttt{ls}}\ s\ m\}$.

3.2 Termination of linear search

Observe that the witness x mentioned in the pre-condition is not used in the search loop – it is not available at the informative level. Moreover, `btwn` $Q\ s\ x$ is not assumed, that is, x is not necessarily minimal in $\texttt{Gr\_than}_P(s)$. But x can be used to compute a *termination certificate* since its very existence guarantees that the search loop eventually halts. The usual argument, in an imperative setting, consists in proving that $x - s$ is a loop variant. However, as mentioned in the introduction, we can take advantage of an essential feature of type theory of Coq to provide a direct inductive characterization of termination of sort `Prop` called $\mathbb{D}\texttt{ls}$, to be used as follows: `Fixpoint loop` $n\ (d : \mathbb{D}\texttt{ls}\ n)\ \{\texttt{struct}\ d\} := \ldots$ Only n can be used in the informative part of the computation; on the other hand, a strict sub-term of d has to be provided in any recursive call. Consistently, d is erased at extraction time. The design of $\mathbb{D}\texttt{ls}$ follows the pattern of recursive calls of the target program and keeps track of the information resulting from the tests carried out. Here is our definition of $\mathbb{D}\texttt{ls} : \texttt{nat} \to \texttt{Prop}$:

$$\frac{c : D_{\texttt{test}}\ n \wedge P\ n}{\mathbb{D}\texttt{ls\_stop}\ c : \mathbb{D}\texttt{ls}\ n} \qquad \frac{t : D_{\texttt{test}}\ n \qquad d_{\texttt{S}} : \mathbb{D}\texttt{ls}\,(\texttt{S}\ n)}{\mathbb{D}\texttt{ls\_next}\ t\ d_{\texttt{S}} : \mathbb{D}\texttt{ls}\ n}$$

Defining $\mathbb{D}\texttt{ls\_}\pi_1\ d$ as the immediate $D_{\texttt{test}}\,n$ component of $d : \mathbb{D}\texttt{ls}\,n$ using an easy pattern-matching, and $\mathbb{D}\texttt{ls\_}\pi_2\ d\ q$ as the immediate $\mathbb{D}\texttt{ls}\,(\texttt{S}\ n)$ component of d when $q : Q\ n$, using a more subtle pattern-matching, a version of `loop` returning a simple natural number is

```
Fixpoint loop n (d : 𝔻ls n) {struct d} : nat :=
  match test n (𝔻ls_π₁ d) with
  | left p  ⇒ n
  | right q ⇒ loop (S n) (𝔻ls_π₂ d q)
  end.
```

The second projection $\mathbb{D}\texttt{ls\_}\pi_2\,d$ returns a λ-term of type $Q\,n \to \mathbb{D}\texttt{ls}\,(\texttt{S}\ n)$ for each constructor. In the easy (and "intended") case where d is $\mathbb{D}\texttt{ls\_next}\ t\ d_{\texttt{S}}$, it just returns $\lambda\,\_,\ d_{\texttt{S}}$. And when d is $\mathbb{D}\texttt{ls\_stop}\ \langle t, p\rangle$, then t, p and q conspire with `PQ_abs` to construct a proof of the empty type `False`, on which an additional pattern matching provides a strict sub-term of any proof (of any inductive predicate).

```
Definition 𝔻ls_π₂ {n} (d : 𝔻ls n) : Q n → 𝔻ls (S n) :=
  match d with
  | 𝔻ls_stop ⟨t, p⟩ ⇒ λ q, match PQ_abs t p q with end
  | 𝔻ls_next _ d_S ⇒ λ _, d_S
  end.
```

The braces around the first parameter $\{n\}$ mark an *implicit* parameter, and the Coq code of $\mathbb{D}\texttt{ls\_}\pi_2$ witnesses structural decrease in a way suitable for a human to check at first glance.

3.3 Building an initial termination certificate

A termination certificate d for s can be computed (in the hidden realm of propositions and proofs) from the existence of x such that $\mathbb{P}\texttt{re}_{\texttt{ls}}\, s\, x$. It is easy to perform from a suitable inductive characterization of `btwn` but, in order to stick to its arithmetical definition (1), we simulate the two constructors and a special induction principle for `btwn` by proving:

$\texttt{btwn}_{\texttt{refl}}\ \{A\ n\} : \texttt{btwn}\ A\ n\ n$
$\texttt{btwn}_{\texttt{next}}\ \{A\ n\ m\} : \texttt{btwn}\ A\ n\ m \to A\ m \to \texttt{btwn}\ A\ n\ (\texttt{S}\ m)$
$\texttt{btwn}_{\texttt{ind}}\ A\ a\ b : \texttt{btwn}\ \big(\lambda n,\ A\ (\texttt{S}\ n) \to A\ n\big)\ a\ b \to A\ b \to A\ a.$

See the file `between.v` for the code of `btwn` and its tools that we use here to get

$\texttt{Lemma}\ \mathbb{P}\texttt{re}_{\texttt{ls}}\_\mathbb{D}\texttt{ls}\ \{s\} : (\exists x,\ \mathbb{P}\texttt{re}_{\texttt{ls}}\ s\ x) \to \mathbb{D}\texttt{ls}\ s$

by applying $\texttt{btwn}_{\texttt{ind}}$ to $\mathbb{D}\texttt{ls}$ and conclude with the monotonicity of `btwn`.

3.4 A tail-recursive program for the full loop

In order to prove that the output of the previous version of `loop` satisfies the desired post-condition, we could promote its type from `nat` to $\{m : \texttt{nat} \mid \mathbb{P}\texttt{ost}_{\texttt{ls}}\ s\ m\}$. This would lead to a decomposition of the result of the recursive call into a pair $\langle\!\langle m, po \rangle\!\rangle$, where m is the found value and po the associated proof of post-condition, followed by the construction of a similar pair $\langle\!\langle m, po' \rangle\!\rangle$, only different on its proof component, to be returned. A more elegant way is to proceed with proofs as with data in functional programming, when mimicking `while` loops using recursivity and accumulators. A remarkable point is that in the proofs-as-programs paradigm, proof accumulators turn out to be (proofs of) *loop invariants*, as illustrated below by b becoming $\texttt{btwn}_{\texttt{next}}\ b\ \langle t, q \rangle$ in the recursive call.

In more detail, we first fix a starting value s and take as invariant $\texttt{btwn}\ (D_{\texttt{test}} \wedge_1 Q)\ s\ n$. The linear search algorithm then calls `loop` with s as the initial input value for n, $(\mathbb{P}\texttt{re}_{\texttt{ls}}\_\mathbb{D}\texttt{ls}\ e)$ as the initial termination certificate, where e is a proof of $(\exists n,\ \mathbb{P}\texttt{re}_{\texttt{ls}}\ s\, n)$, and $\texttt{btwn}_{\texttt{refl}}$ as the initial (proof of the) invariant. In the course of the loop, we assume a proof of the invariant for n named b; the proof of $D_{\texttt{test}}\ n$ derived from d is first bound to t; in the recursive call, the (proof of the) invariant for $\texttt{S}\ n$ is derived from b, t and $q : Q\ n$ using $\texttt{btwn}_{\texttt{next}}$; and finally, when the test provides a proof $p : P\ n$, the desired proof of $\mathbb{P}\texttt{ost}_{\texttt{ls}}\ s\ n$ is just $\langle t, p \rangle$ paired with the invariant b. Altogether, the extended code of LS is rather short.

```
Fixpoint loop n (d : DDls n) (b : btwn (D_test ∧1 Q) s n) : {m | Post_ls s m} :=
  let t := DDls_π1 d in
  match test n t with
  | left p  ⇒ ⟪n, ⟨t, p, b⟩⟫
  | right q ⇒ loop (S n) (DDls_π2 d q) (btwn_next b ⟨t, q⟩)
  end.
Definition linear_search : (∃x, Pre_ls s x) → {m | Post_ls s m} :=
  λ e, loop s (Pre_ls_DDls e) btwn_refl.
```

3.5 From linear search to μ-minimization

To use the above linear search program, we only have to instantiate P, Q, $D_{\texttt{test}}$ and the `test` function, to provide a corresponding proof of `PQ_abs`, and possibly to add stubs to adapt the pre- and post-conditions. For instance in the case of Constructive Epsilon, we are

given an arbitrary predicate P on `nat`, with the hypothesis `P_dec` : $\forall n, \{P\ n\} + \{\neg P\ n\}$. Then we keep P and take $\neg P$ for Q, $(\lambda \_, \texttt{True})$ for $D_{\texttt{test}}$ and $\lambda n\, \_, \texttt{P\_dec}\ n$ for `test`; the proof of `PQ_abs` is trivial.

The case of μ-minimization is more interesting, in particular, we have a non-trivial instantiation for $D_{\texttt{test}}$. We are given a functional relation F : `nat` $\to$ `nat` $\to$ `Prop` such that, if n is in the domain of F, then the y such that $F\,n\,y$ can be computed by a program $f : \forall n, \texttt{ex}\,(F\ n) \to \texttt{sig}\,(F\ n)$. We want to transfer this to the minimization of F, that is, assuming the existence of a minimal m such that F is defined and strictly positive for all $x < m$, and $F\ m\ \texttt{O}$, we want to provide a computation returning this (unique) m. Formally, we define:

$$\texttt{def\_at}\ F := \lambda n, \exists y, F\,n\,y \qquad \texttt{ze\_at}\ F := \lambda n, F\,n\,\texttt{O} \qquad \texttt{pos\_at}\ F := \lambda n, \exists k, F\,n\,(\texttt{S}\,k)$$
$$\texttt{umin}_0\ F\ y := \texttt{ze\_at}\ F\ y \wedge \forall n, n < y \to \texttt{pos\_at}\ F\ n$$

The formal specification of μ-minimization is then $\texttt{ex}\ (\texttt{umin}_0\ F) \to \texttt{sig}\ (\texttt{umin}_0\ F)$. As a natural and cheap generalization, we also define $\texttt{umin}\ F\ s\ y := \texttt{ze\_at}\ F\ y \wedge \texttt{btwn}\ (\texttt{pos\_at}\,F)\ s\ y$. Then μ-minimization is the special case, with $s := 0$, of

$$\forall s, \texttt{ex}\ (\texttt{umin}\ F\ s) \to \texttt{sig}\ (\texttt{umin}\ F\ s) \tag{2}$$

This is very close to the specification of linear search, with $D_{\texttt{test}}$ instantiated as $\texttt{def\_at}\,F$, P as $\texttt{ze\_at}\,F$ and Q as $\texttt{pos\_at}\,F$ respectively. In order to use its implementation given in Section 3.4, we just have to feed it with a proof of `PQ_abs` (simple, using functionality of F and discrimination between `O` and `S` _) and a suitable `test` program:

```
Let test n (t : D_test n) : {P n} + {Q n} :=
  let (k, e_k) := f n t in
  match k return F _ k → _ with O ⇒ λ e, left e | S r ⇒ λ e, right ⟪r, e⟫ end e_k
```

where the term $e_k : F\,n\,k$ is analyzed as either $e : F\,n\,\texttt{O}$ (when k matches `O`) or $e : F\,n\,(\texttt{S}\,r)$ (when k matches $\texttt{S}\,r$).

However, the code obtained is this way is somewhat unsatisfactory, because each call to `loop` first constructs a dependent Boolean from the input and next immediately performs a pattern matching on the latter, and this intermediate Boolean would be reflected at extraction stage. A simple way to improve this state of affairs consists in performing a program transformation on the Coq `loop`, taking advantage, in passing, of $P \subseteq D_{\texttt{test}}$ and $Q \subseteq D_{\texttt{test}}$. Those transformations have only an impact on the loop (6 lines of code):

```
Fixpoint loop n (d : 𝔻ls D_test P n) (b : btwn Q s n) : sig (umin F s) :=
  let (k, e_k) := f n (𝔻ls_π1 D_test P d) in
  match k return F _ k → _ with
  | O   ⇒ λ e, ⟪n, ⟨e, b⟩⟫
  | S _ ⇒ λ e, loop (S n) (𝔻ls_π2 D_test … d ⟪_, e⟫) (btwn_next b ⟪_, e⟫)
  end e_k.
```

and the code of `linear_search` can be reproduced as is.

In conclusion to the file `umin_compute.v`, the program $\texttt{umin}_0\texttt{\_compute}$, which is needed in our full development of μ-recursive algorithms, is defined in two lines by functional composition of `linear_search` and simple monotonicity lemmas relating the various statements of `umin`. The reader can consult the file `umin_compute_details.v` to get a gradual derivation of the above `loop` from the one used in the original `linear_search`. As a side remark, it is

possible to present μ-minimization as just an instance of a parametric version of linear search, allowing us to share not only the $\mathbb{D}$`ls` predicate but also the code of the loop whatever the type of the result of the `test` function and the resulting extracted program. But, as symptomatic to many generic code constructions, it is unfortunately not yet as short and reasonably explainable as the above compromise between code sharing and code readability, where we transform a small part of the generic code to get μ-minimization.

4 Representing μ-algorithms in Coq

Basing on our implementation of μ-minimization, we now switch to the intentional encoding of all μ-recursive algorithms in Coq. We follow the idea already developed in [9, 10] of capturing the syntax of μ-algorithms in a type dependent on the arity (number of parameters of the corresponding function), and nested with a parametric type of dependent vectors. However, in this pearl, we insist on giving a minimized and as clean as possible account of these types.

We do not use lists at all. Instead, for a given base type X, we use *vectors* of type denoted $\texttt{vec}\,X\,n$, which are lists but augmented with a further dependency on their length, herein denoted using $n : \texttt{nat}$. Components of vectors are accessed through indices in the finite type $\texttt{idx}\,n : \texttt{Set}$ also dependent on n. In the sequel, $\langle\,\rangle$ denotes the empty vector, $a :: v$ denotes the vector made of a followed by v, and $v.[i]$ denotes the ith element of v. We also might write $\langle a; b; c\rangle$ for the vector $a :: b :: c :: \langle\,\rangle$. Our files `index.v` and `vec.v` reproduce the types `Fin.t` and `Vector.t` of the Coq standard library but are better tailored towards clean extraction. Technical details on our library are postponed to 4.4.

4.1 μ-algorithms as a nested dependent type

Usually leaving arities (denoted using the letters a, b) as implicit arguments of Coq constructors or terms, we define the type of μ-algorithm of arity a as $\texttt{recalg}\,a$ or simply $\mathcal{A}_a : \texttt{Set}$ herein, with the following constructors (inductive rules):

$$\frac{}{\texttt{ra\_zero} : \mathcal{A}_0} \qquad \frac{}{\texttt{ra\_succ} : \mathcal{A}_1} \qquad \frac{i : \texttt{idx}\,a}{\texttt{ra\_proj}\,i : \mathcal{A}_a}$$

$$\frac{f : \mathcal{A}_b \qquad g : \texttt{vec}\,\mathcal{A}_a\,b}{\texttt{ra\_comp}\,f\,g : \mathcal{A}_a} \qquad \frac{f : \mathcal{A}_a \qquad g : \mathcal{A}_{2+a}}{\texttt{ra\_prec}\,f\,g : \mathcal{A}_{1+a}} \qquad \frac{f : \mathcal{A}_{1+a}}{\texttt{ra\_umin}\,f : \mathcal{A}_a}$$

This inductive data-structure that we define in `recalg.v` mimics that in [9] but we use slightly different schemes to better match those of [1] that serve as our textbook reference here. For instance, we do not have constants (of arity 0) except for the zero constant itself which appears both at arity 0 and at arity 1 in [9].

The constructor $\texttt{ra\_comp}\,\_\,\_$ for *composition* nests the type $\mathcal{A}\_$ with the type of vectors $\texttt{vec}\,\_\,\_$, hence Coq fails to automatically derive a powerful enough eliminator for that nested inductive type. For completeness or further extensions, we provide a hand written general eliminator `recalg_rect` for $\mathcal{A}_a$ in the file `recalg.v` (as a suitable fixpoint), but we will not need it in this pearl since we will always reason or compute inductively on $\mathcal{A}_a$ using hand-written fixpoints, i.e. by inlining `recalg_rect`.

4.2 The semantics of μ-algorithms

We characterize the semantics $[\![S_a]\!]\ v_a\ o$ of the μ-algorithm of $S_a : \mathcal{A}_a$ by interpreting it as a binary relation between an input vector $v_a : \texttt{vec}\,\texttt{nat}\,a$ and an output value in $o : \texttt{nat}$, hence for $\mathcal{IO}_a := \texttt{vec}\,\texttt{nat}\,a \to \texttt{nat} \to \texttt{Prop}$, we seek to define $[\![S_a]\!] : \mathcal{IO}_a$. The denotation S_a is intended

to recall that this is *S(ource code) for arity a*. Notice that this provides an extensional meaning to S_a that restricts the possible algorithmic interpretations (or intentional meaning) of S_a which must realize that specification. To do so, we define $\texttt{ra\_sem} : \forall\{a\}, \mathcal{A}_a \to \mathcal{IO}_a$ as a fixpoint where we denote $[\![S_a]\!] := \texttt{ra\_sem}\, S_a$:

```
Fixpoint ra_sem {a} (S_a : A_a) : IO_a :=
  match S_a with
  | ra_zero   ⇒ Zr       | ra_comp S_b S_ab  ⇒ Cn [[S_b]] (vec_map [[·]] S_ab)
  | ra_succ   ⇒ Sc       | ra_prec S_a S_a'' ⇒ Pr [[S_a]] [[S_a'']]
  | ra_proj i ⇒ Id i      | ra_umin S_a'      ⇒ Mn [[S_a']]
  end where [[S_a]] := (ra_sem S_a).
```

Notice the nesting of `vec_map` which applies `ra_sem`, to every component of the vector $S_{ab} : \texttt{vec}\, \mathcal{A}_a\, b$, and the references to `Zr`, `Sc`, `Id`, `Cn`, `Pr`, `Mn` which correspond to the Coq encoding of μ-recursive schemes as defined in [1]. The fixpoint proceeds by structural induction on S_a but the guard-checker *inspects the code* of the nested instance of `vec_map` to ensure $[\![\cdot]\!]$ is called only on sub-terms of the vector S_{ab}. Notice that since the inductive type $\mathcal{A}_a$ nests the type of vectors $\texttt{vec}\, \mathcal{A}_a\, b$ in the constructor $\texttt{ra\_comp}\ \_\ \_$, the only way to traverse such structures is via nested fixpoints which, if properly written, can fortunately be accepted by the guard condition of the type-checker.

As a side note, instead of a direct fixpoint, we could use the general recursor `recalg_rect` (see file `recalg.v`), but this would have unfortunate consequences: it would hinder a unified presentation of `ra_sem` and `ra_compute`. Indeed, the induction hypothesis for the `ra_comp` constructor does not expect a vector but a (dependent) map, and would thus be incompatible with the output type of `vec_map`, hence involving some glue code. That glue code would also be necessary in the upcoming fixpoint `ra_compute` in section 4.3, and would there unfortunately reflect in the extracted OCaml code. To get a unified presentation of `ra_sem` and `ra_compute`, we choose to inline `recalg_rect` in both cases.

We follow precisely our reference textbook [1, p. 63] using reversely ordered vectors to represent tuples. See file `recalg_semantics.v` where we define

```
Definition Zr : IO_0 := λ _ y, y = 0.
Definition Sc : IO_1 := λ v_1 y, y = 1 + vec_head v_1.
Definition Id {a} (i : idx a) : IO_a := λ v_a y, y = v_a.[i].
```

We follow up with composition of a b-ary μ-algorithm with a vector of a-ary μ-algorithms:

```
Definition Cn {a b} (φ_b : IO_b) (ψ_ab : vec IO_a b) : IO_a :=
          λ v_a y, ∃v_b, φ_b v_b y ∧ ∀i, ψ_ab.[i] v_a v_b.[i]
```

to be found in [1, p. 64]. Primitive recursion is mechanically best described using a higher-order primitive recursive scheme (like that of e.g. Gödel system T). We define

```
Definition Pr {a} (φ_a : IO_a) (ψ_a'' : IO_2+a) : IO_1+a :=
          vec_S_inv (λ n v_a, prim_rec φ_a (λ w_a m y, ψ_a'' (m :: y :: w_a)) v_a n)
```

where `prim_rec` is defined in `schemes.v` as the following instance `nat_rect`, the dependent eliminator/recursor for the `nat` type and $\diamond$ denotes the right-associative composition of a binary relation with a unary one:

```
Context {X Y : Type} (F : X → Y → Prop) (G : X → nat → Y → Y → Prop).
Definition prim_rec (x_0 : X) := nat_rect (λ _, Y → Prop) (F x_0) (λ n p, (G x_0 n) ◇ p).
```

Informally, this would read as `prim_rec` $x_0\ n := \big(G\,x_0\,(n-1)\big) \diamond \cdots \diamond (G\,x_0\,0) \diamond (F\,x_0)$. We check that `Pr` satisfies the following two definitional equations, which correspond to the characterization of primitive recursive scheme in [1, p. 67].

1. `Pr` $\varphi_a\ \psi_{a''}\ (\mathtt{O} :: v_a)\ y = \varphi_a\ v_a\ y$;
2. `Pr` $\varphi_a\ \psi_{a''}\ (\mathtt{S}\,n :: v_a)\ y = \exists o,$ `Pr` $\varphi_a\ \psi_{a''}\ (n :: v_a)\ o \wedge \psi_{a''}\ (n :: o :: v_a)\ y$.

We finish with the scheme of unbounded minimization (starting at `O`), defined via a more general scheme of minimization starting at a value given as extra parameter:

```
Definition Mn {a} (φ_{a'} : 𝓘𝓞_{1+a}) : 𝓘𝓞_a := λ v_a, umin_0 (λ y, φ_{a'}(y :: v_a)).
```

The definitions of `umin` and `umin`$_0$ occur in the file `schemes.v` and are also discussed in Section 3.5. We check that `Mn` satisfies the following definitional equation:

$$\mathtt{Mn}\ \varphi_{a'}\ v_a\ y\ =\ \varphi_{a'}\ (y :: v_a)\ \mathtt{O} \wedge \forall n, n < y \to \exists k, \varphi_{a'}\ (n :: v_a)\ (\mathtt{S}\,k)$$

which mimics the definition of the minimization scheme in [1, p. 70].

Having defined the semantic (extensional) interpretation of μ-algorithms, we verify that $[\![S_a]\!]$ is a functional relation. We proceed by structural induction on S_a,

```
Theorem ra_sem_fun {a} (S_a : 𝓐_a) : functional [[S_a]]
```

directly with a fixpoint, by compositionally exploiting the fact that μ-recursive schemes preserve functional relations.

4.3 The interpreter for μ-algorithms

Following the approach hinted in the introduction, given a specification predicate $P : X \to$ `Prop` over a type X, we characterize a *specified partial value* by a term $t : (\exists x, P\,x) \to \{x \mid P\,x\}$ in which the specified value $\{x \mid P\,x\}$, that is, an x paired with a proof of $P\,x$, is guarded by its existence $(\exists x, P\,x)$:

- The unary predicate $P : X \to$ `Prop` gives the specification of the value. In our case, we instantiate e.g. $P := [\![S_a]\!]\ v_a :$ `nat` $\to$ `Prop`, hence, thanks to `ra_sem_fun`, P will hold for at most one x;
- the Coq term t computes a value x such that $P\,x$, provided it is given a certificate for its algorithm to terminate the computation, stated as the non-informative existence of an (output) value satisfying P.

In the file `compute_def.v`, we define the predicate capturing specified partial values as `compute` $\{X\}\ (P : X \to$ `Prop`$) := (\exists x, P\,x) \to \{x \mid P\,x\}$.[2] This encoding of partiality allows a direct generalization of the code of the semantic `ra_sem` (noted $[\![\cdot]\!]$) into an interpreter `ra_compute` (noted $[\![\cdot]\!]_o$) as described below, with relatively short proof terms for pre/post conditions.

As a side note, our approach contrasts with the "partiality monad" of [2] where a partial value is of type $\{Q :$ `Prop` $\mid Q \to X\}$, i.e. though guarded by a predicate Q, it refers to an output type X *only*, and is not further specified in that type. In our case, a `compute` value is always specified w.r.t. a predicate over a type (i.e. P) which in practice characterizes that value uniquely. It comes with a proof that the value satisfies its specification (i.e. $P\,x$). Hence that guard and the specification are linked together. Also, having an output specification allows us to compositionally derive a correct-by-construction interpreter.

[2] Notice that this definition is not intended to hint at the traditional notion of computability.

We can now present the Coq term for μ-algorithms that is going to be extracted into OCaml as a natural interpreter for μ-algorithms in that programming language. For $S_a : \mathcal{A}_a$, the term `ra_compute` S_a : $\forall v_a$: `vec nat` a, `compute` $\big([\![S_a]\!]\, v_a\big)$ will realize the extensional interpretation $[\![\cdot]\!]$, by directly computing the output from the input along the lines of the given μ-algorithm. In the file `interpreter.v`, we write the following `Fixpoint ra_compute` $\{a\}$ S_a also denoted $[\![S_a]\!]_{\mathrm{o}}$ (for a more compact notation) reusing the same scheme as that of the code of `ra_sem` $\{a\}$ $S_a = [\![S_a]\!]$. The suffix in the new notation $[\![\cdot]\!]_{\mathrm{o}}$ is intended to recall that we do not simply define a proposition but instead, an o(utput) value is now computed:

```
Fixpoint ra_compute {a} (S_a : A_a) {struct S_a} : ∀v_a : vec nat a, compute ([[S_a]] v_a) :=
  match S_a with
  | ra_zero       ⇒ Zr_compute
  | ra_succ       ⇒ Sc_compute
  | ra_proj i     ⇒ Id_compute i
  | ra_prec S_a S_a'' ⇒ Pr_compute [[S_a]]_o [[S_a'']]_o
  | ra_umin S_a'  ⇒ Mn_compute [[S_a']]_o
  | ra_comp S_b S_ab ⇒ Cn_compute [[S_b]]_o (λ v_a dv_a, vec_map_compute ([[·]]_o v_a) S_ab dv_a)
  end where [[S_a]]_o := (ra_compute S_a).
```

The sub-term $[\![\cdot]\!]_{\mathrm{o}}\, v_a$ has type $\forall S_a$, `compute` $\big([\![S_a]\!]\, v_a\big)$ which states that $[\![\cdot]\!]\, v_a$ is a partial function, which can be computed in Coq if fed with a certificate that its output exists (termination), and it is passed to `vec_map_compute` which generalize `vec_map` to the application of partial functions on every component of a vector, but still by *structural recursion* on the vector. Then the computation follows a natural interpretation of μ-algorithms as functional programs via extraction.

We further comment on the code of the `ra_compute` fixpoint, focusing on how Coq establishes termination. First, it proceeds by structural recursion on S_a. Hence the guard-checker verifies that `ra_compute` is only applied to sub-terms of S_a. And for this, it has to inspect the code of `vec_map_compute` which inevitably nests a call to `ra_compute` (noted $[\![\cdot]\!]_{\mathrm{o}}$) because S_{ab} : `vec` $\mathcal{A}_a\, b$ is a (nested) vector of sub μ-algorithms in $\mathcal{A}_a$. Because `vec_map_compute` proceeds by recursion on S_{ab}, the guard checker accepts this nesting.

In the case of `ra_umin` $S_{a'}$, we see that `Mn_compute` receives $[\![S_{a'}]\!]_{\mathrm{o}}$, the fixpoint itself applied to $S_{a'}$, as first parameter, which obviously passes the guard-checker. The code of `Mn_compute` can be found in the `compute.v` and is based on that of `umin`$_0$`_compute`; see the file `umin_compute.v` and explanations in Section 3.5.

```
Variables (a : nat) (S_a' : A_{1+a}) (cS_a' : ∀v_a', compute ([[S_a']] v_a')).
Definition Mn_compute v_a : compute (Mn [[S_a']] v_a) :=
          umin_0_compute (λ _, ra_sem_fun _ _) (λ n dn, cS_a' (n :: v_a) dn).
```

The case of `ra_prec` $S_a\, S_{a''}$ is similar to that of `ra_umin` $S_{a'}$. The case of `ra_comp` $S_b\, S_{ab}$ is however more complicated because of the nesting with `vec_map_compute` that is mandated to recursively iterate $[\![\cdot]\!]_{\mathrm{o}}\, v_a$ over the components of the vector of sub μ-algorithms S_{ab}.

```
Variables (a b : nat) (S_b : A_b) (cS_b : ∀v_b, compute ([[S_b]] v_b))
          (S_ab : vec A_a b) (cS_ab : ∀v_a, compute (λ v_b, ∀i, [[S_ab.[i]]] v_a v_b.[i])).
Definition Cn_compute : ∀v_a, compute (Cn [[S_b]] (vec_map [[·]] S_ab) v_a) :=
          λ v_a dv_a, let (v_b, v_a v_b) := cS_ab v_a _ in let (y, v_b y) := cS_b v_b _ in ⟨y, _⟩.
```

We leave out as holes _ the three (small) proof obligations that can be studied further in code file `compute.v`. The term `vec_map_compute` is used to fill the argument cS_{ab} of `Cn_compute` and its code is described in the file `map_compute.v`. It generalizes the code of `vec_map` to deal with specifications (i.e. pre/post conditions), but extracts the same.

4.4 Remarks on a carefully crafted library of indices and vectors

The types for indices and vectors are defined inductively with the following rules/constructors:

$$\texttt{idx} : \texttt{nat} \to \texttt{Type} := \mathbb{O} : \forall\{n\}, \texttt{idx}\,(\texttt{S}\,n) \mid \mathbb{S} : \forall\{n\}, \texttt{idx}\,n \to \texttt{idx}\,(\texttt{S}\,n)$$
$$\texttt{vec}\,X : \texttt{nat} \to \texttt{Type} := \langle\,\rangle : \texttt{vec}\,X\,\texttt{O} \mid \_ :: \_ : \forall\{n\}, X \to \texttt{vec}\,X\,n \to \texttt{vec}\,X\,(\texttt{S}\,n)$$

We can analyze the content of vectors by standard pattern matching, or, on nonempty vectors, using the standard $\texttt{vec\_head} : \texttt{vec}\,X\,(\texttt{S}\,n) \to X$ and $\texttt{vec\_tail} : \texttt{vec}\,X\,(\texttt{S}\,n) \to \texttt{vec}\,X\,n$ functions. But to access the components in a more versatile way, as sometimes required by the definition of μ-algorithms, we define the projection $\texttt{vec\_prj}\,\{X\,n\} : \texttt{vec}\,X\,n \to \texttt{idx}\,n \to X$. The Coq fixpoint defining `vec_prj` is carefully written by structural recursion on the vector;[3] and the use of $\texttt{idx\_inv}\,\{0\} : \texttt{idx}\,0 \to \texttt{False}$ allows to dispose of the impossible case in a guard-checker friendly way.

```
Fixpoint vec_prj {n} (u : vec X n) : idx n → X :=
  match u in vec _ m return idx m → X with
  | ⟨⟩     ⇒ λ i, match idx_inv i with end
  | x :: v ⇒ λ i, match i in idx m return vec _ (pred m) → X with
              | 𝕆   ⇒ λ _, x
              | 𝕊 j ⇒ λ v, vec_prj v j
              end v
  end
```

The type of $\texttt{idx\_inv}\,\{n : \texttt{nat}\}$ is a bit more general (by dependent pattern matching on n), to also allow inversions of indices in $\texttt{idx}\,(\texttt{S}\,n)$ but the idea is the same. Actually, besides the definition of $\texttt{idx}\,n$, the statement of the lemma `idx_inv` together with its short proof is the only tool defined in our library for indices (in file `index.v`). Notice that we avoid `idx_inv` by inlining it in the second match case $x :: v$ because using it would introduce an additional level of constructors/matches in the extracted code.

Then $v.[i]$ is just a convenient notation for $\texttt{vec\_prj}\,v\,i$. As a consequence of our definition, the identities $(x :: v).[\mathbb{O}] = x$ and $(x :: v).[\mathbb{S}\,i] = v.[i]$ hold by definitional equality. But more importantly, any component $v.[i]$ is recognized as a *sub-term* of v by the guard-checker when type-checking a fixpoint nesting a call to `vec_prj`. Additionally, `vec_prj` extracts to desirable OCaml code:

```
let rec vec_prj u i = match u with
  | ⟨⟩     → assert false
  | x :: v → match i with O → x | S j → vec_prj v j
```

The projection `vec_prj` allows to view the inductive type $\texttt{vec}\,X\,n$ as an extensional representation of the type $\texttt{idx}\,n \to X$: two vectors are equal iff their components are equal, i.e. $\big(\forall i,\, v.[i] = w.[i]\big) \to v = w$, which is not the case for "functional vectors" in $\texttt{idx}\,n \to X$.

Complementary to `vec_head` and `vec_tail`, we also provide versatile inversion lemmas for vectors in either $\texttt{vec}\,X\,\texttt{O}$ or $\texttt{vec}\,X\,(\texttt{S}\,n)$ of types (see `vec.v` for detailed explanations):

$$\texttt{Definition vec\_O\_inv}\ \{X\}\ \{P : \texttt{vec}\,X\,\texttt{O} \to \texttt{Type}\} : P\,\langle\,\rangle \to \forall u,\, P\,u.$$
$$\texttt{Definition vec\_S\_inv}\ \{X\,n\}\ \{P : \texttt{vec}\,X\,(\texttt{S}\,n) \to \texttt{Type}\} : \big(\forall x\,v, P\,(x :: v)\big) \to \forall u, P\,u.$$

[3] The Coq standard library version `8.16.1` of `Vector.nth` proceeds by recursion on the index, not on the vector, which would conflict with our guard conditions, but a version of `nth` computationally similar to `vec_prj` has been accepted into future revisions of the standard library as PR #16731.

For instance, the term `vec_S_inv` $(\lambda\, x\, v,\ f\, x\, v)\, u$ can then be seen as a correct (hence type-checkable) way to write something like `match` u `with` $x :: v \Rightarrow f\, x\, v$ `end` for a vector u : `vec` _ (`S` _), that moreover *extracts* into a pattern-matching on u, i.e. of the form

`match` u `with` $\langle\,\rangle \to$ `assert false` $\mid x :: v \to f\, x\, v$.

The alternate code f (`vec_head` u) (`vec_tail` u) would extract less gracefully in two successive pattern-matchings on u performed inside `vec_head` and `vec_tail`.

5 Extraction to OCaml

To shorten a bit the extracted code and make it easier to read, in the file `interpreter.v` we feed the extraction plugin of Coq with several kinds of directives:

- we generally forget about arities because they do not participate in the computation. They exist at proof-level to ensure that e.g. composition (resp. projection) occurs only between vectors (resp. and indices) of proper arities;
- we extract indices in `idx` n as natural numbers directly to avoid duplication of code between `idx` _ and `nat`;
- having forgotten their arity, we can extract vectors as native OCaml lists to present the reader with a familiar notation for tuples;
- we inline some Coq terms to avoid duplicating OCaml names for the same functions and avoid steps that need no factorization because they are only used once.

With those directives, in a first iteration of extraction, we get the OCaml interpreter for μ-algorithms as presented in the introduction, with two minor differences that we describe and discuss how they can be addressed below.

The first difference is that Coq does not generate partial `match` filters and thus, we get extra `assert false` statements instead of missing match cases. Computationally the only difference this makes is in the name of the generated exception. However, they should not be triggered unless the OCaml interpreter is called on a context which could not be typed within Coq, e.g. the input vector has a shorter length that the arity of the μ-algorithm.[4]

The second difference is the occurrence of `__` = `Obj.t` OCaml type and object that the extraction plugin uses to circumvent the type system of OCaml on Coq types which are too general for it.[5] This difference is more important to tackle in our opinion.

Let us start with an explanation of why these `__` appear in the extracted code in the first place. They come from e.g. the second (non implicit) argument of $\texttt{umin}_0\texttt{\_compute}$, which is a partial value of type $f : \forall n, \texttt{compute}\ (F\, n)$ or, by expanding the definition of `compute`, of type $f : \forall n, (\exists y, F\, n\, y) \to \{y \mid F\, n\, y\}$. The extraction plugin is not able to recognize that it can safely erase $(\exists y, F\, n\, y)$ because f is itself *an argument* of $\texttt{umin}_0\texttt{\_compute}$. No directive of our knowledge is able to inform the extraction plugin with non-informative data in the types of the arguments of extracted terms.

We present two ways of getting rid of `__`. Both consist in hiding the proposition $(\exists y, F\, n\, y)$ in the propositional part of a Σ-type. We think it is better to describe these tricks as a diff on the Coq code rather than directly exposing a more convoluted variant of the interpreter; see files `unit.diff` and `hide.diff`.

[4] We do not know if it is possible to instruct the extraction plugin to dismiss impossible match cases.

[5] Additional issues could be raised by the call-by-value evaluation strategy of OCaml; anyway, extracting to Haskell produces similar extra arguments of type `any`, showing that the discussion below still makes sense even with a lazy strategy, especially the second improvement.

The first approach consists in adding a new parameter of type `unit` and packing it with the proposition $(\exists y, F\,n\,y)$, hence we get the following definition of $\texttt{compute}_u$:

$$\texttt{Definition compute}_u\ \{X\}\ (P : X \to \texttt{Prop}) := \{\_ : \texttt{unit} \mid \exists x, P\,x\} \to \{x \mid P\,x\}$$

the type of the parameter f in $\texttt{umin}_0\texttt{\_compute}$ becoming $f : \forall n, \texttt{compute}_u\,(F\,n)$. We do not need to upgrade `compute` into $\texttt{compute}_u$ everywhere though, only when a parameter is a partial function, e.g. in the definitions of `prim_rec_compute`, or `vec_map_compute` or that of `umin_compute` and $\texttt{umin}_0\texttt{\_compute}$.

This solution has the advantage of symmetry (see below) and conceptual simplicity. The simplest way to visualize small amount of needed updates in the code is through the diff file `unit.diff`. The resulting extracted code is the same as in the first iteration except that `Obj.t` (resp. `__`) gets substituted with `unit` (resp. `()`). So there is no trick to circumvent the OCaml type system anymore but still, an extra dummy/`unit` argument remains.

The second approach gives us an extraction where the `__` parts are completely removed from the code. This is quite satisfying and not much more complicated than the `unit` trick but we lose symmetry in the treatment of the arguments of Coq terms. The trick consists in hiding $(\exists y, F\,n\,y)$ directly under the *last argument* it depends on, hence n in the case of $\texttt{umin}_0\texttt{\_compute}$. So we get the following type for its second argument:

$$f : \forall p : \{n \mid \exists y, F\,n\,y\},\ \{y \mid F\ (\pi_1\,p)\ y\}.$$

Again, we only need to make that change on the type of the arguments that represent partial functions, e.g. f, not on the terms implementing partial functions, e.g. $\texttt{umin}_0\texttt{\_compute}$. We recall that the simplest way to visualize the small amount of required modifications in the Coq code is via the diff file `hide.diff`.

6 Conclusion

Program extraction was advocated as an interesting approach to the study of the correctness (by construction) of functional programs for a long time, and the issue of partial functions, especially possibly non-terminating programs, was raised very early, both in untyped settings such as PX [6] and in strongly typed logical settings where only terminating (functional) programs can be expressed such as Nuprl [3]. Parametric ways to deal with partial values in Coq include [2, 4], allowing for the development of synthetic computability theory [5].

In addition to theoretical considerations, the issue of partiality is not that easy from a practical point of view, notably when partial functions are mixed with higher-order functional programs: when the latter are basically structurally recursive, it is desirable to keep their conceptual simplicity as much as possible.

We think that the example of μ-recursive functions contains a significant summary of the issues raised, so the work presented here may help to understand how they can be dealt with in CIC as implemented in the Coq proof assistant. We could have just tried to follow the Braga method [11], i.e., to provide an inductive definition of the domain of the *full* desired interpreter (`ra_compute`) either directly, or from an inductive presentation of its input-output graph. Clearly, the resulting development would have been much more convoluted. Instead, we have limited the use of the machinery of [11] – actually a tail-recursive variant of it – at the single place where it is relevant (unbounded minimization), using ordinary structural recursion on the input data at all other places. Beyond careful explanations on why and how guard condition for termination are satisfied, the resulting development is made conceptually

simple and concise. To this effect, a very simple, hence easy to overlook idea turned out to be surprisingly effective: use $\forall x,\ (\exists y, G\,x\,y) \to \{y \mid G\,x\,y\}$ as a general shape for specifying `ra_compute` and its auxiliary functions.

Note that, as a bonus, the results previously presented in [9] can then be obtained with much shorter and elegant proofs.

References

1. George S. Boolos, John P. Burgess, and Richard C. Jeffrey. *Computability and Logic*. Cambridge University Press, 4 edition, 2002. doi:10.1017/CBO9781139164931.
2. Mario Carneiro. Formalizing Computability Theory via Partial Recursive Functions. In *ITP 2019*, volume 141, pages 12:1–12:17. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. doi:10.4230/LIPIcs.ITP.2019.12.
3. Robert L. Constable et al. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.
4. Yannick Forster. Church's Thesis and Related Axioms in Coq's Type Theory. In *CSL 2021*, volume 183, pages 21:1–21:19. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CSL.2021.21.
5. Yannick Forster. Parametric church's thesis: Synthetic computability without choice. In Sergei Artemov and Anil Nerode, editors, *Logical Foundations of Computer Science*, pages 70–89, Cham, 2022. Springer International Publishing.
6. Susumu Hayashi. *Extracting Lisp Programs from Constructive Proofs: A Formal Theory of Constructive Mathematic Based on Lisp*, volume 19, pages 169–191. Publications of the Research Institute for Mathematical Sciences, 1983.
7. Stephen C. Kleene. λ-definability and recursiveness. *Duke Mathematical Journal*, 2(2):340–353, 1936. doi:10.1215/S0012-7094-36-00227-2.
8. Stephen C. Kleene. Recursive predicates and quantifiers. *Trans. Amer. Math. Soc.*, 53:43–73, 1943. doi:10.1090/S0002-9947-1943-0007371-8.
9. Dominique Larchey-Wendling. Typing Total Recursive Functions in Coq. In *ITP 2017*, pages 371–388. Springer, 2017. doi:10.1007/978-3-319-66107-0_24.
10. Dominique Larchey-Wendling and Yannick Forster. Hilbert's Tenth Problem in Coq (Extended Version). *Logical Methods in Computer Science*, Volume 18, Issue 1:35:1–35:41, March 2022. doi:10.46298/lmcs-18(1:35)2022.
11. Dominique Larchey-Wendling and Jean-François Monin. *The Braga Method: Extracting Certified Algorithms from Complex Recursive Schemes in Coq*, chapter 8, pages 305–386. World Scientific, 2021. doi:10.1142/9789811236488_0008.
12. Joseph R. Shoenfield. *Mathematical Logic*. Addison-Wesley series in logic. Addison-Wesley, 1967.
13. Vincent Zammit. A Proof of the S-m-n theorem in Coq. Technical report, The Computing Laboratory, The University of Kent, Canterbury, Kent, UK, March 1997. URL: http://kar.kent.ac.uk/21524/.

Group Cohomology in the Lean Community Library

Amelia Livingston ✉
King's College London, UK

Abstract

Group cohomology is a tool which has become indispensable in a wide range of modern mathematics, like algebraic geometry and algebraic number theory, as well as group theory itself. For example, it allows us to reformulate classical class field theory in cohomological terms; this formulation is essential to landmarks of modern number theory, like Wiles's proof of Fermat's Last Theorem. We explore the challenges of formalising group cohomology in the Lean theorem prover in a generality suitable for inclusion in the community library `mathlib`.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Type theory

Keywords and phrases formal math, Lean, mathlib, group cohomology, homological algebra

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.22

Supplementary Material *Software*: `https://github.com/101damnations/ITP2023SupplementaryMaterial`, archived at `swh:1:dir:d647ad13fc6cc27a6fe9f3b49691d0e1716a532c`

Funding This work was supported by the Engineering and Physical Sciences Research Council [EP/S021590/1]. The EPSRC Centre for Doctoral Training in Geometry and Number Theory (The London School of Geometry and Number Theory), University College London

Acknowledgements I am very grateful to Kevin Buzzard for his ongoing mathematical and Lean-related support and guidance. I am also indebted to Joël Riou for his explanation of the simplicial interpretation of group cohomology and his thorough reviewing of and advice regarding my work, and for his formalisation of some of the results I used. I also depended heavily on Scott Morrison's development of Lean's representation theory library and the category theory library more generally. Finally, thanks to anyone who answered my questions on the Xena Project Discord server and the Lean Zulip.

1 Introduction

1.1 Motivating group cohomology

There are many cohomology theories in mathematics. They associate simpler, "linear" invariants (vector spaces, or more generally modules, and linear maps between them) to more complicated objects, and analysing these invariants can answer questions about the complicated objects.

We want a cohomology theory for groups. They are ubiquitous in maths. Groups themselves often appear as invariants of more complex objects: we can study a topological space by studying its fundamental group, or field extensions by their Galois groups, or rings by their K-groups in algebraic K-theory. But they are still more complicated than "linear" invariants, and abstract group theory itself is not easy. The simpler invariants we obtain in group cohomology come from asking how a group acts on other objects, rather than analysing it internally.

This is the spirit of group cohomology – but there are multiple ways to actually define it. This is often the case in maths, and different definitions lend themselves to different exploits. There might be a particularly abstract formulation, expressing a concept as a special case of some more general category-theoretic notion. This perspective tends to give us access to


© Amelia Livingston;
licensed under Creative Commons License CC-BY 4.0

powerful techniques for developing theory. But because the tools come from a more general setting, they are not specialised for the computation of examples, or for proving theorems that depend on the specificities of our situation. For these, we need a different, more down to earth approach, which will necessarily vary from setting to setting.

Because cohomology theories typically have a similar abstract foundation, there are many cohomological examples of this abstraction versus practicality dichotomy. For example, in algebraic topology, we find a variety of ways to compute singular cohomology, e.g. via simplicial, cellular or de Rham cohomology. More generally, we can calculate sheaf cohomology using Čech cohomology.

The situation is no different for group cohomology. Given a group G acting on an abelian group M, group cohomology is a family of groups $\mathrm{H}^n(G, M)$ for $n \in \mathbb{N}$. Mathematicians have analysed the groups of low degree, i.e. for $n \leq 2$, via explicit calculation since before "group cohomology" was a term [14, p. 10]. For example, H^1 appears in Hilbert's Theorem 90, originally proved by Kummer in 1855 [8, p. 213]; among many other applications, this result parametrises the solutions to certain Diophantine equations [7, p. 3]. Meanwhile, H^2 classifies group extensions, as explained by Baer and others in the early 20th century [14, p. 10]. Given an abelian group A and another group G, this means classifying the groups E having A as a normal subgroup and such that $E/A \cong G$, revealing the ways in which bigger groups can be built from smaller ones.

On the other hand, the abstract story, which was developed in the mid-20th century [14, p. 11–12], gave us more tools to analyse $\mathrm{H}^n(G, M)$ for arbitrary n, and explore relationships between these groups as n, G and M vary, via homological techniques. It also connects group cohomology to topological cohomology. Thus we wish to formalise both the abstract and concrete approaches, and prove their equivalence. This equivalence is the focus of the paper.

1.2 Lean and `mathlib`

Lean is an interactive theorem prover that uses dependent type theory; every "object" in Lean is a term of a unique type. A Lean file mainly consists of definitions, lemmas and theorems, which the user must prove with the assistance of tactics that provide some degree of automation. We will meet some tactics and structural features of Lean during the paper, but try to explain these with as little code as possible. When this is unreasonable, we provide links to Lean files, indicated with the symbol 🔗, which illustrate details in more depth. An archive containing saved versions of the non-permanent links in the paper can be found at 🔗.

Like maths, formalisation is a collaborative process. In order to make progress, we must make use of work that has been done before; some of this work is collected in a library called `mathlib` 🔗. The library comprises folders for each mathematical subfield, and each file consists of a collection of definitions and facts, which we call an API, relating to a particular mathematical concept. But there are typically many ways to formalise the same object, and it is not always obvious which ways are "right": which implementation can feasibly be used in the formalisation of further material. Lean cannot work this out for us; many factors influence the extensibility of Lean code, and typically we must make an educated guess at the correct formalisation and potentially refactor when a future user runs into difficulties. This makes the task of writing `mathlib`-suitable code significantly harder than code which just compiles, and the `mathlib` library has a rigorous community review process to try and avoid having to refactor new work down the line. There is no algorithm for making sure the growth of `mathlib` is sustainable; it seems to require human insight.

Often this is because the library needs to meet the needs of humans. On the one hand, it should be organised coherently, so the user has some idea of where to find what they need. On the other hand, APIs should be fleshed out enough that a user does not need to know

the specific implementation the author chose – to an extent, there should be support for any other user imagining a different implementation. It seems to the author that these aims can conflict.

But not all challenges in `mathlib` design are due to the limitations of humans. One reason for this is the complexity of the algebraic hierarchy (a portion of which can be seen in [10, p. 4]). Everything in `mathlib` should be stated in the maximum possible generality within this hierarchy, so that it can be used in any setting in which it applies. In simple examples this is easy to ensure – it is not hard to check if a lemma about groups actually applies to monoids too.

In more complicated settings, however, the pursuit of generality is less straightforward. Firstly, it often means needing to create more new API than previously anticipated: it took a surprisingly long time for real manifolds to enter `mathlib`, as contributors needed to develop a wide variety of more general material, like the Bochner integral. But this is not the only difficulty. In principle, abstraction should simplify code – but of course we will need to apply the general material to more specific situations, and this is where complexity arises. For instance, in our simple example, a lemma about groups becomes a lemma about monoids which we are applying to a group, so we have to use something extra: the fact that a group is a monoid. Obviously, this is fine. But the many iterations of this principle in complicated settings has been a factor in most of the challenges in this project, as it can slow Lean down and make errors more difficult to troubleshoot.

The project has taken a long time to develop considering its mathematical simplicity. The author has now contributed two definitions of group cohomology to `mathlib`, as well as proof of their equivalence. This required 10 sizeable pull requests, and often the material had been rewritten to increasing degrees of abstraction. The paper will illustrate in detail the development of this code.

Section 2 explains the essentials of the maths we wish to formalise, and explores some fundamental design decisions and Lean principles. Section 3 describes our formal version of a key object called the standard resolution, and Section 4 discusses how we use this object to define group cohomology in Lean. In Section 5 we conclude and detail the future of the project: the author is currently using the definitions explained to create further API and prove various group cohomological results, although the code is not yet prepared enough for presentation.

2 Preliminaries

2.1 Mathematical background

▶ **Definition 1.** *Given an abelian category $\mathcal{C}$ (for example, the category of abelian groups, or the category of modules over a ring) a* ***cochain complex*** *X in $\mathcal{C}$ indexed by $\mathbb{N}$ is a sequence*

$$0 \to X_0 \xrightarrow{d_0} X_1 \xrightarrow{d_1} \dots \xrightarrow{d_{n-1}} X_n \xrightarrow{d_n} \dots$$

of objects $X_n \in \mathcal{C}$ and morphisms $d_n : X_n \to X_{n+1}$, $n \in \mathbb{N}$, satisfying $d_{n+1} \circ d_n = 0$ for all n.

We call the morphisms differentials. The condition $d_{n+1} \circ d_n = 0$ means the image of d_n is contained in the kernel of d_{n+1}, allowing us to define

▶ **Definition 2.** *The* ***nth cohomology*** *of X, $\mathrm{H}^n(X)$, is $\mathrm{Ker}(d_n)/\mathrm{Im}(d_{n-1})$.*

A **chain complex** is the same, but with the morphisms in the other direction: d_n is a morphism $X_{n+1} \to X_n$, and the analogous invariant is called **homology**, denoted $\mathrm{H}_n(X)$.

With this, we can explain the abstract and concrete perspectives on group cohomology, starting with the latter. The material can be found here [2, Chapter 4].

Given a group G, we call an additive commutative group M a G-module if it has a map $\cdot : G \times M \to M$ satisfying $gh \cdot m = g \cdot h \cdot m$ for $g, h \in G, m \in M$ and which also distributes over addition, i.e. $g \cdot (m + n) = g \cdot m + g \cdot n$ for $g \in G, m, n \in M$.

▶ **Definition 3.** *The **nth group cohomology** of G and M, denoted $\mathrm{H}^n(G, M)$, is the nth cohomology of the cochain complex*

$$0 \to M \to \mathrm{Fun}(G, M) \to \mathrm{Fun}(G^2, M) \to \mathrm{Fun}(G^3, M) \to \dots \tag{1}$$

where $\mathrm{Fun}(G^n, M)$ is the set of functions from G^n to M, and the differential d^n maps $f : G^n \to M$ to the function sending $(g_0, \dots, g_n)$ to

$$g_0 \cdot f(g_1, \dots, g_n) + \sum_{i=0}^{n-1} (-1)^{i+1} f(g_0, \dots, g_i g_{i+1}, \dots, g_n) + (-1)^{n+1} f(g_0, \dots, g_{n-1})$$

(and $d^0(m) = (g \mapsto g \cdot m - m)$).

The differential is slightly messy, but this is the formulation that in some sense "shows up in nature"; for example, it makes clearer the correspondence between $\mathrm{H}^2(G, M)$ and certain equivalence classes of group extensions. We call the $\mathrm{Fun}(G^n, M)$ **inhomogeneous cochains**.

The abstract perspective, meanwhile, is often introduced in a category of modules over a ring, rather than in the category of G-modules (where morphisms are group homomorphisms satisfying $f(g \cdot x) = g \cdot f(x)$ for all $g \in G$). Like `mathlib`, the average undergraduate student contains a far larger API for modules than for other concrete abelian categories, so when explaining group cohomology it is natural to exploit the equivalence between the category of G-modules and the category of modules over the following ring:

▶ **Definition 4.** *The **group ring** $\mathbb{Z}[G]$ is the free abelian group on G (functions $G \to \mathbb{Z}$ which are nonzero at finitely many elements of G), with multiplication induced by that of G.*

We denote its elements as sums $\sum n_i g_i$ for $n_i \in \mathbb{Z}, g_i \in G$. We can inject G into $\mathbb{Z}[G]$ by sending g to the function which is 1 at g and 0 everywhere else; we will often abuse notation and just denote this function by g.

Given this category equivalence G-Mod $\cong \mathbb{Z}[G]$-Mod, we can express group cohomology as objects called Ext groups. Ext is an example of a derived functor; these are abstract objects equipped with high-powered theory and are useful in many algebraic fields of maths. We summarise their definition; the material can be found in [13, Chapters 1, 2].

▶ **Definition 5.** *A **morphism of chain complexes** $f : X \to Y$ is a family of morphisms $f_n : X_n \to Y_n$ making the resulting diagram commute – that is, $d^Y_n \circ f_{n+1} = f_n \circ d^X_n$ for each n.*

A chain complex morphism induces maps on each homology group $\mathrm{H}(f_n) : \mathrm{H}_n(X) \to \mathrm{H}_n(Y)$; we call f a **quasi-isomorphism** if each $\mathrm{H}(f_n)$ is an isomorphism. Analogous definitions can be made for morphisms of cochain complexes and cohomology.

To analyse an object X in an appropriate category $\mathcal{C}$, we can sometimes associate to it a chain complex of simpler objects, called a "resolution" of X, and study that instead. More precisely:

▶ **Definition 6.** *A **projective resolution of** X is a chain complex P and a quasi-isomorphism of chain complexes $f : P \to X[0]$ such that each P_n is **projective** (a certain "nice" property).*

By $X[0]$ we mean the complex whose 0th object is X, with every other object 0. All the f_n for $n > 0$ are necessarily 0. The requirement that f is a quasi-isomorphism means $\mathrm{H}^n(P)$ must be trivial for all $n > 0$; we say P is **exact except at the right**.

When we analyse what certain "nice" functors F do to X, we can learn more by applying them to all of P, and then taking the (co)homology of the resulting complex, which in general will no longer be trivial. However, due to the conditions in the definition of a projective resolution, $\mathrm{H}^0(F(P))$ will always be isomorphic to $F(X)$, so we do not lose information. The $\mathrm{H}^n(F(P))$ are independent of the projective resolution chosen, and can be extended to functors; we call them the derived functors of F.

Given an object Y, one such F is $\mathrm{Hom}(-, Y)$. This functor is contravariant, meaning it flips the directions of maps: a map $\phi : X_1 \to X_2$ is sent to the map $\mathrm{Hom}(X_2, Y) \to \mathrm{Hom}(X_1, Y)$ given by precomposition with ϕ. Because of this contravariance, the functor sends a chain complex to a cochain complex, giving us

$$0 \to \mathrm{Hom}(P_0, Y) \xrightarrow{-\circ d_0} \mathrm{Hom}(P_1, Y) \xrightarrow{-\circ d_1} \mathrm{Hom}(P_2, Y) \to \dots$$

▶ **Definition 7.** *For P a projective resolution of X, the nth cohomology of the above complex is called $\mathrm{Ext}^n(X, Y)$. It is independent of the resolution chosen.*

Now, returning to group cohomology, we can appeal to the undergraduate's module API to observe that

$$\begin{aligned}\mathrm{Fun}(G^n, M) &\cong \mathrm{Hom}_{\mathbb{Z}}(\mathbb{Z}[G^n], M) \\ &\cong \mathrm{Hom}_{\mathbb{Z}[G]}(\mathbb{Z}[G], \mathrm{Hom}_{\mathbb{Z}}(\mathbb{Z}[G^n], M)) \cong \mathrm{Hom}_{\mathbb{Z}[G]}(\mathbb{Z}[G] \otimes_{\mathbb{Z}} \mathbb{Z}[G^n], M) \end{aligned} \tag{2}$$

Here Hom_R denotes morphisms in the category of R-modules, and the $\mathbb{Z}[G]$-module structure on $\mathbb{Z}[G] \otimes_{\mathbb{Z}} \mathbb{Z}[G^n]$ is given by $x \cdot (y \otimes z) = xy \otimes z$.

This isomorphism suggests that our concrete group cohomology groups are actually Ext groups of some sort. Indeed, the modules $\mathbb{Z}[G] \otimes_{\mathbb{Z}} \mathbb{Z}[G^n]$ are not just projective, but free, a stronger property. A **free module** is a module with a basis; for example, a vector space is a module over a field, and since all vector spaces have a basis, every module over a field is free. Since the $\mathbb{Z}[G^n]$ are free $\mathbb{Z}$-modules, the $\mathbb{Z}[G] \otimes_{\mathbb{Z}} \mathbb{Z}[G^n]$ are free $\mathbb{Z}[G]$-modules, and hence projective. We will also apply an isomorphism $\mathbb{Z}[G] \otimes_{\mathbb{Z}} \mathbb{Z}[G^n] \cong \mathbb{Z}[G^{n+1}]$; for our concrete group cohomology to agree with certain Ext groups, we then seek a projective resolution whose nth object is $\mathbb{Z}[G^{n+1}]$. Indeed:

▶ **Definition 8.** *The **standard resolution** is the chain complex*

$$\dots \xrightarrow{d_2} \mathbb{Z}[G^3] \xrightarrow{d_1} \mathbb{Z}[G^2] \xrightarrow{d_0} \mathbb{Z}[G] \to 0 \tag{3}$$

with differential sending $g = (g_0, \dots, g_n) \in G^{n+1}$ to

$$\sum_{j=0}^{n+1} (-1)^j (g_0, \dots, g_{j-1}, g_{j+1}, \dots, g_n).$$

The cokernel of d_0, i.e. $\mathbb{Z}[G]/\mathrm{Im}(d_0)$, is $\mathbb{Z}$. Indeed, this is a projective resolution of $\mathbb{Z}$ considered as a trivial $\mathbb{Z}[G]$-module – that is, $g \cdot m = m$ for each $g \in \mathbb{Z}[G], m \in \mathbb{Z}$. We will use this resolution to show that the nth group cohomology of M is in fact $\mathrm{Ext}^n_{\mathbb{Z}[G]}(\mathbb{Z}, M)$, and thus connect the concrete and abstract interpretations of group cohomology.

2.2 Initial formalisation considerations

There are many respects in which our formalisation is not a direct translation of the maths just outlined. To start explaining the code, we must address the more fundamental of these, which appear at every point in the project.

2.2.1 Complexes in Lean

We first explain the `mathlib` definition of complexes, which is a little counterintuitive, and which illustrates one of the Lean community's adaptations to the quirks of dependent type theory.

We first note that complexes in Lean have always permitted a more general indexing type than $\mathbb{N}$, and although the issues we will detail can also arise for $\mathbb{N}$, the most natural examples are for $\mathbb{Z}$.

If we followed our nose, our definition of a cochain complex in Lean would involve a function `X:` $\mathbb{Z} \to \mathcal{C}$ and a function `d` sending $n \in \mathbb{Z}$ to a morphism `X(n)` $\to$ `X(n + 1)`. This is essentially how complexes were originally formalised, and is fine in set theory, but not when we translate to dependent type theory. For example, when $\mathcal{C}$ is the category of abelian groups, we might want to know whether a term `x : X(n)` is in the image of the differential. But the type of this differential, according to our setup, is `d: X(n - 1)` $\to$ `X(n - 1 + 1)`. Of course, $n - 1 + 1$ is equal to n, but the problem is that type theory has multiple notions of equality.

The simplest one is syntactic equality: two objects are syntactically equal if they are "the same characters in the same order". $A = A$ is a syntactic equality. Next we have definitional equality – when, after unfolding definitions, two objects reduce to being syntactically equal:

```
def X : ℕ := 5
def Y : ℕ := 5
example : X = Y := rfl
```

where `rfl` proves the statement by unfolding the definitions of X and Y and applying reflexivity of equality. We note that not all tactics unfold terms like `rfl`. Given an equality or an iff statement whose lefthand side is syntactically equal to something in the goal, the `rewrite` and `simp` tactics can replace that expression in the goal with the righthand side of the equality/iff statement. The "simplifier" tactic `simp` is an example of Lean's automation: it searches through all lemmas tagged `@[simp]`, looking for statements whose lefthand side is syntactically equal to something in the goal, and then rewrites those lemmas. If it were to unfold terms as well as search its library, it would perhaps be rendered uselessly slow.

Finally, we have propositional equality: when two objects can be proved to be equal. For example:

```
example (a : ℕ) : a + 0 = a := rfl -- succeeds
example (a : ℕ) : 0 + a = a := rfl -- fails
```

since $\mathbb{N}$ is an inductive type, and addition is defined by induction on the second variable, not the first; we can prove the second statement by inducting on a, at which point we can appeal to definitional equalities. Hence the first statement is a definitional equality, and the second only a propositional equality.

Thus our issue is that $n - 1 + 1$ is not definitionally equal to n; its proof is the lemma `int.sub_add_cancel`. And in dependent type theory, everything is a term of a unique type; $d(y)$ cannot have type `X(n - 1 + 1)` and type `X(n)` simultaneously. If $n - 1 + 1$ had been

definitionally equal to n, Lean could unify `X(n - 1 + 1)` and `X(n)` by unfolding definitions, but since these expressions are only propositionally equal, the statement $d(y) = x$ will not typecheck.

We can, of course, compose with an isomorphism `X(n - 1 + 1)` $\cong$ `X(n)`; this is the approach taken in Domínguez and Rubio's formalisation of chain complexes in Coq [4, p. 6]. The UniMath library also uses the intuitive definition of chain complexes 🔗, since the univalence axiom means that such an isomorphism is equivalent to an equality. We note that in simple type theory, meanwhile, one must take a different approach: Isabelle/HOL defines exact sequences inductively as a certain kind of set of pairs of objects and functions 🔗.

However, carrying around these extra isomorphisms is unwieldy (in Lean, at least), and to deduce things about the resulting maps the user must prove heterogeneous equalities: equalities between terms of different types, denoted == .

Discussion here 🔗 concerns similar issues raised by commutative differential graded algebras. Over a commutative ring R, these are families of R-modules A_n indexed by $\mathbb{N}$ with, among other things, a family of R-bilinear "multiplication" maps $A_i \times A_j \to A_{i+j}$ for $i, j \in \mathbb{N}$ satisfying certain axioms. But the natural statement of associativity, for example, does not typecheck.

Ultimately, for complexes, the Lean community settled on a different implementation: define cochain complexes (and chain complexes) to have a differential between *every* pair of indices i, j, and require a proof that these are equal to 0 unless $i + 1 = j$, as well as a proof that any two differentials compose to give zero. This was first suggested by Johan Commelin here 🔗 on March 9th, 2021. Whilst this definition seems strange, it means we only have to identify non-definitionally equal types or check $i + 1 = j$ during proofs, and not when defining data. This is much easier to work with, as demonstrated here 🔗: we can use the `cases` tactic on hypotheses like $i + 1 = j$ and replace the lefthand side with the right in the types involved in the goal. The definition of complexes was refactored to use this approach in 2021, by Scott Morrison, in this pull request 🔗.

2.2.2 The right generality

Secondly, when we apply the `mathlib` tenet of maximum generality, we notice something: our exposition of group cohomology all works if we replace $\mathbb{Z}$ by an arbitrary commutative ring k, and ask that M is a $k[G]$-module, where $k[G]$ is defined analogously to $\mathbb{Z}[G]$; the rest of this paper will always use general k. We made a choice here – the alternative would be to continue developing the theory over $\mathbb{Z}$, and then just tensor with k when we want a more general result [12, Tag 0DVD]. Likewise, we could define additive commutative groups as the special case of k-modules when $k = \mathbb{Z}$. The "hierarchy of generality" seems to contain cycles. In this latter case it is easier to define additive commutative groups on their own, as the definition of module naturally extends the definition of additive commutative group. However, in our case the other approach is preferable, as it is simpler to replace $\mathbb{Z}$ with k than to have to involve tensor products.

2.2.3 Exploiting typeclass inference

Thus we are now concerned with the group k-algebra $k[G]$ and $k[G]$-modules; this raises further choices. In maths we denote any "scalar-like" action by $\cdot$. We can emulate this in Lean using typeclasses – a strategy also used in other theorem provers, like Agda, Coq and Isabelle [1, p. 1]. Maths is built from complex hierarchies of structures; typeclasses make it easier to formalise these hierarchies in an efficient and usable way. They enable us to reuse API for

simple structures when reasoning about more complicated superstructures; for example, we can use the notation + when dealing with any structure inheriting a `has_add` instance. Often, a structure can inherit an instance via multiple different paths, called diamond inheritance. This is fine when the inherited instances are definitionally equal. For instance, the semiring instance on a commutative ring coming from the inherited ring instance is definitionally equal to the one coming from the inherited commutative semiring instance, so it does not matter which path typeclass inference uses. But in practice, different inheritance paths will not always lead to definitionally equal instances, and this is problematic. Moreover, whilst concise notation is an advantage of typeclasses, it can make it harder to see when Lean's behaviour is not what we want. In practice we often want to consider multiple different $k[G]$-actions on the same object. Here 🔗 is an example of two different instances clashing; the action of $k[k^\times]$ on itself naturally extending the action of $k^\times$ on coefficients is not the same as the action of $k[k^\times]$ on itself by multiplication. But even two equal instances can conflict. Lean cannot unify instances that are only propositionally equal: the point of typeclass inference is to reduce the need to supply arguments explicitly, so we cannot provide a proof to the inference system of such an equality.

If we have a non-definitionally equal diamond and still wish to exploit typeclass inference, a possible solution is type aliases: a nickname for a type.

```
variables (R : Type*) [ring R]
def copy := R -- 'copy' is a nickname for 'R'
```

If we declare an instance on the type alias, it will not pollute the underlying type. For example, we can try defining 0 in `copy R` to be the 1 of the underlying ring:

```
instance : has_zero (copy R) := ⟨(1 : R)⟩
example : (0 : copy R) = 1 := rfl -- succeeds
example : (0 : R) = 1 := rfl -- fails
```

Conversely, Lean will not apply instances on the underlying type to the type alias, unless asked to:

```
instance : ring (copy R) := infer_instance -- fails
instance : ring (copy R) := by unfold copy; apply_instance -- succeeds
```

We can also use type aliases to organise API. For any type X and any type k with a 0, `finsupp X k` is the type of finitely supported functions $X \to k$. When G is a monoid and k is a commutative ring, `finsupp G k` is the k-algebra $k[G]$, with multiplication induced by that of G – but instead of creating this instance, `mathlib` defines a type alias `monoid_algebra k G` for `finsupp G k`, on which the k-algebra instance is defined. Hence results and instances relying on multiplication in G can be organised into the `monoid_algebra k G` API.

Thus, if we have a $k[G]$-action on a type M which could create diamonds if we declared it as an instance, we can instead make a type alias for M, and limit the scope of the instance.

In `mathlib`'s category theory library, we do something essentially equivalent to using type aliases constantly, but working in Lean's category `Module (monoid_algebra k G)` (i.e. $k[G]$-Mod) for our purposes is still tricky. Outside this library, an R-module is 3 different variables: a term M of type `Type*`, an additive commutative group instance `[add_comm_group M]`, and an R-module instance `[module R M]`. But when we work category-theoretically, an R-module is one variable: a term M of type `Module R`, which is a structure with 3 fields:

```
structure Module :=
(carrier : Type v)
[is_add_comm_group : add_comm_group carrier]
[is_module : module R carrier]
```

meaning the R-module structure on a term `M : Module R` is built into its type and is unambiguous. This achieves the same thing as declaring an alias for M, and then only defining the specific R-module structure we want on the alias.

But for us, even this will not suffice. If we package a G-module M with a compatible k-module structure as an object in $k[G]$-Mod, we still want to be able to talk about the underlying k-module structure, and the natural k-module structure on terms of type `Module (monoid_algebra k G)` is not definitionally equal to the k-module structure we started with, as proved here 🔗.

Instead, we use a further alternative. We bundle our actions of k and G as **k-linear representations of G**: a k-module M equipped with a monoid homomorphism $\texttt{M.}\rho$ of type $G \to \mathrm{End}_k(M)$, where $\mathrm{End}_k(M)$ is the ring of k-linear maps from M to itself. When developing representation theory, `mathlib` contributors were unsure whether to define representations this way, or as objects with a separate k-action and G-action, or as $k[G]$-modules, and Antoine Labelle and Eric Wieser vouch for the first definition here 🔗 on April 19th, 2022. It is similar to the definition chosen in Coq's Mathematical Components library 🔗. This way we only deal with one k-module instance, and the G-action on M is unambiguous. However, since the action has become an explicit homomorphism $\texttt{M.}\rho$ we cannot use typeclass inference or the notation $\cdot$. This does not even increase the number of arguments we need to give Lean, though; functions that would otherwise require the arguments k, G, M now only require M, since $\texttt{M.}\rho$ contains the information of k, G and M in its type. We call the category `Rep k G`:

```
structure Rep (k G : Type u) [comm_ring k] [group G] :=
(V : Module k)
(ρ : G →* End (Module k))
```

which we will subsequently denote $G\text{-Rep}_k$. Since this category is equivalent to $k[G]$-Mod, it has "enough" projective objects for us to talk about the derived functor Ext. We state everything in terms of representations.

3 Formalising the standard resolution

Now we are ready to discuss the content of the project. We started by constructing the standard projective resolution of the trivial k-module k, which we will denote P from now on. For each n, its nth object is the k-module $k[G^{n+1}]$ equipped with the representation induced by the diagonal action of G on G^{n+1}. The differentials, meanwhile, are easy to define, but we have to prove that $d_n \circ d_{n+1} = 0$ for all n. There is a sense in which this was already in `mathlib`; if we can build our resolution abstractly enough to use the `mathlib` proof, we will avoid some code duplication. We summarise what this entails.

3.1 Simplicial objects

Our resolution will come from something called a simplicial object, which we now define.

▶ **Definition 9.** *The* ***simplex category*** *is the category whose objects are the totally ordered sets* $[n] := \{0, 1, \ldots, n\}$ *for* $n \in \mathbb{N}$*, and whose morphisms are the order-preserving functions.*

In `mathlib` we just represent the objects as individual natural numbers; `simplex_category` is a type alias for $\mathbb{N}$. The category is generated by the maps $\delta_n(i), \sigma_n(i)$, where $\delta_n(i)$ is the unique order preserving injection $[n] \to [n+1]$ which misses i, and $\sigma_n(i)$ is the unique order preserving surjection $[n+1] \to [n]$ which hits i twice.

▶ **Definition 10.** *A* ***simplicial object*** *in* $\mathcal{C}$ *is a contravariant functor from the simplex category* Δ *to* $\mathcal{C}$*.*

If we apply a simplicial object X to $\delta_n(i)$, we get a map $X([n+1]) \to X([n])$, which we call the face maps of X. When $\mathcal{C}$ is abelian we can then define the **alternating face map complex** associated to X [11, Def 2.6]: a chain complex whose nth object is $X([n])$ and whose differential is given by the alternating sum of the face maps

$$\sum_{i=0}^{n+1} (-1)^i \cdot X(\delta_n(i))$$

which looks like the differential in our resolution, and also like the boundary maps of a topological simplicial complex. We then have the proof that this squares to zero 🔗 formalised by Joël Riou, which uses the fact that $\delta_{n+1}(i) \circ \delta_n(j+1) = \delta_{n+1}(j) \circ \delta_n(i)$. But why are we using the term "face"?

This is because any simplicial set X (i.e. simplicial object in Set; we can also view any of the simplicial objects we will be concerned with as simplicial sets) has a "geometric realisation" $|X|$: there is a functor from simplicial sets to the category of compactly-generated Hausdorff topological spaces. Essentially, we replace the elements of each $X([n])$ with standard topological n-simplices Δ^n, and how they glue together depends on how X acts on morphisms [5, Section I.2]. This is where the topological interpretation of group cohomology comes from. We have

$$H^n(BG, \mathbb{Z}) \cong H^n(G, \mathbb{Z}),$$

[13, Thm 6.10.5], where the lefthand side is the topological cohomology of BG, which is the classifying space of G: the fundamental group of BG is G and its higher homotopy groups are trivial. The classifying space BG is the quotient of a contractible space EG by an action of G, and EG is determined by the structure of G; it is the universal cover of BG.

3.2 Constructing the resolution using EG

Using the comparisons described above between certain topological spaces, simplicial objects and chain complexes, we can ultimately derive our projective resolution (3) from EG, as suggested by Joël Riou here 🔗, June 3rd 2022. The author formalised his suggestion; this approach to the standard resolution ultimately involved more lines of code, but the resulting formalisation was more suited to `mathlib` in its abstraction, motivated the creation of more API for objects like the Čech nerve, and taught the author material. We sketch the maths involved. Most of the code is here 🔗; anything else is here 🔗 or here 🔗. We also provide links to any key result formalised by someone else (i.e. Joël Riou).

As an overview of the strategy, we will define a simplicial object EG in the category G-Set (types with an action of G which respects multiplication in G). As a simplicial set, its geometric realization is the universal cover of BG. We can later "linearise": compose EG

with the free k-module functor from G-Set to G-Rep$_k$. Then, since G-Rep$_k$ is an abelian category, we can take the alternating face map complex associated to the resulting simplicial k-linear G-representation, which will be the standard resolution we seek. As hoped, defining the resolution using this EG gives us a proof that composition of the differentials equals zero for free. Moreover, we will show that our resolution is homotopy equivalent to $k[0]$ (the chain complex with k at 0 and 0 everywhere else) – this will define a quasi-isomorphism – and again, the simplicial approach gives us a more general proof of this than if we were to prove it directly. The homotopy equivalence's topological analogue is the contractibility of $|EG|$.

With this overview in mind, we explain the process in more detail. Given an appropriate morphism f in a category, we can define a certain simplicial object $C(f)$ called a Čech nerve. We first show that for a G-Set X, the Čech nerve of the unique morphism $X \to \top$ to the terminal object (in G-Set, this is the type with 1 term) sends $[n]$ to X^{n+1}. Now, considering G as a G-set, acting on itself by left multiplication, EG is the Čech nerve of $G \to \top$.

Now, recall that we are not only eventually defining a complex P, but also a morphism to $k[0]$ which we want to show is a homotopy equivalence. Note that since $k[0]$ is only nontrivial in degree 0, such a morphism is determined by a map $f_0 : P_0 \to k$ and a proof that $f_0 \circ d_0 = 0$, where d_0 is the last differential in P. Call the data of a chain complex and a morphism to a complex concentrated in degree 0 an "augmented chain complex"; analogously, an augmented simplicial object is a simplicial object X plus a morphism from $X([0])$ to some object Y satisfying a similar property. In a "nice" enough category, there is a natural augmentation of any simplicial object through which all other augmentations factor, which in the case of EG is given by the map $G \to \top$.

We have said that P being homotopy equivalent to $k[0]$ corresponds to $|EG|$ being contractible in the topological world; the analogue of contractibility for a simplicial object is that its natural augmentation has an **extra degeneracy**. Given a simplicial object X augmented by $f_0 : X([0]) \to Y$, an extra degeneracy is a family of maps $s : Y \to X([0])$ and $s_n : X([n]) \to X([n+1])$ for $n \geq 0$ satisfying certain properties, listed in [5, p. 200].

Now, it is a fact that the natural augmentation of the Čech nerve of a split epimorphism has an extra degeneracy 🔗, as formalised by Joël Riou. For our map of interest, $G \to \top$, to be a split epimorphism, there must be a morphism $\tau : \top \to G$ such that $\tau \circ \epsilon = \mathrm{id}$. But no such map of G-sets exists; unless G is trivial, any function $\top \to G$ does not respect the action of G. So we will have to compose EG with the forgetful functor to Set (it simply forgets the G-action), thus giving us an extra degeneracy for EG as an augmented simplicial *set*. But this is still sufficient for our purposes.

Indeed, when we forget the G-action, the resulting simplicial set is still a Čech nerve, so has an extra degeneracy. Then, when we compose with the free k-module functor, the resulting simplicial k-module is no longer a Čech nerve. Thus discarding the k-action initially was necessary for the proof strategy, and not just for the sake of generality. But it still has an extra degeneracy, as these are preserved by any functor.

Now that we are in an abelian category, we can take the alternating face map complex. The result is our standard resolution P as a complex of k-modules – we have forgotten the representation structure. Given an augmented simplicial object with an extra degeneracy, the natural augmentation of the resulting alternating face map complex is a homotopy equivalence, as formalised here 🔗, by Joël Riou. Applying this gives us a homotopy equivalence, and hence a quasi-isomorphism, of complexes of k-modules between P and $k[0]$. We need to upgrade this to a quasi-isomorphism of complexes of representations. But this amounts to showing our map of k-modules $P_0 \to k$ comes from a map of representations, and then checking properties determined on the level of sets – hence since they hold on the level of k-modules, due to our quasi-isomorphism, they also hold in G-Rep$_k$, and we are fine.

With all this in place, we can define the chain complex `group_cohomology.resolution` and its quasi-isomorphism. We define the complex to be the alternating face map complex of EG composed with the "linearisation functor" from G-Set to G-Rep$_k$, which is induced by the free k-module functor on Set.

```
def group_cohomology.resolution :=
(algebraic_topology.alternating_face_map_complex (Rep k G)).obj
  (classifying_space_universal_cover G ⋙ (Rep.linearization k G).1.1)
```

Given this definition, the objects in the complex are definitionally isomorphic to $k[G^{n+1}]$, and `simp` proves that the differential agrees with (3).

We note that up to here we have only required G to be a monoid.

3.3 Freeness of $k[G^{n+1}]$

The main remaining task is to show the objects in the resolution are projective, and for this we shall need G to be a group. Since they are not only projective, but in fact free, we show this instead. This is the only place we will use the category of $k[G]$-modules, for its free object API. We do this by first constructing the isomorphism

$$k[G] \otimes_k k[G^n] \cong k[G^{n+1}] \tag{4}$$

as representations, where the representations on $k[G^{n+1}]$ and $k[G]$ are induced by left multiplication of G, whilst $k[G^n]$ has the trivial representation. Then, passing to the $k[G]$-module category, we can send the natural k-basis of $k[G^n]$ to a $k[G]$-basis of $k[G] \otimes_k k[G^n]$, and transport this across the isomorphism. The author constructed (4) twice; first at the very start of the project, and secondly whilst writing this paper. We will review each formalisation, and compare them. The crux of the original formalisation is here 🔗, with the rest here 🔗 and here 🔗. The new formalisation is here 🔗.

Originally, we defined a map $G^n \to G^{n+1}$ which sends

$$(g_1, \dots, g_n) \mapsto (1, g_1, g_1g_2, \dots, g_1 \dots g_n),$$

and extended this to a k-linear map $k[G] \otimes_k k[G^n] \to k[G^{n+1}]$ that sends $g \otimes (g_1, \dots, g_n)$ to $g \cdot (1, g_1, g_1g_2, \dots, g_1 \dots g_n)$, in `of_tensor_aux`.

The type of a morphism in G-Rep$_k$ is a structure with two fields: a k-module morphism, and a proof it is compatible with the representations. If we put `of_tensor_aux` in the first field and then try to prove the statement in the second field, we get timeouts when using common tactics like `dsimp` (performs some definitional reduction, typically making the goal easier to read) and `simp`. Thus we prove the required compatibility result in a separate lemma. This difficulty surprised the author, as the objects involved seemed relatively low level. However, we are marrying some category-theoretic material (the definition of G-Rep$_k$ morphisms) and some non-category-theoretic material (everything else) – a task which has seemed to cause basic tactics to time out at other points in the project too. Meanwhile, we can state the separate lemma without category-theoretic terms, so we can prove it with our usual tactics.

Similarly, we define the inverse map `to_tensor`, which sends

$$(g_0, g_1, \dots, g_n) \mapsto g_0 \otimes (g_0^{-1}g_1, \dots, g_0^{-1}g_n),$$

and again factor out the proof of compatibility. We must also prove the two maps are left and right inverse to one another, facts we cannot leave to automation but which are not too troublesome to prove. The only other awkwardness in this formalisation was organisation:

having to name the underlying k-linear maps separately (suffixed with `aux`) and deciding when to add API for the auxiliary k-linear maps or for the G-Rep$_k$ morphisms `of_tensor, to_tensor`.

In the refactor, we instead define an isomorphism of G-sets $G \times G^n \cong G^{n+1}$, with G acting by left multiplication on G^{n+1} and G but trivially on G^n. We then apply the linearisation functor G-Set $\to$ G-Rep$_k$. But G-Set and G-Rep$_k$ are monoidal categories – they have a binary operation $\otimes$ on objects satisfying certain properties. In G-Set, $\otimes$ is induced by $\times$ on the underlying sets, and in G-Rep$_k$, $\otimes$ is induced by the usual tensor product $\otimes_k$. The linearisation functor is monoidal, meaning it commutes with $\otimes$, so we end up with $k[G] \otimes_k k[G^n] \cong k[G^{n+1}]$ as before. This approach uses more abstract tools already in `mathlib` than the first, and means we no longer have to prove that the resulting maps define morphisms in G-Rep$_k$. Moreover, the construction itself is more general: we define it in the simpler category G-Set, and instead of using the somewhat messy functions of the previous formalisation, we assemble it inductively from some more general building blocks. For example, for a G-set X, we define $G \times X \cong G \times X$ where G acts on the first X by the G-action `X.`ρ but on the second X trivially: the map sends $(g, x) \mapsto (g,\, \rho(g^{-1})(x))$.

However, the work left for us to do has changed: now we need to prove that the resulting isomorphism agrees with those messy maps from before. The proofs are not painless: we are proving something non-category-theoretic about objects constructed with heavy dependence on the category theory library, and as in the first version, this means avoiding `dsimp`. However, `simp` was useful when used appropriately. Additionally, these lemmas are not being factored out of some structure field or proved about some auxiliary function, so we avoid the ugly duplication of the first approach. Finally, the refactor improves performance. In the original formalisation, the representation morphism $k[G^{n+1}] \to k[G] \otimes_k k[G^n]$ takes 27 seconds to compile on the author's machine. Similarly, Lean is slow to elaborate the lemmas describing how the isormorphism acts on simple elements, despite the proofs being one line (simply using the corresponding lemmas about the underlying k-linear maps). Naïvely applying said corresponding lemmas takes about a minute to compile (on the author's machine). However, we can speed up the lemmas (though not the morphism's definition) by prefixing their proofs with `by apply`. Writing `by apply foo` achieves the same thing as writing `(foo : _)`; it makes Lean elaborate `foo` without an expected type. Otherwise, when compiling the old lemmas, it seems Lean spends too much time struggling with unification.

In the new formalisation, meanwhile, the isomorphism compiles in 5 seconds, and the lemmas describing its action on simple elements take less than 5 seconds, despite the proofs being more involved.

Regardless of its construction, given this isomorphism, we can now pass to the category of $k[G]$-modules, to transport a $k[G]$-basis of $k[G] \otimes_k k[G^n]$ across to $k[G^{n+1}]$. This requires some care, though; the category equivalence sends a G-Rep$_k$ M to a type alias `M.`ρ`.as_module`, equipped with the $k[G]$-module instance defined $\sum n_i g_i \cdot v := \sum n_i \cdot \rho(g_i)(v)$. But taking `as_module` of the lefthand side of the isomorphism gives a $k[G]$-module structure which is only propositionally equal to the one we want in order to use the relevant $k[G]$-module API. Instead, since there is a k-module isomorphism underlying (4), we use this to define the *functions* in our $k[G]$-module isomorphism, and then prove that this commutes with the $k[G]$-action we actually want. This allows us to define the $k[G]$-basis as desired.

We have a few loose ends (a collection 🔗 of various category-theoretic details) to tie up before we can assemble our results into a term of type `ProjectiveResolution k`. These concern how certain functors interact with projectiveness and quasi-isomorphisms, and were not much trouble to formalise. Bringing together everything we have done so far allows us to define `group_cohomology.ProjectiveResolution` as hoped.

```
def group_cohomology.ProjectiveResolution :
  ProjectiveResolution (Rep.trivial k G k) :=
(ε_to_single₀ k G).to_single₀_ProjectiveResolution (X_projective k G)
```

4 Defining group cohomology

We can immediately give one definition of group cohomology. The isomorphism `functor.left_derived_obj_iso` shows that applying $\mathrm{Hom}(-, M)$ to our resolution and taking cohomology calculates $\mathrm{Ext}_{G\text{-}\mathrm{Rep}_k}(k, M)$. However, before we can finish the definition, Lean times out:

```
def group_cohomology.Ext_iso (M : Rep k G) (n : ℕ) :
  ((Ext k (Rep k G) n).obj ...).obj M ≅ ... := sorry
```

(where we omit opaque code, and the tactic `sorry` allows us to leave a declaration unfinished without (typically) giving an error). This is strange; when we replace the `sorry` with the correct isomorphism, Lean no longer times out. Meanwhile, replacing `def` with `lemma` also stops the timeout. This is what is known as the `def/lemma` issue: Lean will try and work out whether a definition is computable, even if we mark it as noncomputable, as we have done here. It is this computability check which is timing out. On the other hand, since Lean is proof irrelevant, it does not check lemmas are computable, so temporarily making the definition a lemma fixes the issue. There is now a less ad-hoc solution to this problem, due to Gabriel Ebner: prefixing a definition with `noncomputable!` will force it to be noncomputable before we have filled in the `sorry`; see [3, p. 16] for details.

But the isomorphism we really want is between cohomology of the complex in (1) and the Ext groups. First, we need an isomorphism of the objects in each complex; recall that this will come from (2). We have already defined the other isomorphism needed, $k[G] \otimes_k k[G^n] \cong k[G^{n+1}]$, when constructing a basis. Meanwhile, (2) relies on an adjunction of functors. There is some module API which would be useful for defining the adjunction, but (despite the author's, at this point, misguided efforts), this is still an inappropriate place to be working in $k[G]$-Mod, for the reasons discussed earlier 🔗. We should instead generalise, and work with the notion of a monoidal closed category.

We omit the mathematical details involved; the code can be found here 🔗. Concisely, by defining a monoidal closed instance on G-Rep_k, we get the desired adjunction – but then we must prove it behaves as it should. This means showing that when we evaluate on elements, the abstract, category-theoretic maps in our adjunction agree with the maps in the tensor-hom adjunction for k-modules, which are simpler and not defined with category theory. As in the refactor of the isomorphism $k[G] \otimes_k k[G^n] \cong k[G^{n+1}]$, we are relating a structure many layers deep in the category theory library with much simpler objects. In similar situations prior, the author had accepted slow compilation times as an inevitable part of life, without learning the "art" of using the category theory library. By this point such an approach no longer worked: `dsimp` and `simp` would often time out, or were otherwise unusably slow, and the lemmas could not be stated without use of category theory, unlike when originally defining the morphisms in (4). Moreover, the goals were too complicated to sanely close without any automation. As far as the author could tell, the user must restrict their range of proof techniques and appeal only to syntactic equalities: this means adding `rfl`-lemmas (lemmas which are true by definition, i.e. whose proof is `rfl`) to the library and rewriting these, outsourcing the work of `dsimp` to `simp` and to the user themselves. On the

occasions `dsimp` does work, it would help to be able to ask which definitional equalities it applied, so the user can replace its use (since it was slowing down proofs so badly) with the corresponding `rfl`-lemmas. The tactic `squeeze_simp` tells us this regarding `simp`, and is very useful; the corresponding tactic `squeeze_dsimp` almost never works. Nonetheless, the desired results were provable and didactic for developing an instinct on how to work with category theory in Lean.

Given this, we define the complex of inhomogeneous cochains in the simplest way – by essentially translating (1) directly into Lean (with k instead of $\mathbb{Z}$). However, we also prove that each differential $\mathrm{Fun}(G^n, M) \to \mathrm{Fun}(G^{n+1}, M)$ agrees with

$$\mathrm{Fun}(G^n, M) \xrightarrow{\sim} \mathrm{Hom}_{G\text{-}\mathrm{Rep}_k}(k[G^{n+1}], M) \xrightarrow{-\circ d_n} \mathrm{Hom}_{G\text{-}\mathrm{Rep}_k}(k[G^{n+2}], M) \xrightarrow{\sim} \mathrm{Fun}(G^{n+1}, M)$$

where d_n is the differential in the standard resolution. This gives us for free the proof that the composition of two differentials is zero, which to do directly is somewhat onerous, as seen in work of Shenyang Wu 🔗.

With this done, there is one more obstacle to defining the isomorphism between "concrete group cohomology" and the Ext groups. On one side we have cohomology of a complex with objects in k-Mod, and on the other we have homology of a chain complex with objects in the opposite category $k\text{-Mod}^{op}$ – an instance of something we do not think about in real life but which takes a non-trivial amount of code 🔗 to formalise. However, the process was straightforward, and allows us to define group cohomology, here 🔗.

```
def group_cohomology [group G] (A : Rep k G) (n : ℕ) :=
(inhomogeneous_cochains A).homology n
```

5 Conclusion and future work

In real life, keeping exposition of a mathematical concept self-contained is a good thing, and group cohomology is quite amenable to this. But the trajectory of our project demonstrates just how irrelevant this quality is as an aim when contributing to `mathlib`. Indeed, the need to prioritise generality, to keep the growth of a library sustainable, means recognising and exploiting as many connections between different mathematical objects as possible, in search of concepts' common principles and structural "ancestors".

But the quest for abstraction has to stop somewhere. All of this material has been merged with `mathlib`: we conclude it is possible to honour the maxims of `mathlib` design to a considerable extent, and still connect the resulting convoluted, abstract definition of group cohomology with the down to earth definition used for computation. The power afforded by results in the category theory library can, in practice, interact with `mathlib`'s lower-level objects.

However, we have illustrated this statement's caveats. Currently, it seems to the author that there is an "art" to using the category theory library, meaning it can be frustrating to work with for the naïve user. Of course, exactly the same could be said of Lean in general; a learning curve is unavoidable. But as more people apply category theory to simpler structures in the library, documentation of this "art" will increase.

Alternatively, the new version of Lean, Lean 4, promises many advances in performance, and this could make it easier to use automation with category theory. This project was done in Lean 3, which has been the most current version for most of Lean's history. But this is changing – `mathlib` is currently being ported from Lean 3 to Lean 4: a huge undertaking. When our project eventually transitions to Lean 4, we suspect it may look very different, with the obstacles outlined in this paper perhaps diminished.

In the meantime, there is considerably more to be added before `mathlib` has the facts about group cohomology taught in a typical introductory course. Most of these are proved using the concrete formulation, and the author has a repository containing work in this direction. Because there is not much structure involved, the high-powered tools of category theory are irrelevant, making the results simple to formalise. Thus, although this code has not been tidied up with `mathlib` in mind, the author is fairly confident it will not go through as many reformulations as the rest of the material so far.

We have written an API for cohomology in degree $n \leq 2$, and will open a pull request for this soon. We have also shown that given a group homomorphism $f : G \to H$, a G-representation A, an H-representation B and a k-linear map $\phi : B \to A$ such that $\phi(\rho_B(f(g))(x)) = \rho_A(g)(\phi(x))$ for all $g \in G, x \in B$, then we get an induced k-linear map of cohomology groups $\mathrm{H}^n(H, B) \to \mathrm{H}^n(G, A)$ for all n. We used this to formalise the "inflation-restriction" exact sequence, and have also formalised Hilbert's theorem 90, along with the fact that $\mathrm{H}^1(G, A) \cong \mathrm{Hom}(G, A)$ when the G-action on A is trivial, and about half of the work in using H^2 to classify group extensions. The code is fast enough and readable; the only disappointment is that in real life we can view a group's operation as either multiplicative or additive when convenient, and this is messier in Lean. A representation A is an additive group with an action of a multiplicative group, so to describe the set of group morphisms $\mathrm{Hom}(G, A)$ we must write `G →* multiplicative A`, for example.

Similarly, we have done some non-`mathlib`-style work on Galois cohomology. Many nice group cohomology facts assume G is finite; Galois groups are profinite, meaning they are limits of families of finite groups. Proving that the group cohomology of a profinite group is a limit of the cohomology of the constituent finite groups lets us extend results to Galois groups, and this is used everywhere in algebraic number theory. To formalise this, the author has proved some of the requisite topological group facts; similar to the concrete group cohomology API, the code was enjoyable to write. Instead of complicated definitions, it involves complicated proofs, which can be preferable in Lean.

A different story is the remaining abstract material, like universal delta functors and spectral sequences. Spectral sequences [9, Chapter 20, Section 9] will require considerable work to define at all, let alone in a `mathlib`-compatible way. These are necessary to compare cohomology as we vary the group G via the Lyndon-Hochschild-Serre spectral sequence [6, p. 8] – an extension of the "inflation-restriction" exact sequence. We need delta functors, meanwhile, to finish setting up Galois cohomology [12, Tag 0DVG]. They are also needed to show that group cohomology agrees with Ext as functors, and not just in their action on objects [9, Chapter 20, Section 8]. Happily, though, the requisite delta functor material is done, in the Liquid Tensor Experiment 🔗; it has just not been readied for `mathlib`, and given its abstract nature this process may be non-trivial.

All of the abstract material, and essentially any group cohomological results concerning $\mathrm{H}^n(G, M)$ for general n, rely on long exact sequences. These, too, are defined in the Liquid Tensor Experiment, and should be usable in our work after proving a couple of easy, concrete lemmas. But even preparing these for `mathlib` raises challenges – when using them in real life we rely on drawing diagrams, and the clarity this affords is lost when translated into Lean. This file 🔗 gives some demonstration of what "diagram chasing" can look like in Lean 3. However, Wojciech Nawrocki is working on a widget 🔗 for Lean 4 which displays commutative diagrams in the goal state; maybe this will help us chase diagrams in the future.

References

1. Anne Baanen. Use and Abuse of Instance Parameters in the Lean Mathematical Library. 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 4:1–4:20, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ITP.2022.4.
2. John W. S. Cassels and Albrecht Fröhlich. *Algebraic Number Theory*. London Mathematical Society, London, 2010.
3. María Inés de Frutos-Fernández. Formalizing the Ring of Adèles of a Global Field. In *13th International Conference on Interactive Theorem Proving (ITP 2022)*, volume 237, pages 14:1–14:18, 2022. URL: https://drops.dagstuhl.de/opus/volltexte/2022/16723/pdf/LIPIcs-ITP-2022-14.pdf.
4. César Domínguez and Julio Rubio. Computing in coq with infinite algebraic data structures. In *Proceedings of the 10th ASIC and 9th MKM International Conference, and 17th Calculemus Conference on Intelligent Computer Mathematics*, volume 6167, April 2010. doi:10.1007/978-3-642-14128-7_18.
5. Paul G. Goerss and John F. Jardine. *Simplicial Homotopy Theory*. Springer Science & Business Media, 2009. doi:10.1007/978-3-0346-0189-4.
6. Gerhard Hochschild and Jean-Pierre Serre. Cohomology of group extensions. *Transactions of the American Mathematical Society*, 74(1):110–134, 1953. doi:10.1090/S0002-9947-1953-0052438-8.
7. Shin-ichi Katayama. Diophantine Equations and Hilbert's Theorem 90. *Journal of mathematics, the University of Tokushima*, 48:35–40, 2014. URL: https://cir.nii.ac.jp/crid/1574231877578024320.
8. Ernst E. Kummer. Über eine besondere Art, aus complexen Einheiten gebildeter Ausdrücke. *Journal für die reine und angewandte Mathematik*, 1855(50):212–232, 1855. doi:10.1515/crll.1855.50.212.
9. Serge Lang. *Algebra*, volume 211. Springer Science & Business Media, 2012. doi:10.1007/978-1-4613-0041-0.
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.
11. nLab authors. Moore complex. https://ncatlab.org/nlab/show/Moore+complex, February 2023. Revision 59.
12. The Stacks project authors. The stacks project. https://stacks.math.columbia.edu, 2023.
13. Charles A. Weibel. *An Introduction to Homological Algebra*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1994. doi:10.1017/CBO9781139644136.
14. Charles A. Weibel. History of Homological Algebra. In Ioan M. James, editor, *History of Topology*, chapter 28, pages 797–836. North Holland, 1999.

A Formalisation of Gallagher's Ergodic Theorem

Oliver Nash
Imperial College London, UK

Abstract

Gallagher's ergodic theorem is a result in metric number theory. It states that the approximation of real numbers by rational numbers obeys a striking "all or nothing" behaviour. We discuss a formalisation of this result in the Lean theorem prover. As well as being notable in its own right, the result is a key preliminary, required for Koukoulopoulos and Maynard's stunning recent proof of the Duffin-Schaeffer conjecture.

2012 ACM Subject Classification Mathematics of computing → Probability and statistics

Keywords and phrases Lean proof assistant, measure theory, metric number theory, ergodicity, Gallagher's theorem, Duffin-Schaeffer conjecture

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.23

Supplementary Material *Software (Source Code)*: `https://github.com/leanprover-community/mathlib`, archived at `swh:1:dir:a022d44847c12ca961b4bd0c8bca4f9df70cf7df`

Acknowledgements It is a pleasure to thank Andrew Pollington who suggested this project during the conference Lean for the Curious Mathematician, held at Brown University (ICERM) in 2022. I also wish to thank Anatole Dedecker, Heather Macbeth, Patrick Massot, and Junyan Xu, all of whom were of direct assistance. Lastly I especially wish to thank Sébastien Gouëzel for many helpful suggestions and Kevin Buzzard for many useful conversations.

1 Introduction

In addition to recognising extraordinary achievements of young mathematicians, the Fields Medal provides a valuable service to the wider mathematical community: it draws attention to important recent results. In recent years, such attention has had a significant positive impact in the formalisation community. Buzzard, Commelin, and Massot's formalisation of the definition of a perfectoid space [4] and Commelin, Topaz et al.'s spectacular success with the Liquid Tensor Experiment [6] were both the results of projects which formalised work of 2018 Fields Medalist Peter Scholze. Amongst other things, these projects demonstrate that today's proof assistants are capable of handling the complicated constructions of contemporary mathematics.

In 2022, James Maynard was awarded a Fields Medal with a citation that highlighted his work on the structure of prime numbers as well as on *Diophantine approximation*. In the long form of the citation we read that:

> Maynard has also produced fundamental work in Diophantine approximation, having solved the Duffin-Schaeffer conjecture with Koukoulopoulos.

Shortly after the announcement of Maynard's award, Andrew Pollington suggested to the author that a formalisation of Koukoulopoulos and Maynard's proof of the Duffin-Schaeffer conjecture would be a worthy target for formalisation. Recognising that this would be an enormous undertaking, he suggested focusing on various necessary preliminaries. Perhaps the most important of these is Gallagher's ergodic theorem [8] (see also Koukoulopoulos and Maynard [12] lemma 5.1). The statement is as follows:


© Oliver Nash;
licensed under Creative Commons License CC-BY 4.0

▶ **Theorem 1** (Gallagher's theorem). *Let $\delta_1, \delta_2, \dots$ be a sequence of real numbers and let:*

$$W = \{x \in \mathbb{R} \mid \exists\ q \in \mathbb{Q}, |x - q| < \delta_{\mathrm{denom}(q)}\ \textit{i.o.}\}.$$

Then W is almost equal to either $\emptyset$ or $\mathbb{R}$.

This deserves a few remarks:

- The notation $\mathrm{denom}(q)$ denotes the denominator of q (in lowest terms). It is a strictly positive natural number.
- Special attention should be paid to the letters "i.o." appearing in the definition of W: these abbreviate the phrase "infinitely often". The notation means that $x \in W$ iff there exists an *infinite sequence* of rationals $q_0, q_1, \dots$ (which may depend on x) with $\mathrm{denom}(q_0) < \mathrm{denom}(q_1) < \cdots$ satisfying $|x - q_i| < \delta_{\mathrm{denom}(q_i)}$ for all i.
- The phrase "almost equal" characterises this as a theorem of *metric number theory*: it means that the sets are equal up to a set of Lebesgue measure zero.
- It is striking and not at all obvious that W should exhibit such dichotomous behaviour.

It is the purpose of this article to discuss the author's formalisation of Gallagher's theorem. It was carried out using the Lean proof assistant together with its `mathlib` library [14]. More precisely we formalised the following:

Listing 1 Gallagher's theorem

```
theorem add_well_approximable_ae_empty_or_univ
  (δ : ℕ → ℝ) (hδ : tendsto δ at_top (𝓝 0)) :
  (∀ᵐ x, ¬ add_well_approximable 𝕊 δ x) ∨ ∀ᵐ x, add_well_approximable 𝕊 δ x :=
```

The notation will be explained in the pages to come. With further work, one could drop the hypothesis $h\delta$, which says that $\delta_n \to 0$ as $n \to \infty$. In fact this highlights a curious feature of the proof: one makes two totally separate arguments, a measure-theoretic argument which assumes the hypothesis $h\delta$ and a number-theoretic argument assuming its negation. One then invokes the law of excluded middle to deduce the result unconditionally. The measure-theoretic argument assuming $h\delta$ is much harder and is what we have formalised.

The paper is intended for non-experts and the structure is as follows. In section 2 we outline relevant basic concepts so that we can reinterpret Gallagher's theorem as a result about the lim sup of thickenings of finite-order points in the circle. We also make some general remarks about metric number theory. In section 3 we introduce the most important foundational result required. Lebesgue's density theorem is the workhorse of Gallagher's proof. In section 4 we discuss a key measure-theoretic lemma due to Cassels which is of some independent interest in its own right. In section 5 we discuss the results about ergodic maps which we needed, emphasising the ergodicity of certain maps of the circle. In section 6 we introduce points of approximately finite order and use this language to give a proof of Gallagher's theorem. We finish with section 7 where, amongst other things, we discuss further directions this work could be taken. For the most part we do not enter into the details of proofs. The main exception to this is the proof of Gallagher's theorem itself since we hope our presentation of Gallagher's ideas may make them more accessible than other accounts intended for specialists (such as [8] Theorem 1 or [11] Theorem 2.7(B)).

In keeping with `mathlib`'s stress on mathematical unity, all work was added directly to `mathlib`'s master branch in a series of 27 pull requests, collectively adding just over 3,500 net new lines of code. This work is thus automatically available to all future `mathlib` users. Throughout this text we also provide permalinks to relevant locations in `mathlib`; each one is indicated with the symbol ⧉. We also provide a judiciously chosen set of code listings (such as listing 1 above) containing Lean code. Often our intention is to assist the reader who wishes to compare a key informal statement with its formal equivalent.

2 Basic concepts

We outline some basic concepts to fix notation and to assist non-experts.

2.1 Almost equal sets

Given a measurable space with measure μ, when there is no possibility of ambiguity about the measure, we shall use the notation:

$$s =_{a.e.} t, \tag{1}$$

to say two subsets s, t are almost equal with respect to μ. We recall that this is equivalent to the following pair of measure-zero conditions:

$$\mu(s \setminus t) = 0 \quad \text{and} \quad \mu(t \setminus s) = 0. \tag{2}$$

2.2 Obeying a condition infinitely often

Gallagher's theorem concerns a set of points obeying a condition "infinitely often". In general, given a sequence of subsets $s_0, s_1, \ldots$ of some background type X the expression $\exists \cdots i.o.$ is used to define[1]:

$$\{x : X \mid \exists\, n \in \mathbb{N}, x \in s_n \ i.o.\} = \{x : X \mid \text{the set } \{n \in \mathbb{N} \mid x \in s_n\} \text{ is infinite}\}. \tag{3}$$

In fact there is another expression for this set; it is easy to see that:

$$\{x : X \mid \exists\, n \in \mathbb{N}, x \in s_n \ i.o.\} = \limsup s, \tag{4}$$

where:

$$\limsup s = \bigcap_{n \geq 0} \bigcup_{i \geq n} s_i.$$

When formalising a result about the set of points belonging to some family of subsets infinitely often, one can thus phrase it in language of (3) or in the language of lim sup. We opted for the latter. This was preferable because lim sup makes sense for any complete lattice whereas (3) is specific to the lattice of subsets of a type. All API developed was thus more widely applicable.

In the course of the proof it is useful to work with the lim sup bounded by a predicate $p : \mathbb{N} \to \text{Prop}$. This can be defined:

$$\limsup_p s = \bigcap_{n \geq 0} \bigcup_{p(i), i \geq n} s_i. \tag{5}$$

The actual definition which we added `filter.blimsup` is slightly different so that it also applies in a conditionally complete lattice (such as $\mathbb{R}$) but we provided a lemma showing the equivalence to (5) for complete lattices (such as `set` $\mathbb{R}$). Using `blimsup`, we can work with the lim sup of the subfamily defined by the predicate p without having to pass to the subtype of the indexing type. For example given two predicates p, q, we can express the useful identity:

$$\limsup_{p \vee q} s = \limsup_p s \cup \limsup_q s, \tag{6}$$

formally as:

[1] This is standard notation appearing throughout the informal literature.

Listing 2 lim sup bounded by the logical or of two predicates

```
@[simp] lemma blimsup_or_eq_sup :
  blimsup u f (λ x, p x ∨ q x) = blimsup u f p ⊔ blimsup u f q :=
```

without involving any subtypes. This is a standard design pattern used throughout `mathlib`.

2.3 Thickenings

Using the language introduced above, the set W appearing in the statement of theorem 1 may be defined as:

$$W = \limsup_{n>0} s,$$

where:

$$s_n = \{x \in \mathbb{R} \mid \exists\, q \in \mathbb{Q}, \mathrm{denom}(q) = n, |x - q| < \delta_n\}.$$

These subsets s_n have a special form: they are *thickenings*. In general, given a metric space X, if $s \subseteq X$ and $\delta \in \mathbb{R}$, the (open) δ-thickening of s is:

$$\mathrm{Th}(\delta, s) = \{x \in X \mid \exists\, y \in s, d(x, y) < \delta\}.$$

This generalises the concept of an open ball. Fortunately thickenings already existed in `mathlib` thanks to the work of Gouezel on the Gromov-Hausdorff metric [9]. We can thus express W as:

$$W = \limsup_{n>0} \mathrm{Th}(\delta_n, \{q \in \mathbb{Q} \mid \mathrm{denom}(q) = n\}).$$

Using the language of thickenings turned out to be very convenient formally, not just for Gallagher's theorem but also for example in lemma 5 (discussed below).

2.4 The circle as a normed group

The subset W appearing in theorem 1 trivially satisfies the periodicity condition[2]:

$$1 + W = W,$$

and thus descends to a subset of the circle $\mathbb{S} = \mathbb{R}/\mathbb{Z}$. This quotient is actually a *normed* group. We recall that a normed (additive) group G is a group carrying a real-valued function:

$$G \to \mathbb{R}$$
$$g \mapsto \|g\|$$

such that the distance function $d(g, h) = \|g - h\|$ satisfies the metric space axioms.

In our case, given $x \in \mathbb{R}$ representing the coset $\hat{x} \in \mathbb{S}$, the norm obeys:

$$\|\hat{x}\| = |x - \mathrm{round}(x)| \tag{7}$$

where $\mathrm{round}(x)$ is the nearest integer to x. Since the norm gives us a metric, we may speak of thickenings of subsets of $\mathbb{S}$.

[2] If q approximates x then $1 + q$ approximates $1 + x$ with the same error and $\mathrm{denom}(1 + q) = \mathrm{denom}(q)$.

Furthermore, $\mathbb{Q}/\mathbb{Z} \subseteq \mathbb{S}$ is exactly the set of points of finite order in $\mathbb{S}$. Gallagher's theorem may thus be regarded as establishing a special property enjoyed by the circle $\mathbb{S}$ in the category of normed groups. As we shall see this is a useful point of view since the proof of Gallagher's theorem depends on the fact that certain transformations are ergodic when regarded as maps $\mathbb{S} \to \mathbb{S}$.

When this work began, `mathlib` already contained a model of the circle as complex numbers of unit length, called `circle`. Although this model is naturally equivalent to $\mathbb{R}/\mathbb{Z}$, the equivalence uses the exponential and logarithm maps which are irrelevant for our work. We thus introduced a second model called `add_circle` defined to be $\mathbb{R}/\mathbb{Z}$:

Listing 3 The additive circle

```
def add_circle {𝕂 : Type*} [linear_ordered_add_comm_group 𝕂]
  [topological_space 𝕂] [order_topology 𝕂] (p : 𝕂) :=
𝕂 ⧸ zmultiples p
```

When $p = 1$ this is exactly $\mathbb{R}/\mathbb{Z}$ but we allow a general value of p to support other applications[3].

As the names suggest, `circle` carries an instance of `mathlib`'s `group` class and `add_circle` carries an instance of `add_group`. It is interesting that `mathlib`'s additive-multiplicative design pattern so conveniently allows both models to coexist.

Substantial API for `add_circle` was then developed, notably an instance of the class `normed_add_comm_group` satisfying the identity (7) and a characterisation of the finite-order points using rational numbers:

$$\{y \in \mathbb{S} \mid o(y) = n\} = \{[q] \in \mathbb{S} \mid q \in \mathbb{Q}, \operatorname{denom}(q) = n\}, \tag{8}$$

where the notation $o(y) = n$ means that y has order n.

Using all of our new language, the statement of Gallagher's theorem becomes:

▶ **Theorem 2** (Gallagher's theorem). *Let $\delta_1, \delta_2, \ldots$ be a sequence of real numbers and let:*

$$W = \limsup_{n>0} \operatorname{Th}(\delta_n, \{y \in \mathbb{S} \mid o(y) = n\}).$$

Then $W =_{a.e.} \emptyset$ or $W =_{a.e.} \mathbb{S}$.

Using (7) and (8), theorem 2 is trivially equivalent to theorem 1.

2.5 Metric number theory

Metric number theory is the study of arithmetic properties of the real numbers (and related spaces) which hold "almost everywhere" with respect to the Lebesgue measure. The arithmetic property in the case of Gallagher's theorem is approximation by rational numbers.

To illustrate, consider the set $\mathbb{I}$ of real numbers which have infinitely-many quadratically-close rational approximations:

$$\begin{aligned}\mathbb{I} &= \{x \in \mathbb{R} \mid \exists\, q \in \mathbb{Q}, |x - q| < 1/\operatorname{denom}(q)^2 \ i.o.\} \\ &= \limsup_{n>0} \operatorname{Th}(1/n^2, \{q \in \mathbb{Q} \mid \operatorname{denom}(q) = n\}).\end{aligned}$$

[3] For example `mathlib` uses $p = 2\pi$ to define angles.

It has been known at least since the early 19$^{\text{th}}$ Century, that $\mathbb{I}$ is just the set of irrational numbers[4]:

$$\mathbb{I} = \mathbb{R} \setminus \mathbb{Q}. \tag{9}$$

Considering this result from the point of view of metric number theory, we notice that since $\mathbb{Q}$ has Lebesgue measure zero, $\mathbb{I}$ is almost equal to $\mathbb{R}$. Thus the metric number theorist would be content to summarise (9) by saying that

$$\mathbb{I} =_{a.e.} \mathbb{R},$$

without worrying about exactly which numbers $\mathbb{I}$ contains.

The benefit of metric number theorist's point of view is that a great many questions have answers of this shape. Gallagher's theorem is an especially-beautiful example of this phenomenon.

3 Doubling measures and Lebesgue's density theorem

Lebesgue's density theorem is a foundational result in measure theory, required for the proof of Gallagher's theorem. Although we only needed to apply it to the circle, the density theorem holds quite generally and so we took some trouble to formalise it subject to quite weak assumptions[5].

3.1 Doubling measures

A convenient class of measures for which the density theorem holds is the class of doubling measures (more precisely *uniformly locally* doubling measures).

▶ **Definition 3.** *Let X be a measurable metric space carrying a measure μ. We say μ is a doubling measure if there exists $C \geq 0$ and $\delta > 0$ such that for all $0 < \epsilon \leq \delta$ and $x \in X$:*

$$\mu(B(x, 2\epsilon)) \leq C\mu(B(x, \epsilon)).$$

where $B(x, r)$ denotes the closed ball of radius r about x.

The corresponding formal definition, which the author added to `mathlib` for the purposes of formalising the density theorem, is:

Listing 4 Definition of doubling measures

```
class is_doubling_measure
  {α : Type*} [metric_space α] [measurable_space α] (μ : measure α) :=
  (exists_measure_closed_ball_le_mul [] : ∃ (C : ℝ≥0), ∀ᶠ ε in 𝓝[>] 0, ∀ x,
    μ (closed_ball x (2 * ε)) ≤ C * μ (closed_ball x ε))
```

The parameter δ is not explicitly mentioned in the code above because we use `mathlib`'s standard notation for the concept of a predicate holding eventually along a filter.

[4] Thanks to Michael Geißer and Michael Stoll, `mathlib` knows this fact.

[5] We were lucky that Sébastien Gouëzel had recently added an extremely general theory of Vitali families which made this possible.

For our application, we needed to apply the density theorem to the Haar measure [15] on the circle. Of course this turns out to be the familiar arc-length measure and so the volume of a closed ball of radius ϵ is given by :

$$\mu(B(x, \epsilon)) = \min(1, 2\epsilon).$$

Taking $C = 2$ we thus see that the Haar measure on the circle is doubling. We registered this fact using a typeclass instance as follows:

Listing 5 The circle's doubling measure

```
instance : is_doubling_measure (volume : measure (add_circle T)) :=
```

The unit circle corresponds to taking $T = 1$, but the code allows any $T > 0$. Thanks to this instance, Lean knows that any results proved for doubling measures automatically holds for the Haar measure on the circle.

3.2 The density theorem

The version of the density theorem which we formalised is:

▶ **Theorem 4.** *Let X be a measurable metric space carrying a measure μ. Suppose that X has second-countable topology and that μ is doubling and locally finite. Let $S \subseteq X$ and $K \in \mathbb{R}$, then for almost all $x \in S$, given any sequence of points $w_0, w_1, \ldots$ and distances $\delta_0, \delta_1, \ldots$, if:*

- *$\delta_j \to 0$ as $j \to \infty$ and,*
- *$x \in B(w_j, K\delta_j)$ for large enough j,*

then:

$$\frac{\mu(S \cap B(w_j, \delta_j))}{\mu(B(w_j, \delta_j))} \to 1,$$

as $j \to \infty$.

Even in the special case $K = 1$ and $w_0 = w_1 = \cdots = x$, the result is quite powerful[6]. A point x satisfying the property appearing in the theorem statement is known as a point of density 1. Using this language, Lebesgue's density theorem asserts that almost all points of a set have density 1. In particular if $\mu(S) > 0$ then there must exist a point of density 1 . As an example, if $X = \mathbb{R}$ and S is the closed interval $[0, 1]$, the set of points of density 1 is the open interval $(0, 1)$.

In fact the formal version which we added to `mathlib` is very slightly more general since it allows w and δ to be maps from any space carrying a filter. After a preparatory `variables` statement:

Listing 6 Variables for the density theorem

```
variables {α : Type*} [metric_space α] [measurable_space α] (μ : measure α)
  [is_doubling_measure μ] [second_countable_topology α] [borel_space α]
  [is_locally_finite_measure μ]
```

it looks like this:

[6] Indeed this is probably the most common version one finds in the literature.

Listing 7 Lebesgue's density theorem for doubling measures

```
lemma is_doubling_measure.ae_tendsto_measure_inter_div (S : set α) (K : ℝ) :
  ∀ᵐ x ∂μ.restrict S, ∀ {ι : Type*} {l : filter ι} (w : ι → α) (δ : ι → ℝ)
    (δlim : tendsto δ l (𝓝[>] 0))
    (xmem : ∀ᶠ j in l, x ∈ closed_ball (w j) (K * δ j)), tendsto (λ j,
    μ (S ∩ closed_ball (w j) (δ j)) / μ (closed_ball (w j) (δ j))) l (𝓝 1) :=
```

The method of proof is essentially to develop sufficient API for `is_doubling_measure` to show that such measure spaces carry certain natural families of subsets called Vitali families and then to invoke the lemma `vitali_family.ae_tendsto_measure_inter_div` added by Gouëzel as part of an independent project [10].

4 Cassels's lemma

A key ingredient in the proof of Gallagher's theorem is the following result due to Cassels.

▶ **Lemma 5.** *Let X be a measurable metric space carrying a measure μ. Suppose that X has second-countable topology and that μ is doubling and locally finite. Let $s_0, s_1, \ldots$ be a sequence of subsets of X and $r_0, r_1, \ldots$ be a sequence of real numbers such that $r_n \to 0$ as $n \to \infty$. For any $M > 0$ let:*

$$W_M = \limsup \mathrm{Th}(Mr_n, s_n),$$

then:

$$W_M =_{a.e.} W_1,$$

i.e., up to sets of measure zero, W_M does not depend on M.

This essentially appears as lemma 9 in [5] in the special case that:
(a) X is the open interval $(0, 1)$,
(b) μ is the Lebesgue measure,
(c) s_n is a sequence of points rather than a sequence of subsets.

Reusing the `variables` from listing 6, the formal version of lemma 5 which we added to `mathlib` looks like this:

Listing 8 Cassels's lemma

```
theorem blimsup_thickening_mul_ae_eq
  (p : ℕ → Prop) (s : ℕ → set α) {M : ℝ} (hM : 0 < M)
  (r : ℕ → ℝ) (hr : tendsto r at_top (𝓝 0)) :
  (blimsup (λ i, thickening (M * r i) (s i)) at_top p : set α) =ᵐ[μ]
  (blimsup (λ i, thickening (r i) (s i)) at_top p : set α) :=
```

Several remarks are in order:

- The syntax `s =ᵐ[μ] t` is `mathlib`'s notation for sets (or functions) s, t being almost equal with respect to a measure μ. It is the formal equivalent of the popular informal notation (1).
- The type ascriptions `: set α` appear because of an unresolved *typeclass diamond* in `mathlib`'s library of lattice theory. The issue is that the type `set α` is definitionally equal to $\alpha \to$ Prop. Since Prop is a complete boolean algebra it follows that $\alpha \to$ Prop is a complete boolean algebra. Unfortunately the definition of the complete boolean algebra structure on `set α`, though mathematically equal, is not definitionally equal

to that on $\alpha \to$ Prop. Strictly speaking, because `set` α is a type synonym, this is a permissible diamond. The point is that ideally all lemmas and statements about `set` α (including results about its lattice structure) should depend only on its API and not in its definition. However in practice such "definitional abuse" does occur and can be convenient so it would be useful to resolve the diamond.[7]

- Listing 8 is stated in terms of `blimsup`, i.e., a lim sup bounded by a predicate p. As discussed in section (2.2), this allows us to avoid having to deal with subtypes. We will see that this is convenient when applying this lemma in the proof of Gallagher's theorem.
- The key ingredient in the proof of Cassels's lemma is Lebesgue's density theorem 4. In view of (2), Cassels's lemma requires us to establish a pair of measure-zero conditions. According to whether $M < 1$ or $M > 1$, exactly one of these two conditions is trivial for the two sets appearing in the statement of Cassels's lemmma 5. To prove the non-trivial measure-zero condition, one argues by contradiction by assuming the measure is strictly positive, applying the density theorem to obtain a point of density 1, and showing that this is impossible for a doubling measure. The only non-trivial dependency is Lebesgue's density theorem.
- Although the modifications required for the generalisation of this lemma from its original form in [5] are straightforward, the generalisation (c) from points to subsets (equivalently from balls to thickenings) is extremely useful formally. In the application of this lemma required for Gallagher's theorem, s_n is the set of points of order n in the circle. In the informal literature, the version of lemma 5 for sequences of points can be applied because the circle has only finitely-many points of each finite order and so one can enumerate all points of finite order as a single sequence of points. This would be messy formally (for example it requires choosing arbitary orderings) and in any case is not necessary given the more general result.

5 Ergodic theory

Ergodic theory is the study of measure-preserving maps. Given measure spaces (X, μ_X) and (Y, μ_Y), a measurable map $f : X \to Y$ is measure-preserving if:

$$\mu_X(f^{-1}(s)) = \mu_Y(s),$$

for any measurable set $s \subseteq Y$. For example, given any $c \in \mathbb{R}$, taking Lebesgue measure on both domain and codomain, the translation $x \mapsto c + x$ is always measure-preserving whereas the dilation $x \mapsto cx$ is measure-preserving only if $c = \pm 1$. Fortunately `mathlib` already contained an excellent theory of measure-preserving maps.

5.1 Ergodic maps, general theory

Within ergodic theory, special attention is paid to ergodic maps.

▶ **Definition 6.** *Let (X, μ) be a measure space and $f : X \to X$ be measure-preserving. We say f is ergodic if for any measurable set $s \subseteq X$:*

$$f^{-1}(s) = s \implies s \text{ is almost equal to } \emptyset \text{ or } X.$$

[7] The diamond is recorded in `mathlib` issue 16932. In fact it is only the `Inf` and `Sup` fields in the complete boolean algebra structures that differ definitionally so this should be fairly easy to resolve.

Ergodicity is key concept in the proof of Gallagher's theorem and so we added the following definitions to `mathlib`:

Listing 9 Definition of pre-ergodic and ergodic maps:

```
structure pre_ergodic (μ : measure α . volume_tac) : Prop :=
  (ae_empty_or_univ : ∀ ⦃s⦄, measurable_set s →
    f⁻¹' s = s → s =ᵐ[μ] (∅ : set α) ∨ s =ᵐ[μ] univ)

structure ergodic (μ : measure α . volume_tac) extends
  measure_preserving f μ μ, pre_ergodic f μ : Prop
```

The reason for the intermediate definition `pre_ergodic` is to support the definition of quasi-ergodic maps which we also defined, but which do not concern us here.

We then developed some basic API for ergodic maps including the key result:

▶ **Lemma 7.** *Let X be a measurable space with measure μ such that $\mu(X) < \infty$. Suppose that $f : X \to X$ is ergodic, $s \subseteq X$ is measurable, and the image $f(s)$ is almost contained in s, then s is almost equal to $\emptyset$ or X.*

This result is elementary but not quite trivial and appears formally as follows:

Listing 10 Sets that are almost invariant by an ergodic map:

```
lemma ae_empty_or_univ_of_image_ae_le [is_finite_measure μ]
  (hf : ergodic f μ) (hs : measurable_set s) (hs' : f '' s ≤ᵐ[μ] s) :
  s =ᵐ[μ] (∅ : set X) ∨ s =ᵐ[μ] univ :=
```

This is not the first time that ergodic maps have been formalised in a theorem prover and so we have kept the above account very brief. Indeed the Archive of Formal Proofs for Isabelle/HOL contains an impressive body of results about ergodic theory due to Sébastien Gouëzel with contributions from Manuel Eberl, available at the Ergodic Theory entry. This entry contains many results about general ergodic theory that have not yet been added to `mathlib`. On the other hand, we needed to know that certain specific maps on the circle are ergodic and our formalisations of these results do appear to be the first of their kind. We discuss these next.

5.2 Ergodic maps on the circle

In order to prove Gallagher's theorem, we needed the following result:

▶ **Theorem 8.** *Given $n \in \mathbb{N}$, the map:*

$$\mathbb{S} \to \mathbb{S}$$

$$y \mapsto ny$$

is measure-preserving if $n \geq 1$ and is ergodic if $n \geq 2$.

The fact that $y \mapsto ny$ is measure-preserving follows from general uniqueness results for Haar measures. In fact the result holds for any compact, Abelian, divisible topological group. Thanks to `mathlib`'s extensive theory of Haar measure [15], it was easy to add a proof of this. We encourage readers who are encountering this fact for the first time to examine figure 1 and appreciate why this result holds for $\mathbb{S}$ despite failing for $\mathbb{R}$.

The proof that $y \mapsto ny$ is ergodic is harder. We proved it as corollary of the following lemma. We sketch a proof to give a sense of what is involved; it is not essential that the reader follow the details: the main point is that we needed to use Lebesgue's density theorem.

Figure 1 The map $f : y \mapsto 2y$ is measure-preserving.

▶ **Lemma 9.** *Let $s \subseteq \mathbb{S}$ be measurable and $u_0, u_1, \ldots$ be a sequence of finite-order points in $\mathbb{S}$ such that:*

- *$u_i + s$ is almost equal to s for all i,*
- *the order $o(u_i) \to \infty$ as $i \to \infty$.*

Then s is almost equal to $\emptyset$ or X.

Proof. The result is fairly intuitive: s is almost equal to $u_i + s$ iff it is composed of a collection of $o(u_i)$ components, evenly-spaced throughout the circle, up to a set of measure zero. Since this holds for all i and $o(u_i) \to \infty$, such components must either fill out the circle or be entirely absent, up to a set of measure zero.

The way to turn the above intuitive argument into rigorous proof is to use Lebesgue's density theorem 4. We must show that if s is not almost empty then $\mu(s) = 1$. Lebesgue tells us that if s is not almost empty it must contain some point d of density 1. Using d, we construct the sequence of closed balls B_i centred on d such that $\mu(B_i) = 1/o(u_i)$. Because $u_i + s$ is almost s,

$$\mu(s \cap B_i) = \mu(s)/o(u_i) = \mu(B_i)\mu(s).$$

However since d has density 1, we know that:

$$\mu(s \cap B_i)/\mu(B_i) \to 1.$$

These two results force us to conclude that $\mu(s) = 1$. ◀

The formal version is very slightly more general and appears in `mathlib` as follows:

Listing 11 Formal statement of lemma 9

```
lemma add_circle.ae_empty_or_univ_of_forall_vadd_ae_eq_self
  {s : set $ add_circle T} (hs : null_measurable_set s volume)
  {ι : Type*} {l : filter ι} [l.ne_bot] {u : ι → add_circle T}
  (hu₁ : ∀ i, ((u i) +ᵥ s : set _) =ᵐ[volume] s)
  (hu₂ : tendsto (add_order_of ∘ u) l at_top) :
  s =ᵐ[volume] (∅ : set $ add_circle T) ∨ s =ᵐ[volume] univ :=
```

Theorem 8 follows from lemma 9 because any set s satisfying $f^{-1}(s) = s$ for $f : y \mapsto ny$ satisfies $u_i + s = s$ for the sequence:

$$u_i = [1/n^i] \in \mathbb{S}.$$

Note that we need $n \geq 2$ in order to have $o(u_i) = n^i \to \infty$. The formal statement appears in `mathlib` as follows:

Listing 12 Formal statement of theorem 8

```
lemma add_circle.ergodic_nsmul {n : ℕ} (hn : 1 < n) :
  ergodic (λ (y : add_circle T), n · y) :=
```

In fact we needed the following mild generalisation of theorem 8:

▶ **Theorem 10.** *Given $n \in \mathbb{N}$ and $x \in \mathbb{S}$, the map:*

$$\begin{aligned} &\mathbb{S} \to \mathbb{S} \\ &y \mapsto ny + x \end{aligned}$$

is measure-preserving if $n \geq 1$ and is ergodic if $n \geq 2$.

This follows easily from theorem 8 because if we define the measure-preserving equivalence:

$$\begin{aligned} &e : \mathbb{S} \to \mathbb{S} \\ &\quad y \mapsto \frac{x}{n-1} + y \end{aligned}$$

then a quick calculation reveals:

$$e \circ g \circ e^{-1} = f,$$

where $f : y \mapsto ny$ and $g : y \mapsto ny + x$. As a result, theorem 10 follows from theorem 8 via:

Listing 13 The reduction of theorem 10 to theorem 8

```
lemma ergodic_conjugate_iff {e : α ≃ᵐ β} (h : measure_preserving e μ μ') :
  ergodic (e ∘ f ∘ e.symm) μ' ↔ ergodic f μ :=
```

6 Gallagher's theorem

6.1 Points of approximate order

Recall the definition of the set $W \subseteq \mathbb{S}$ appearing in the statement of theorem 2:

$$W = \limsup_{n>0} \mathrm{Th}(\delta_n, \{y \in \mathbb{S} \mid o(y) = n\}).$$

Key to the proof of theorem 2 is the way in which the sets $\mathrm{Th}(\delta_n, \{y \in \mathbb{S} \mid o(y) = n\})$ interact with the group structure of $\mathbb{S}$. We thus made the following definition:

▶ **Definition 11.** *Let A be a seminormed group, $n \in \mathbb{N}$ (non-zero), and $\delta \in \mathbb{R}$. We shall use the notation:*

$$\mathbb{AO}(A, n, \delta) = \mathrm{Th}(\delta, \{y \in A \mid o(y) = n\}),$$

for the set of points that have approximate order n, up to a distance δ.

For example, as shown in figure 2, $\mathbb{AO}(\mathbb{S}, n, \delta)$ is a union of $\varphi(n)$ arcs of diameter 2δ, centred on the points $[m/n]$ with $0 \leq m < n$ and m coprime to n (where φ is Euler's totient function).

The formal counterpart of definition 11 is:

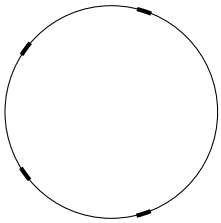


Figure 2 The set of points of approximate order 5 in $\mathbb{S}$, up to a distance $\delta \approx 0.01$.

Listing 14 Points of approximate order in a normed group

```
@[to_additive] def approx_order_of
  (A : Type*) [seminormed_group A] (n : ℕ) (δ : ℝ) : set A :=
thickening δ {y | order_of y = n}
```

Using this language, the only properties of $\mathbb{AO}(A, n, \delta)$ that we needed are as follows:

▶ **Lemma 12.** *Let A be a seminormed commutative group, $\delta \in \mathbb{R}$, $a \in A$, and $m, n \in \mathbb{N}$ (both non-zero). Then*[8]*:*

(i) $m \cdot \mathbb{AO}(A, n, \delta) \subseteq \mathbb{AO}(A, n, m\delta)$ *if m, n are coprime*,
(ii) $m \cdot \mathbb{AO}(A, nm, \delta) \subseteq \mathbb{AO}(A, n, m\delta)$,
(iii) $a + \mathbb{AO}(A, n, \delta) \subseteq \mathbb{AO}(A, o(a)n, \delta)$ *if $o(a)$ and n are coprime*,
(iv) $a + \mathbb{AO}(A, n, \delta) = \mathbb{AO}(A, n, \delta)$ *if $o(a)^2$ divides n*.

In fact property (iv) holds under the weaker assumption that $r(o(a))o(a)$ divides n where $r(l)$ denotes the radical of a natural number l, but we needed only the version stated in the lemma.

We made one last definition in support of theorem 2:

▶ **Definition 13.** *Let A be a seminormed group and $\delta_1, \delta_2, \ldots$ a sequence of real numbers. We shall use the notation:*

$$\mathbb{WA}(A, \delta) = \limsup_{n>0} \mathbb{AO}(A, n, \delta_n),$$

for the set of elements of A that are well-approximable by points of finite order, relative to δ.

Note that $W = \mathbb{WA}(\mathbb{S}, \delta)$ where W is the set appearing in the statement of theorem 2. The formal counterpart of definition 13 is:

Listing 15 The set of well-approximable elements of a normed group

```
@[to_additive] def well_approximable
  (A : Type*) [seminormed_group A] (δ : ℕ → ℝ) : set A :=
blimsup (λ n, approx_order_of A n (δ n)) at_top (λ n, 0 < n)
```

The additive version of this definition is `add_well_approximable`.

6.2 The main theorem

We are finally in a position to assemble everything and provide a proof of our main result. For the reader's convenience we reproduce the formal statement which appeared above in listing 1:

[8] If $s \subseteq A$ the notation $m \cdot s$ means $\{my \mid y \in s\}$.

Listing 16 Gallagher's theorem

```
theorem add_well_approximable_ae_empty_or_univ
  (δ : ℕ → ℝ) (hδ : tendsto δ at_top (𝓝 0)) :
  (∀ᵐ x, ¬ add_well_approximable 𝕊 δ x) ∨ ∀ᵐ x, add_well_approximable 𝕊 δ x :=
```

The notation $\forall^m x, \cdots$ should be read "for almost all $x \cdots$" and is standard `mathlib` notation. Using the lemmas `filter.eventually_eq_empty` and `filter.eventually_eq_univ` the statement in listing 16 is equivalent to:

▶ **Theorem 14** (Gallagher's theorem with $\delta \to 0$)**.** *Let $\delta_1, \delta_2, \ldots$ be a sequence of real numbers such that $\delta_n \to 0$ as $n \to \infty$. Then $\mathbb{WA}(\mathbb{S}, \delta)$ is almost equal to either $\emptyset$ or $\mathbb{S}$.*

Proof. For each prime $p \in \mathbb{N}$ we define three sets[9]:

$$A_p = \limsup_{n>0, p\nmid n} \mathbb{AO}(\mathbb{S}, n, \delta_n),$$

$$B_p = \limsup_{n>0, p\| n} \mathbb{AO}(\mathbb{S}, n, \delta_n),$$

$$C_p = \limsup_{n>0, p^2\mid n} \mathbb{AO}(\mathbb{S}, n, \delta_n).$$

Let $W = \mathbb{WA}(\mathbb{S}, \delta)$; bearing in mind (6) it is clear that for any p:

$$W = A_p \cup B_p \cup C_p. \tag{10}$$

We claim that these sets have the following properties:
(a) A_p is almost invariant under the ergodic map: $y \mapsto py$,
(b) B_p is almost invariant under the ergodic map: $y \mapsto py + [1/p]$,
(c) C_p is invariant under the map $y \mapsto y + [1/p]$.

To see why (a) holds, consider:

$$\begin{aligned} p \cdot A_p &= p \cdot \limsup_{n>0, p\nmid n} \mathbb{AO}(\mathbb{S}, n, \delta_n) \\ &\subseteq \limsup_{n>0, p\nmid n} p \cdot \mathbb{AO}(\mathbb{S}, n, \delta_n) \\ &\subseteq \limsup_{n>0, p\nmid n} \mathbb{AO}(\mathbb{S}, n, p\delta_n) && \text{by lemma 12 part (i)} \\ &=_{a.e.} A_p && \text{by lemma 5.} \end{aligned}$$

A very similar argument shows why (b) holds except using parts (ii), (iii) of lemma 12 instead of part (i).

Claim (c) is actually the most straightforward and holds by direct application of lemma 12 part (iv).

Now if A_p is not almost empty for any prime p, then because it is almost invariant under an ergodic map, lemma 7 tells us that it must be almost equal to $\mathbb{S}$. Since $A_p \subseteq W$, W must also almost equal $\mathbb{S}$ and we have nothing left to prove.

We may thus assume A_p is almost empty for all primes p. By an identical argument, we may also assume B_p is almost empty for all primes p. In view of (10), this means that:

$$W =_{a.e.} C_p \quad \text{for all } p.$$

Thus, by (c), W is almost invariant under the map $y \mapsto y + [1/p]$ for all primes p. The result then follows by applying lemma 9. ◀

[9] The notation $p\| n$ means that p divides n exactly once.

Omitting code comments, the formal version of this ~ 30 line informal proof in `mathlib` requires 101 lines.

7 Final words

7.1 Removing the $\delta_n \to 0$ hypothesis

As mentioned in the introduction, the hypothesis that $\delta_n \to 0$ in theorem 14 may be removed. A nice follow-up project would be to supply the proof in this case. By replacing δ_n with $\max(\delta_n, 0)$, we may assume $0 \leq \delta_n$ for all n. Given this, if $\delta_n \not\to 0$, then in fact:

$$\mathbb{WA}(\mathbb{S}, \delta) = \mathbb{S}.$$

Note that this is a true equality of sets; it is not a measure-theoretic result. The main effort would be to establish some classical bounds on the growth of the divisor-count and totient functions.

In fact Bloom and Mehta have already formalised some of the required bounds as part of their impressive Unit Fractions Project formalising Bloom's breakthrough [2, 3]. Once the relevant results are migrated to `mathlib`, removing the $\delta_n \to 0$ hypothesis will become even easier.

7.2 The Duffin-Schaeffer conjecture

Given some sequence of real numbers $\delta_1, \delta_2, \ldots$, Gallagher's theorem tells us that $\mathbb{WA}(\mathbb{S}, \delta)$ is almost equal to either $\emptyset$ or to $\mathbb{S}$. The obvious question is how to tell which of these two possibilities actually occurs for the sequence in hand. The Duffin-Schaeffer conjecture, now a theorem thanks to Koukoulopoulos and Maynard, provides a very satisfying answer:

$$\mathbb{WA}(\mathbb{S}, \delta) =_{a.e.} \begin{cases} \emptyset & \text{if } \sum \varphi(n)\delta_n < \infty, \\ \mathbb{S} & \text{if } \sum \varphi(n)\delta_n = \infty. \end{cases}$$

where φ is Euler's totient function.

That $\mathbb{WA}(\mathbb{S}, \delta) =_{a.e.} \emptyset$ if $\sum \varphi(n)\delta_n < \infty$ is very easy (it follows from the "easy" direction of the Borel-Cantelli theorem). The converse is extremely hard. It was first stated in 1941 [7] and was one of the most important open problems in metric number theory for almost 80 years.

A formal proof of the converse would be especially satisfying given how elementary the statement of the result is. After Gallagher's theorem, perhaps the next best target is lemma 5.2 in [12], i.e., theorem 2 in Pollington and Vaughan [13].

7.3 Aistleitner, Borda, and Hauke's quantitative results

The author is grateful to Christoph Aistleitner who contacted him after reading a preprint of this article. Aistleitner highlighted that Gallagher's theorem is required for Koukoulopoulos and Maynard's proof because the combination of Gallagher's theorem with the results of Pollington and Vaughan in [13] turns the Duffin-Schaeffer conjecture into a purely arithmetic problem. He also highlighted that he, Borda, and Hauke have been able to take Koukoulopoulos and Maynard's techniques further and obtain *quantitative* results even without using Gallagher's theorem, see [1].

7.4 Developing against `master`

It would have been impossible to complete the work discussed here without the extensive theories of algebra, measure theory, topology etc. contained within `mathlib`. As we have said, all of our code was added directly to the `master` branch of `mathlib`; most of it is "library code", not specific to Gallagher's theorem.

Although it is harder to develop this way, we believe it is essential in order to permit formalisation of contemporary mathematics. We therefore wish to exhibit this project as further evidence that this workflow can succeed, and we hope to encourage even more people to follow suit.

References

1 Christoph Aistleitner, Bence Borda, and Manuel Hauke. On the metric theory of approximations by reduced fractions: a quantitative Koukoulopoulos-Maynard theorem. *Compos. Math.*, 159(2):207–231, 2023. doi:10.1112/S0010437X22007837.

2 Thomas F. Bloom. On a density conjecture about unit fractions. *(to appear)*, 2021. arXiv:2112.03726.

3 Thomas F. Bloom and Bhavik Mehta. The Unit Fractions Project. *(to appear)*, 2022. URL: https://b-mehta.github.io/unit-fractions/.

4 Kevin Buzzard, Johan Commelin, and Patrick Massot. Formalising perfectoid spaces. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 299–312, 2020. doi:10.1145/3372885.3373830.

5 J. W. S. Cassels. Some metrical theorems in Diophantine approximation. I. *Proc. Cambridge Philos. Soc.*, 46:209–218, 1950. doi:10.1017/s0305004100025676.

6 Johan Commelin, Adam Topaz, et al. The Liquid Tensor Experiment. *(to appear)*, 2022. see also https://www.nature.com/articles/d41586-021-01627-2. URL: https://github.com/leanprover-community/lean-liquid.

7 R. J. Duffin and A. C. Schaeffer. Khintchine's problem in metric Diophantine approximation. *Duke Math. J.*, 8:243–255, 1941. URL: http://projecteuclid.org/euclid.dmj/1077492641.

8 Patrick Gallagher. Approximation by reduced fractions. *J. Math. Soc. Japan*, 13:342–345, 1961. doi:10.2969/jmsj/01340342.

9 Sébastien Gouëzel. Formalizing the Gromov-Hausdorff space. *CoRR*, abs/2108.13660, 2021. arXiv:2108.13660.

10 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.

11 Glyn Harman. *Metric number theory*, volume 18 of *London Mathematical Society Monographs. New Series*. The Clarendon Press, Oxford University Press, New York, 1998.

12 Dimitris Koukoulopoulos and James Maynard. On the Duffin-Schaeffer conjecture. *Ann. of Math. (2)*, 192(1):251–307, 2020. doi:10.4007/annals.2020.192.1.5.

13 A. D. Pollington and R. C. Vaughan. The k-dimensional Duffin and Schaeffer conjecture. *Mathematika*, 37(2):190–200, 1990. doi:10.1112/S0025579300012900.

14 The mathlib community. The Lean mathematical library. In *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, 2020. doi:10.1145/3372885.3373824.

15 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.

An Extensible User Interface for Lean 4

Wojciech Nawrocki
Carnegie Mellon University, Pittsburgh, PA, USA

Edward W. Ayers
Carnegie Mellon University, Pittsburgh, PA, USA

Gabriel Ebner
Microsoft Research, Redmond, WA, USA

Abstract

Contemporary proof assistants rely on complex automation and process libraries with millions of lines of code. At these scales, understanding the emergent interactions between components can be a serious challenge. One way of managing complexity, long established in informal practice, is through varying *external representations*. For instance, algebraic notation facilitates term-based reasoning whereas geometric diagrams invoke spatial intuition. Objects viewed one way become much simpler than when viewed differently. In contrast, modern general-purpose ITP systems usually only support limited, textual representations. Treating this as a problem of human-computer interaction, we aim to demonstrate that *presentations* – UI elements that store references to the objects they are displaying – are a fruitful way of thinking about ITP interface design. They allow us to make headway on two fronts – introspection of prover internals and support for diagrammatic reasoning. To this end we have built an extensible user interface for the Lean 4 prover with an associated `ProofWidgets 4` library of presentation-based UI components. We demonstrate the system with several examples including type information popups, structured traces, contextual suggestions, a display for algebraic reasoning, and visualizations of red-black trees. Our interface is already part of the core Lean distribution.

2012 ACM Subject Classification Human-centered computing → Visualization systems and tools; Software and its engineering → Functional languages

Keywords and phrases user interfaces, human-computer interaction, Lean

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.24

Supplementary Material *Software (User interface)*: `https://github.com/leanprover/vscode-lean4/tree/v0.0.102`, archived at `swh:1:rev:232b31446d71a697ef66cc3f9cdd671e52631317`
Software (ProofWidgets 4): `https://github.com/EdAyers/ProofWidgets4/tree/itp23` archived at `swh:1:dir:2c87d19df4c75dccfab1949cf370d3ca92a37be0`

Funding *Wojciech Nawrocki*: Hoskinson Center for Formal Mathematics.
Edward W. Ayers: Hoskinson Center for Formal Mathematics.

Acknowledgements The Lean team at MSR and KIT: Leonardo de Moura and Sebastian Ullrich for extensive discussions, code review, and improvements to the system, Daniel Selsam for suggesting traces, and Daniel Fabian for input on RPC design. The Penrose team: Wode Ni and Sam Estep for considerable help and implementing several features which made our use possible. Jeremy Avigad and Patrick Massot for suggestions, advice, and feedback on a draft of the paper. Tomáš Skřivan, Joachim Breitner, and Sina Hazratpour for trying our systems and suggesting improvements. Chris Lovett and Mariana Alanis for working on `vscode-lean4`. The Lean Zulip community for technical help and ideas.


© Wojciech Nawrocki, Edward W. Ayers, and Gabriel Ebner;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

Interactive theorem proving (ITP) distinguishes itself from other approaches to formal methods by structuring proof construction as a feedback loop between a human and a machine. Whether by filling typed holes in a partial term (Agda, Idris) or by issuing meta-level instructions in a tactic-based framework (HOL, Isabelle, Coq), users tend to develop proofs incrementally. At each step, the system displays the *goals* which remain to be proven and the user responds with a further refinement of their proof until there are no more goals left. This loop can be viewed as a dialogue between the user and the ITP system. Yet compared to human-to-human communication, modes of human-computer interaction available in today's general-purpose theorem provers are limited in *form* and in *referentiality*.

They are limited in **form** by being exclusively text-based. Text serves its purpose well: it is simple to process, supported in every system configuration, and universally understandable. Nevertheless, textual representations are only one way of displaying formal processes, statements, and their proofs. Cognitive science researchers have long suspected that *external representations* of concepts and objects outside the mind (for example a drawing on a piece of paper, or the physical disks in a Tower of Hanoi puzzle), complementing *internal* representations within the mind, are not merely an aid but rather an integral component of cognition [49]. Restricting the external representations available in ITP systems to only be text is thus a restriction on the way we think [45]. For instance, diagrammatic representations group related information together in ways that sequences of words simply cannot [29]. Since mathematicians and computer scientists rely on graphical calculi and processes such as diagram chases [20], computer mathematics should naturally support graphical representations.

Interactions are furthermore limited in **referentiality** in that we cannot refer to the objects that a displayed representation signifies by interacting with it directly. This is because the representations do not "remember" what they are representations *of*. Suppose for example that Alice and Bob are collaborating on a proof, using natural language and a blackboard. Suppose Bob attempts to commute x past y in the ring R but Alice notes that this cannot be done because R is not known to be commutative and one may not assume that $x \cdot y = y \cdot x$. At this point, Bob may respond by *referring* directly to R or to the term $x \cdot y$ and asking Alice for further facts about these objects in order to understand the issue and make progress on the proof. This illustrates that in dialogue, it is natural to request actions on an object under consideration by referring to it; dialogue is referential.

But replace Alice with an ITP system and suppose the corresponding message from Alice to Bob is that an instance of the `CommRing` typeclass couldn't be synthesized for the type `R`. To obtain detail on why this failed, the best Bob can generally do is copy-paste the offending type into a separate command, either to re-run the failing operation with more verbose output settings, or to print some extra information about it. Such interruptions are a source of friction which obstructs reasoning about the mathematical objects in question. Copy-pasting is only necessary because the displayed typeclass synthesis error is inert text which has "forgotten" details of the failure. The ITP feedback loop is thus not so much a dialogue as it is a sequence of disjoint request-response pairs. Had the system stored an association between the displayed error and input data involved in the failure instead, Bob would be able to inspect this data by interacting with the error message directly.

Failure of referentiality extends beyond the proof refinement loop, generally limiting the amount of information carried by messages originating in all components including parsing, type inference, proof search, decision procedures, and so on. Since in contemporary proof

assistants these components assemble into deep and interconnected stacks, understanding the behaviour of any single component (not to mention emergent phenomena arising from multiple components in combination) can be a serious challenge.

We will show that simply keeping better track of references can improve the state of things. Following Ciccarelli [17, 16], we call reference-preserving UI elements **presentations**. A presentation is a visual or textual display D of an object X with a link back from D to X. Thanks to the link, the *presented* object X can be acted upon in various ways by interacting with D. In our example, Bob could interact with a presentation of the typeclass inference error (by clicking on it or using another input device) in order to obtain more information about `R` or `CommRing`, to jump to their definitions, or to carry out other operations on them. Failure of referentiality can be restated as noting that some UI element is not a presentation.

1.1 Contributions

We report on the design and implementation of a user interface (UI) for the Lean 4 theorem prover [19], of an associated `ProofWidgets 4` library of UI components[1], as well as of supporting features in the metaprogramming framework and in the prover itself. Our system aims to enable more natural and efficient interactions with the prover by combining the following features:

- **Displays of arbitrary form.** We build on HTML5 and the web platform as the underlying technology to make visualization easier. Packages from the rich JavaScript ecosystem may be imported and used in the UI. For instance, in Section 3.1 the Penrose [47] library is used to visualize mathematical objects.
- **Referential presentations.** UI components keep track of, and may act on, the objects they signify. For example, expressions displayed in the UI can be hovered over to see their types and explicit forms (Section 2.1); and goal states can be interacted with in order to make progress on proofs (Section 3.3).
- **User-extensibility with reusable components.** The interface can be modified and extended by users, in Lean itself and in JavaScript. Builtin and user-defined components may be composed in arbitrary ways.
- **Live, interactive displays.** UI components can be used immediately, in the same Lean file they are defined in, with changes reflected in the UI in real-time.
- **On-demand computation**. Our presentations are *reactive* in that they compute lazily, in reaction to requests from the user. We can explore large objects such as computation traces (Section 2.1) by displaying only the relevant parts without processing the rest.
- **General-purpose design.** Like Lean itself, the UI and `ProofWidgets` are not tailored for any specific domain. They enable a variety of applications besides logical reasoning such as plotting, 3D visualisation, and interactive simulations.

While interfaces supporting subsets of the above have been developed, our system appears to be the first to support all of them in a cohesive way. We give a detailed comparison to other systems in Section 5. The UI is part of the core Lean distribution and has been deployed widely to hundreds of active users, whereas the `ProofWidgets` package can be imported for additional functionality. The UI has been integrated in the VS Code extension `vscode-lean4`[2] as well as in the `Lean 4 Web`[3] online editor.

[1] `https://github.com/EdAyers/ProofWidgets4/tree/itp23`
[2] `https://github.com/leanprover/vscode-lean4/`
[3] `https://lean.math.hhu.de/`

Outline. In Section 2, we introduce the user interface and its interactive features. In Section 3, we demonstrate how to extend the interface by means of several examples. In Section 4, internals of the system and aspects of implementation are discussed. We cover related work in Section 5 and conclude in Section 6.

2 The user interface

The layout of the Lean 4 user interface does not diverge from the two-pane view of the world popularized by ProofGeneral [4]. In this layout, the first pane in the prover UI is a text editor with the proof script, whereas the second **infoview** pane displays additional information. This includes the current goal state, errors, and messages for the open buffer. All the UI components and extensions which we will discuss are displayed within the infoview. An example infoview state is shown in Figure 1.

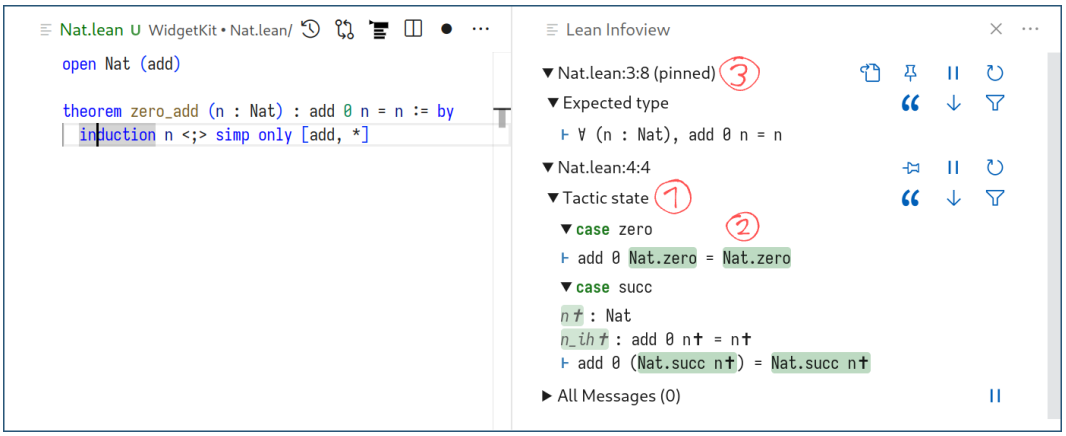


Figure 1 The Lean infoview embedded in `vscode-lean4`. Two tactic-mode goals (Tactic state) at the text cursor are shown (1). Differences in the goals' types and local contexts with respect to the previous state are highlighted (2). A second location containing a term goal (Expected type) is pinned (3).

While the layout is as in ProofGeneral, we do not follow its *waterfall* style of proof script management. In the waterfall style, there is a *checkpoint* to separate the part of the document which had been checked by the prover from that which had not. The checkpoint is advanced manually as an intentional action by the user. It recedes when changes are made to the checked part. Instead, similarly to Isabelle/PIDE [46], Lean adopts a "stateless" approach that checks the entire buffer in real-time. Under the hood, the system keeps track of immutable snapshots of past and present versions of the document, with new snapshots generated whenever the user edits the script. Contents of the infoview are determined by the latest snapshot and the current text cursor position.

When the cursor is inside a tactic-mode proof, the goal state at that position is displayed. In tactic proofs, differences between subsequent goal states are highlighted in green or red depending on whether a subexpression was just added or is about to be removed, respectively. This can be useful to see at a glance how a step has impacted, or will impact, the proof state. For instance when proving `∀ (n : Nat), 0 + n = n` by induction on `n`, in the base case `n` becomes `Nat.zero` and this change is highlighted as in Figure 1. The diff is computed using a heuristic algorithm operating on kernel-level expression trees. Furthermore when the cursor is over a typed hole (or a finished term), the *term goal* is also displayed. The term goal is the expected (or actual) type and local context of the typed hole (or term).

One advantage of the waterfall approach is that the checkpoint can be used as an additional cursor which displays the goal state in one part of the file while we go on to work on another part. We generalize this by allowing one or more text locations to be *pinned* in the infoview. Information about pinned locations is displayed alongside information about the text cursor location. Pinned displays update in real-time which is especially useful to see how changes at one point in the file affect a proof state or evaluation further down.

2.1 Expression and trace presentations

The infoview's design aims to support pervasive interactivity by displaying most objects as presentations. For instance, every displayed expression, and each of its subexpressions, stores a reference to the type-theoretic term it corresponds to. This can be used to learn additional facts about an expression appearing anywhere in the infoview (in a goal state or an error message or a custom component) by clicking on it or hovering over it as in Figure 2. Users can learn expressions' types, see the values inferred at implicit arguments, and jump to symbols' definitions. In this way presentations **increase information locality** by making it retrievable alongside a display of the relevant object. No extra data is computed eagerly; pretty-printing the type of every subexpression, for example, would not be cheap in any sizable goal state. Instead, the link from presentation to underlying object is a memory reference which enables the UI to fetch information from the language server lazily when the user requests it (see Section 4).

One way to frame the addition of presentations is as a kind of refinement process. We imagine starting from a non-referential user interface appearing in a particular scenario. We then ask:

- Which objects are signified by which parts of the UI?
- Given that UI D signifies object X, which actions applicable to X could we carry out using D?

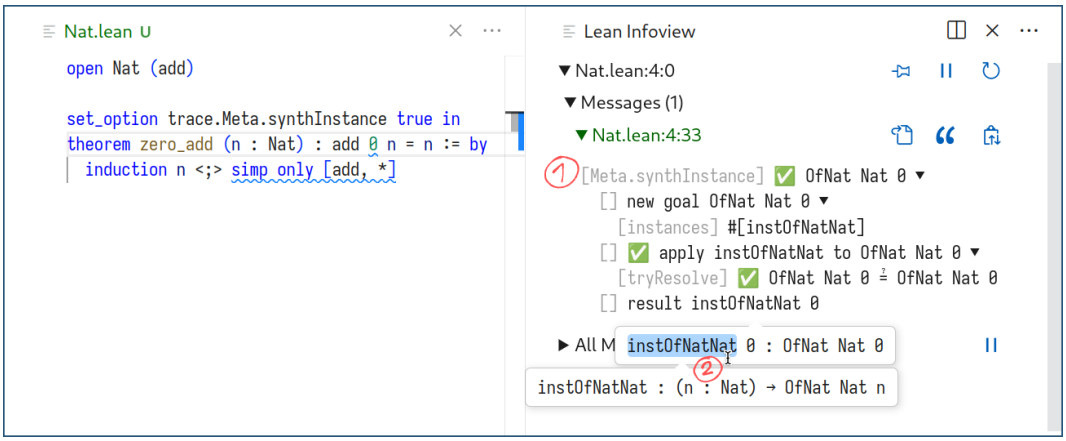


Figure 2 The numeral notation `0 : Nat` is resolved via typeclass search. A structured trace (1) of the search is explored. A presentation of a pretty-printed typeclass instance is clicked on to display its type (2). Subexpressions within an expression can be selected following its tree structure.

Guided by the answers, we can enrich interfaces for programming and proving with new interaction points. Consider messages produced by the prover: in Lean, *structured traces* are a feature of the metaprogramming API which collates messages produced during program execution into a tree-shaped record, with edges corresponding to user-defined execution

boundaries. For example, the backtracking Prolog-like typeclass search procedure [43] of Lean 4 can be traced, with branches representing attempted and abandoned instances. Many search-based tactics produce traces. Traces of expensive procedures can have thousands of nodes, making them unreadable and slow to pretty-print if displayed in full. Similarly to inferring expression types in the UI, we solve this problem by expanding and pretty-printing subtraces lazily, in reaction to user requests. This means we can explore branches through large trace trees limited only by the memory needed to store the trace data rather than the CPU time needed to pretty-print it all. In Figure 2, an example trace of typeclass instance search is shown. Presentations compose so that the structured trace may contain interactive expressions and other interactive components. In the future we hope to also provide a method of filtering and searching through the trace tree.

Presentations interact well with other language features including syntax extensions. In Figure 3, an embedded domain-specific language (EDSL) is used to write down an HTML tree. The tree has an underlying expression of type `Html` which is presented in the infoview using the same EDSL.

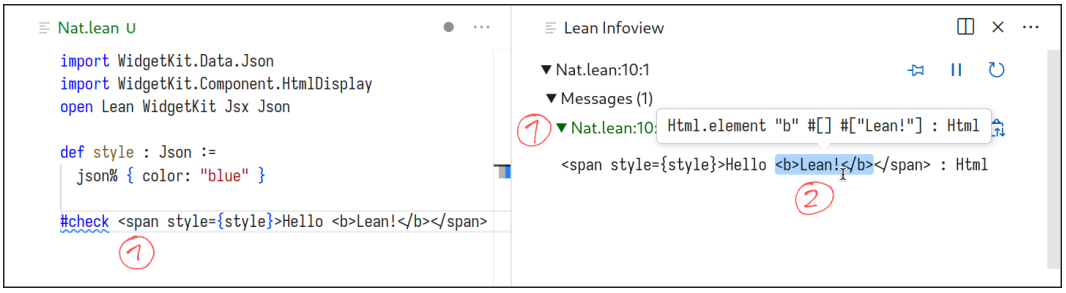


Figure 3 A JSX-like syntax for writing HTML trees inline is used to write down a term of type `Html` in the editor. The `#check` command is used to inspect it in the infoview (1). The type `Html` has an associated pretty-printer which emits the same custom syntax. The pretty-printer sub-output `<b>Lean!</b>` is a presentation of the subterm `Html.element "b" #[] #["Lean!"]` which can be inspected by hovering over it with the mouse (2).

Since presentations are the default, producing them requires no extra effort from the tactic writer. For example, the following snippet defines and then uses a custom command with interactive output. It does this by first using `elab`, a meta-level command that defines new commands with a given syntax, in this case `#check_nat t` where `t` can be any term. The new command is immediately available for use and is invoked with `37` as input.

```
import Lean.Elab.Command
open Lean Elab Command

elab cmd:"#check_nat " t:term : command => liftTermElabM do
  let e : Expr ← Term.elabTerm t (mkConst ``Nat)
  -- The string-like literal m!".." directly embeds expressions {..}.
  logInfoAt cmd m!"{e} has type {mkConst ``Nat}"

#check_nat 37
```

The implementation of `#check_nat` parses and typechecks the term, expecting its type to be `Nat`, and then emits a message. It does this using `logInfoAt` which associates a message with a syntactic span, in this case the span of the `#check_nat` keyword. Just like standard errors and warnings associated with a syntactic range, the message is displayed in the infoview whenever the text cursor is on this span. Since the message directly stores kernel-level expressions (of type `Expr`), they are automatically displayed as interactive presentations.

3 `ProofWidgets 4`: programmable, referential interfaces

While the builtin presentations of expressions, goals and messages provide a common interface for all uses, the design's main strength is its extensibility and composability. Users can build domain-specific interfaces dubbed **user widgets**. A user widget is a ReactJS UI component capable of invoking Lean metaprograms and editing the proof script. User widgets can implement new presentations and new ways of interacting with the prover. User widgets are usually displayed by related tactics or commands – for example the HTML display in Figure 5 is stored by the `#html` command. Storing a widget is analogous to how messages are emitted with `logInfoAt`: informally, instead of stating "there is an error or warning at this syntactic span", we state "there is a user widget at this syntactic span". Both the user interface and the associated tactic code can be developed in tandem alongside each other, allowing for quick development cycles.

In this section we will consider user widgets that extend the goal display in various ways. Here referentiality – the idea that displays should store references to objects they signify – is also core to our approach. Recall that the object displayed by an expression presentation (Section 2.1) is an expression together with its local context (approximately corresponding to a judgment $\Gamma \vdash t : T$ of the type theory). Executing with access to that allows us to, for example, infer its type and display it to the user. Similarly, widgets extending the goal display can reference the current goal state.

3.1 Diagrams for algebra

In Figure 4 the goal is an implication between statements in the language of category theory. We choose to display it as commutative diagrams connected by implication arrows. Here our support for importing JavaScript libraries shines – while it may seem like a trivial engineering choice, the ability to build on the immense NPM software ecosystem dramatically cuts down development time. One such library, Penrose [47], expresses general mathematical diagramming as an optimization problem. The user writes a specification describing which shapes the diagram should include (in `dsl` and `sub` files) as well as which constraints on their layout will make the diagram sound and beautiful (in a `sty` file). An energy minimization solver then runs and an SVG image is generated. The `ProofWidgets` component wrapping Penrose is composable in that it may include further components (in Figure 4 labels on objects and morphisms are interactive expression components) and dually may become part of a larger display. We hope it will prove useful to working algebraists. While the display demonstrated here does not act on the goal, proof methods such as diagram chases could also be implemented with `ProofWidgets`. We expand on this in Section 3.3.

From the user's perspective, implementing a display such as this one proceeds in two steps. First, we wrap Penrose into a reusable `ProofWidgets` component. The Lean definition of the `PenroseDiagram` component is as follows[4]:

```
structure PenroseDiagramProps where
  embeds : Array (String × EncodableHtml)
  dsl    : String
  sty    : String
  sub    : String
  deriving RpcEncoding
```

[4] Details are highly likely to change as the library evolves.

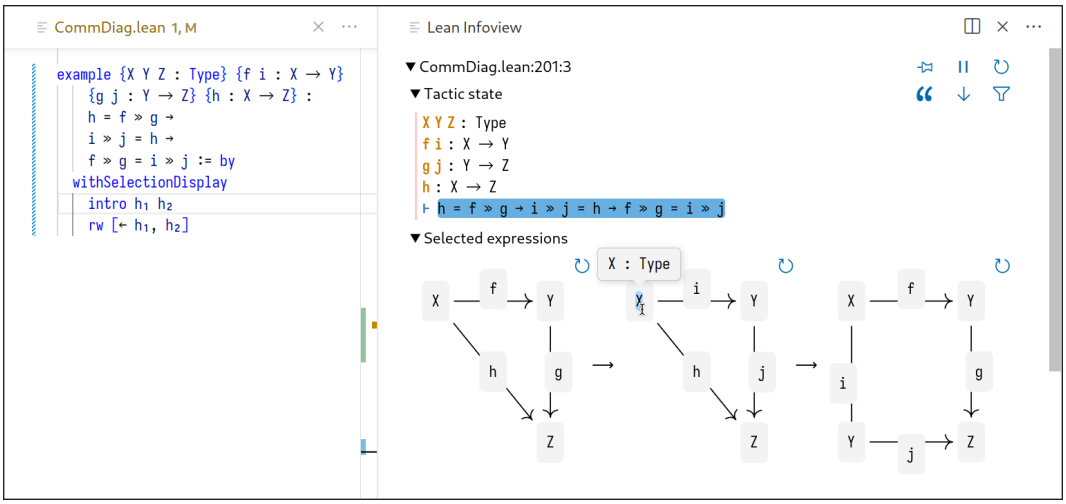


Figure 4 A target type in the language of category theory is selected. The statement is displayed as a sequence of commutative diagrams connected by implication arrows.

```
@[widget_module]
def PenroseDiagram : Component PenroseDiagramProps where
  javascript := ... -- Details omitted
```

Values of type `Component Props` serve to encapsulate JavaScript user widget implementations as Lean definitions. The index type `Props` specifies a Lean encoding of the type of data expected by the component. In this case `Props = PenroseDiagramProps` contains fields describing a specific diagram (`dsl`/`sty`/`sub`) as well as other widgets to nest within it (`embeds`). To give another example, one variant of the interactive expression component has type `Component ExprWithCtx` where `ExprWithCtx` is an expression together with its local context.

The field `javascript` contains a JavaScript implementation of the component. To a first approximation, it could be viewed as having dynamic type `Props → HTML`. It may be written inline but it is preferrable to point at a file on disk. In the latter case one may use tooling we have developed to integrate building TypeScript files into the build of a Lean package using the Lake (Lean Make) build system. Communication with the infoview is set up using the `@[widget_module]` attribute and the `deriving RpcEncoding` annotation. `@[widget_module]` saves the JavaScript code in a global storage from which it can be retrieved for execution in the infoview, whereas `deriving RpcEncoding` generates code to serialize and deserialize values of a type, in this case `PenroseDiagramProps`. This is necessary to support distributed computation (see Section 4).

More complex visualizations are enabled by building on further JavaScript libraries as in Figure 5. For example, a component integrating a plotting library could be a starting point for plotting functions in a formally verified way [34]. Finally, we note that this first step of wrapping JavaScript functionality in a `Component` can be skipped when the necessary UI component already exists. Thus it is desirable to write reusable components. For instance, `PenroseDiagram` is not specific to algebra but supports general constraint-based diagramming; we use it again in Figure 7.

In the second step, we write a Lean metaprogram to display the user widget. There are many ways to do this in general. Since Figure 4 uses an *`Expr` presenter*, we will describe this approach. Like most provers, Lean features an *elaborator* which translates surface-level (*vernacular*) syntax into fully explicit terms of the underlying type theory by filling in

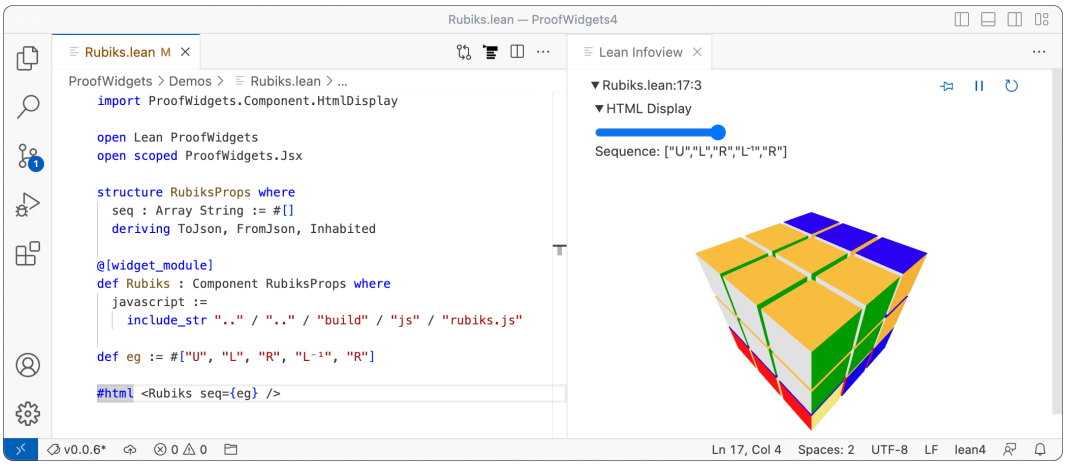


Figure 5 The `Rubiks` component loads the `three.js` library in order to create a 3D visualization of a Rubik's cube. An HTML tree `<Rubiks seq={eg} />` containing an instance of this component is passed to the `#html` command. This command can be used to render HTML trees in the infoview with a user widget (HTML Display). The sequence of rotations `eg` is determined by the Lean script.

implicit arguments, finding typeclass instances, resolving ambiguous notation, inserting coercions, and so on. Lean 4 also contains a *delaborator* which essentially does the inverse – it attempts to make an explicit term human-readable by heuristically removing detail while ensuring that the elaborator can still process the resulting vernacular. Eliding detail, the delaborator has type `Expr → MetaM Term` where `Expr` is the type of kernel terms, `Term` the type of abstract syntax trees corresponding to vernacular terms, and `MetaM` an appropriate monad. By composing with a pretty-printer for syntax trees we get the full pretty-printer of type `Expr → MetaM String`.

An **`Expr` presenter** is a `ProofWidgets` metaprogram which can be viewed as one generalization of the above process. Rather than producing strings, we output HTML trees which may include user widgets. As the name suggests, it is aimed at producing presentations of mathematical objects. The set of `Expr` presenters is user-extensible. We dispatch to the appropriate one based on characteristics of the given `Expr` such as using a known constant at the top level. This echoes the general design philosophy of Lean 4 as a tower of abstractions: some uses of `ProofWidgets` are expressed mostly simply by writing an `Expr` presenter, and for those that are not it is possible to drop to a lower level of abstraction.

To use this framework in our example, we wrote a Penrose specification for general commutative diagrams, as well as an `Expr` presenter that translates equalities of morphisms in a category into diagram descriptions which use that specification. A representative code fragment follows.

```
/-- Expressions to display as labels in a diagram. -/
abbrev ExprEmbeds := Array (String × Expr)

open scoped Jsx in
def mkCommDiag (sub : String) (embeds : ExprEmbeds) : MetaM EncodableHtml := do
  -- Pretty-print kernel terms into interactive labels for the diagram.
  let embeds ← embeds.mapM fun (s, h) =>
    return (s, EncodableHtml.ofHtml
      <InteractiveCode fmt={← Widget.ppExprTagged h} />)
  return EncodableHtml.ofHtml
```

```
    -- Instantiate a PenroseDiagram using a JSX-like EDSL.
    <PenroseDiagram
      embeds={embeds}
      -- Penrose specification of general commutative diagrams.
      dsl={include_str "commutative.dsl"}
      sty={include_str "commutativeOpt.sty"}
      -- The particular diagram we are given.
      sub={sub} />

... -- Definitions of commSquareM? and commTriangleM? elided

/-- Present an expression as a commutative diagram. -/
@[expr_presenter]
def commutativeDiagramPresenter : ExprPresenter where
  userName := "Commutative diagram"
  present type := do
    -- Attempt to deconstruct `type` into a commutative square or triangle
    -- and use `mkCommDiag` if successful.
    if let some d ← commSquareM? type then
      return some d
    if let some d ← commTriangleM? type then
      return some d
    return none
```

3.2 Selection contexts

On a blackboard, we can underline and point to expressions and objects in order to highlight the relevant parts of a formula or depiction when explaining an argument. Analogously, a **selection context** is a subset of (subexpressions of) goals, hypotheses, and (subexpressions of) hypothesis types appearing in a goal state. The user specifies it by shift-clicking on the respective elements in the infoview. The current selection context is passed as input to user widgets that pertain to the goal. In Figure 4 just the target type was selected. The `withSelectionDisplay` combinator, which we use there and in Figure 6, is a tactic combinator that associates a general-purpose widget with the range of the entire nested tactic script, and then runs the script unchanged. The widget displays each selected expression using registered `Expr` presenters (if multiple presenters apply, a choice can be made in the UI).

Selecting more than one subexpression can be helpful in comparing differences between these subexpressions, to figure out what remains to be proven. In Figure 6, we copied a balancing function for red-black trees verbatim from Okasaki [37]. As it turns out, due to overlapping patterns in the definition of `balance`, the reduction law one might expect does not hold in all cases. It does hold when `balance` is called after inserting one node into a well-formed red-black tree because in that case, the invariants ensure that no more than one red-red edge exists. In Figure 6, we can see at a glance from their visual representations that the two selected trees cannot be equal, so an invariant must have been violated. In this way diagrams appearing live during proof development serve as cognitive aids. The visualization of general red-black trees uses the `react-d3-tree`[5] library to do most of the heavy lifting and took less than an hour to prototype. Afterwards, figures from Okasaki's paper are reproduced by the system with no further effort.

[5] `https://github.com/bkrem/react-d3-tree`

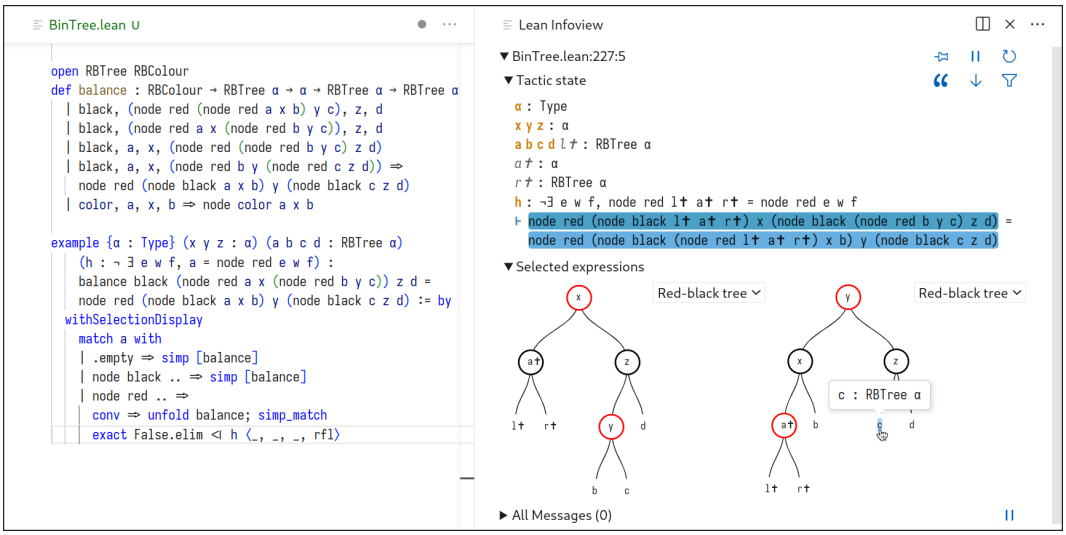


Figure 6 A balancing function for red-black trees is implemented in `balance`. Two terms appearing in the course of a proof about it are selected in the goal and illustrated as trees.

Finally, rather than using `withSelectionDisplay` which treats elements of the selection context as independent, users may choose to visualize the selection context as one entity. This is useful when the global information contained therein can be coherently diagrammed. In Figure 7, two subset relations are relevant to the proof whereas a third one is not. We use `PenroseDiagram` together with Penrose's builtin support for Venn diagrams to display the two relations which imply the conclusion. The combinator `withVennDisplay` used here works similarly to `withSelectionDisplay` except in that, rather than emitting the general-purpose selection display widget, it produces an instance of a Venn diagram specifically.

3.3 Contextual suggestions and graphical calculi

Beyond providing static displays of goal states guided by the selection context, user widgets may invoke Lean metaprograms, access proof states, and edit the proof script. Since metaprograms can also display user widgets, the link between widgets and metaprogramming is bidirectional. It is possible to make progress on proofs through the UI.

One application of this functionality could be *proof by pointing* [10, 11] which, to a first approximation, demands that the UI should allow guiding proof synthesis by pointing (with a mouse, for example) at the term to use, decompose, or otherwise manipulate in the next proof step. Since the selection context already contains terms which the user pointed out, a proof by pointing widget would only need to respond to clicks by inserting appropriate tactics into the proof script. On the other hand Paulson argues [38] that certain specific variations of this idea, such as guiding term rewriting by hand, are better served by powerful automation. Ultimately some combination of both appears most likely to be useful. For example, a piece of Sledgehammer-like automation [12], or a system based on recent advances in deep learning [28], could suggest proof steps that make progress on the proof in a manner related to the current selection context. `ProofWidgets` avoids committing to any single approach by remaining agnostic about which actions or graphical proof methods are available, instead leaving the choice to users and their particular applications. What we hope to achieve is to make the implementation of *any* such method as frictionless as possible by providing a library of basic components. We envision it being used for *contextual suggestions* and *graphical calculi*.

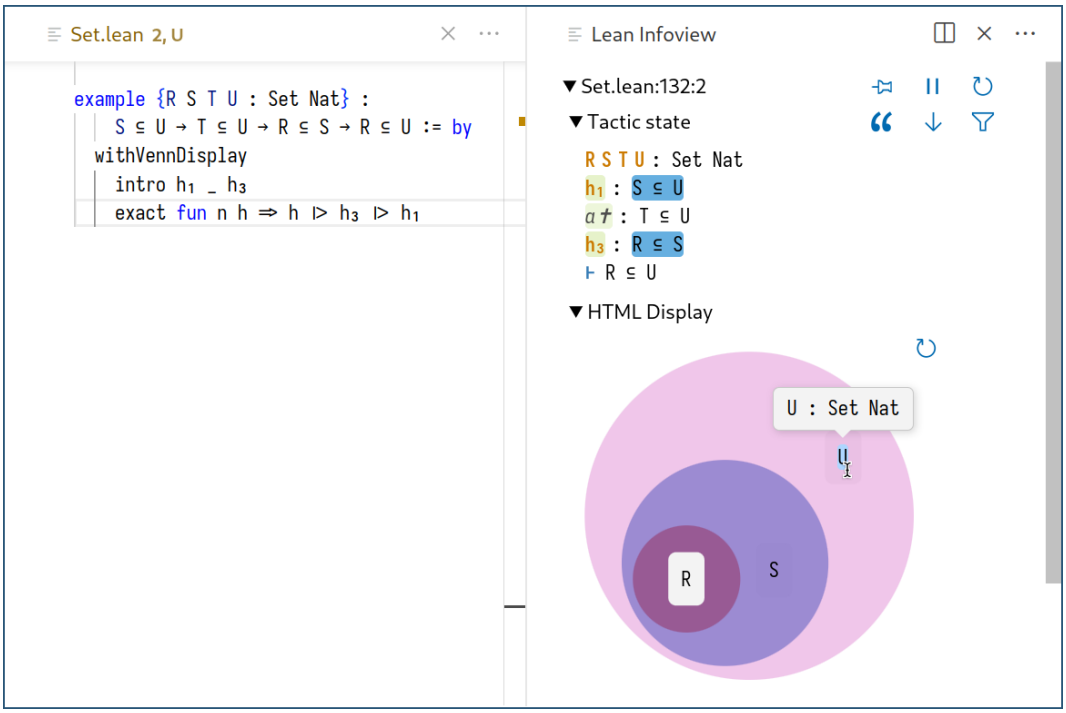


Figure 7 A subset of hypotheses relevant to the proof is selected. The set relationships are visualized in one Venn diagram.

Contextual suggestions are provided by *suggestion providers*. These are metaprograms which, given a goal state and selection context, return a list of relevant or potentially useful tactics that the user may then pick from. For example, proof by pointing implementations could be viewed as suggestion providers which suggest tactics to carry out the desired goal transformation. Like the set of `Expr` presenters, the set of suggestion providers is user-extensible rather than fixed. In Figure 8 we demonstrate how a user widget presenting a suggestion can operate. In Lean, the `conv` tactic mode allows "zooming in" on a subexpression of the target or a hypothesis type in order to apply local transformations. In the figure, a suggestion provider returns a `conv` tactic which would put the selected subexpression in focus. The tactic is then displayed in the infoview and may be inserted by clicking the button.

As we observed in Section 1, diagrams serve as cognitive aids in a variety of mathematical pursuits. **Graphical calculi** are distinguished from general depictions by being *active*, meaning that manipulations of the depiction correspond to steps in a proof; *sound*, meaning that valid manipulations are valid proof steps; and ideally *complete*, meaning that every proof in a chosen class can be expressed graphically. Examples include the Reidemeister moves on knot diagrams [40], manipulations of string diagrams [26], or more specific variants in category theory such as ZX-diagrams [18], Globular proofs [6], and `homotopy.io` [41] proofs. A formalization of any of these graphical languages could be accompanied by a `ProofWidgets` component which translates manipulations of a graphical proof state in the infoview into tactic steps in the Lean proof script.

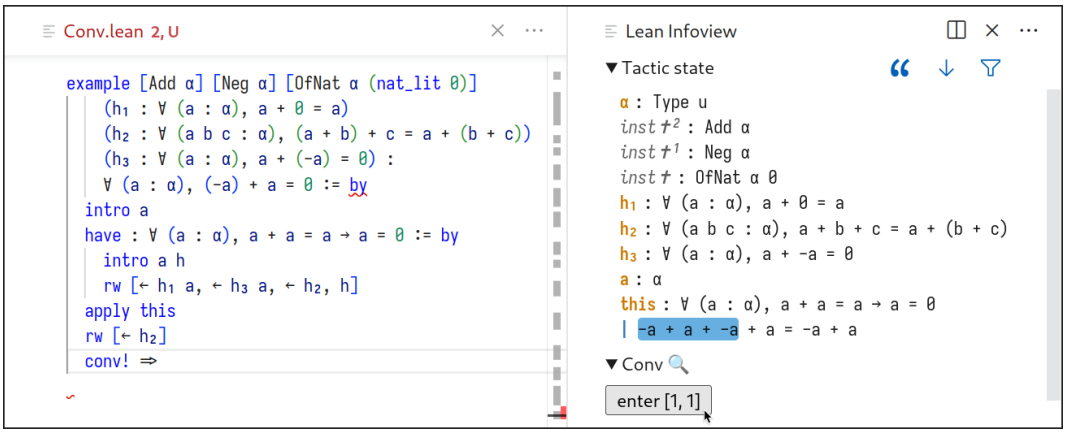


Figure 8 A subterm `-a + a + -a` of the goal in a proof about groups [21] is selected. The `conv` user widget by Robin Böhne and Jakob von Raumer displays a button suggesting a tactic which would zoom in on the selected subterm. Clicking the button inserts the tactic into the proof script.

4 Implementation

A complete setup consists of three components. Figure 9 outlines an example interaction between them.

The language server is written in Lean. It communicates with the editor and with the infoview via the Language Server Protocol (LSP[6]). Through the LSP it provides standard code intelligence facilities – go-to-definition, type hovers, autocompletion, etc. Proof states and related objects such as terms of the type theory are stored in the server.

The infoview is written in TypeScript. It is a self-contained web application displayed by the editor. Client-side JavaScript code from user widgets executes here.

The text editor is chosen by the user. Besides storing the proof script, the editor connects to the server, manages the infoview, and mediates between them. To support both, the editor must be capable of communicating via LSP and displaying web content. For example, the Visual Studio Code extension `vscode-lean4` embeds the infoview in a webview pane which it has control over.

Remote procedure calls. The infoview and the server are independent programs which may not even execute on the same computer. Indeed, this happens when Lean is used over SSH or in a cloud-based service such as Gitpod. In these cases, the editor and infoview execute on the user's local machine whereas the server is remote. Certain objects stored in the server's memory should not be serialized and sent to the infoview over the network due to their size – for instance, the environment (which stores known theorems, definitions, metadata, etc) can weigh several gigabytes in sufficiently large proof developments. Consequently, metaprograms which operate on heavy objects must execute in the server.

Nevertheless, user widgets need to run such metaprograms, for example to try a tactic or infer an expression's type: both of these need access to the environment. Therefore widgets must be able to invoke methods on the server. To enable this, we designed a foreign

[6] `https://microsoft.github.io/language-server-protocol/`

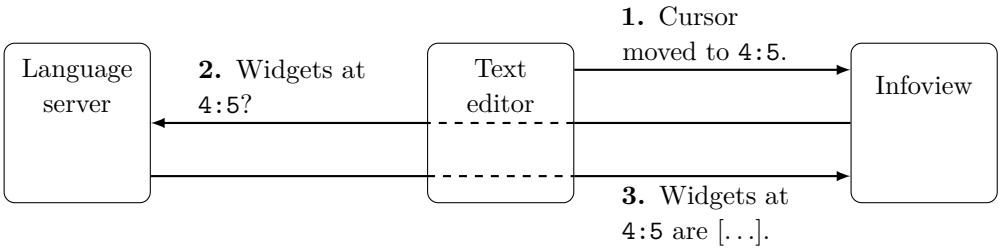


Figure 9 In order to determine which user widgets to show in the infoview, a sequence of messages is exchanged every time the text cursor moves. First, the editor informs the infoview about the new cursor position (here line `4`, column `5`). Then, the infoview queries the language server for the user widgets that should be shown at that position. Finally, the server replies with a list of user widgets. Its contents are then displayed in the infoview. The editor acts as a proxy for infoview–server communication, indicated by dashed lines.

function interface for Lean with support for remote calls from JavaScript. The interface is effectively an extension to LSP. The LSP is based on JSON-RPC[7], a simple protocol for remote procedure calls which encodes argument and output data as JSON. For example, to request a symbol's definition, the editor invokes the `textDocument/definition` method by sending a JSON record of the file, line, and column where an instance of the symbol occurs. The return value sent back by the server is the definition's location. To support arbitrary other functionality, we made the registry of procedures that can be invoked on the server via JSON-RPC user-extensible. To mark a Lean procedure as remotely callable, one annotates it with `@[server_rpc_method]`. A procedure so marked must be of the type `A → RequestM B` where `RequestM` is a monad with access to server state and `A,B` are JSON-serializable types. (De)serialization routines are autogenerated by annotating a type definition with `deriving ToJson, FromJson` or `deriving RpcEncoding`.

Remote references. When making multiple remote calls, widgets need to pass data between the metaprograms they invoke, for example to compute an expression's explicit form and then infer the type of a subexpression of that (as do the two popups in Figure 2). Since the relevant data is not serialized, client-side code needs a way of referencing objects stored in the server's memory. This is achieved by allowing JSON-RPC payloads to contain opaque references to server-side objects. A value of any type may be referenced opaquely by being marked with the `WithRpcRef` type-level function. To ensure type correctness, runtime type information is stored and checked on any remote reference access.

Remote references are the backbone of our implementation of presentations. One use is found in expression presentations. Recall from Section 3.1 that Lean features a *delaborator*, a system for converting kernel-level expressions back into syntax trees. To implement expression presentations, the delaborator has been extended with the ability to tag syntax subtrees with references to subexpressions of the original expression. These references are encoded using `WithRpcRef`. The exact tagging strategy has been described by Ayers and coauthors [5].

Allowing the client to refer to server-side objects presents us with a classic memory management problem – when is it safe for the server to delete objects for which remote references have been created? Conveniently, both Lean and JavaScript are garbage-collected

[7] `https://www.jsonrpc.org/`

languages. Using a `FinalizationRegistry`[8] we can instruct the JavaScript garbage collector to send a memory release instruction to the server when it collects the corresponding client-side reference. This is cooperative and may fail in case of client-side errors: a client which does not release server-side memory could cause it to leak. While we can't prevent this in general using only server-side mechanisms, we require the client to regularly send `keepalive` messages inspired by the Transmission Control Protocol [13]. Upon not seeing any `keepalive`s for sufficiently long, the server frees all remote references. This eliminates a class of disconnection- and hang-related memory leaks.

Dynamic code delivery. User widgets are required to be self-contained JavaScript modules[9]. This is considered a low-level target – users may employ any libraries and toolchains they need (for example TypeScript), as long as the eventual compilation or transpilation output matches the required format. Modules are registered in Lean using the `@[widget_module]` annotation. Upon being so annotated, modules are stored in a content-addressed cache accessible to the server. In order to display a particular user widget, the infoview fetches its source module from the cache using a remote procedure call, and then dynamically loads this source. User widgets execute in a runtime environment including the `@leanprover/infoview` library which they may import. This library exposes builtin functionality of the infoview (it can be used to display expression, structured trace, or goal presentations) as well as methods of communicating with the editor (these can be used for instance to edit the proof script or place another Lean file in focus) and services for making remote procedure calls from user widgets.

5 Related work

Our work descends directly from graphical tooling for Lean 3 (the previous version of Lean), notably `ProofWidgets 3` [5] and the previous infoview. `ProofWidgets 4` is a complete redesign and reengineering. Compared to `ProofWidgets 4`, the previous version was not able to incorporate JavaScript libraries which we make heavy use of; used purely server-side rendering which resulted in disruptive latency approaching seconds [35] in distributed settings where code editor and prover reside on different machines (e.g. cloud-based services such as Gitpod); and could not handle asynchronous events which are necessary to invoke long-running computations from the UI. Compared to the previous version we have lost (and hope to regain) the ability to program UI event handlers directly in Lean rather than in JavaScript. We expect this to become feasible when a JavaScript or WebAssembly backend is developed for Lean. The previous infoview did not support goal diffs, structured traces, interactive messages, or selection contexts.

The work of Mehnert, Christiansen, Korkut, and coauthors on `idris-mode` [33, 16, 27] encouraged us to dream of richer programming environments, and suggested presentations as a useful concept. `idris-mode` is primarily limited by the practical difficulty of embedding web-based and non-textual interfaces in Emacs with which it is tightly integrated.

User interfaces for theorem provers can be broadly categorized along two axes. Along one, they can either be built for a specific domain and use case only, or they can be *tool-making tools* designed for extensibility. Along another axis, they can either be integrated with a

[8] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/FinalizationRegistry`
[9] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules`

special-purpose formal system such as a synthetic axiomatization of geometry, or they can support a general foundation. Our interface is designed from the ground up to fall on the right of both axes, that is to support general interface extensions in a general-purpose theorem prover. Existing work tends to place towards the left of at least one of the axes, with many tools excelling at providing fixed sets of UI functionality.

CtCoq [8] and its successor Pcoq [2] were early systems which focused on displaying formulas and proofs in natural language with mathematical notation, on structured editing of proofs, and on proof by pointing. Some extensions to Pcoq have been developed by its authors, notably GeoView [9], a display for statements in plane geometry. Nevertheless Pcoq does not appear to support general user-extensibility. The GeoProof [36] project improved on GeoView by supporting proof construction, rather than just viewing, in the geometric display. However, GeoProof was developed as a standalone application that did not use Pcoq. Robert's PeaCoq [42] focuses on visualizing proof trees and steps, but not the mathematical objects appearing therein such as the diagrams of Figure 4. The recent Actema project [22] aims to extend the interactions available in proof by pointing to drag-and-drop interfaces. KeY [1] and KeYmaera X [24] provide interfaces specific to software verification and purpose-built logics. The *Incredible Proof Machine* [14] is a browser-based diagrammatic prover. We hope that our framework enables the creation of similar purpose-specific tooling for the Lean proof assistant.

A recent interface which *does* aim at general-purpose proving and domain-specific extensions is that of HolPy [48]. Compared to `ProofWidgets 4`, at this moment HolPy stresses proof by pointing and LaTeX display but not general visualization of objects or computations.

Another class of interfaces and tools are web-based ones including `jsCoq` [3] and Clide [32]. The comparison here is subtler – while `jsCoq` in particular allows building websites intermixing Coq snippets and UI components, it doesn't seem to provide a way for these components to invoke the metaprogramming API and directly manipulate proof state. It may be that the potential for powerful extensions is there, but was simply never realized in practice. The recent `Alectryon` [39] supports proof *archival* in Coq and in Lean (via `LeanInk` [15]) by storing recorded proof states alongside beautified proof scripts. In contrast, our system serves proof *development* by providing a live display with a variety of graphical representations. We would, however, like to store a static form of these representations in `LeanInk` outputs in the future.

Other systems intersect with our featureset in various ways. ProofGeneral [4] used to support expression presentations, but only for the LEGO prover [31]. Feasibility of real-time asynchronous processing was demonstrated in Isabelle/PIDE [46]. Both PeaCoq and Coq itself contain similar goal diffing capabilities to ours. Multi-representation GUIs for proof assistants were pioneered in the 1990s by the *LΩUI* [44], HyperProof [7] and XBarnacle [30] projects.

Finally, we are generally inspired by Engelbart's (to-date not realized!) vision of human intelligence augmented through computer interfaces [23], and the systems of yore which followed it including Smalltalk [25].

6 Conclusion

We designed and implemented an extensible user interface for the Lean 4 theorem prover together with `ProofWidgets 4`, a supporting library of metaprograms and UI components. The interface is based on presentations: UI elements that store references to the objects they are displaying. Presentations enable detailed introspection of tactics and systems comprising

the prover. Extending the interface with `ProofWidgets 4` empowers users to work with a variety of interactive, graphical representations. Building on the JavaScript ecosystem enables quick prototyping. The framework's domain of applicability includes exploring computation traces, symbolic visualization and exploration of mathematical objects and data structures, custom interfaces for tactics and tactic modes, data visualization, function plotting, and interactive simulations. Supporting not only expert users, it could be used in education to build interactive textbooks and tutorials. We demonstrated example user widgets diagramming mathematical data and suggesting possible proof steps from within the UI.

In tune with the overall design philosophy of Lean 4, every layer of the visual stack can be extended. We provide a *tool-making tool* which enables the creation of rich environments for program and proof in science and mathematics.

References

1 Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich. *Deductive Software Verification - The KeY Book*. Lecture Notes in Computer Science. Springer, 2016. doi:10.1007/978-3-319-49812-6.

2 Ahmed Amerkad, Yves Bertot, Loïc Pottier, and Laurence Rideau. Mathematics and Proof Presentation in Pcoq. Technical Report RR-4313, INRIA, November 2001. URL: https://hal.inria.fr/inria-00072274.

3 Emilio Jesús Gallego Arias, Benoît Pin, and Pierre Jouvelot. jscoq: Towards hybrid theorem proving interfaces. In Serge Autexier and Pedro Quaresma, editors, *Proceedings of the 12th Workshop on User Interfaces for Theorem Provers, UITP 2016, Coimbra, Portugal, 2nd July 2016*, volume 239 of *EPTCS*, pages 15–27, 2016. doi:10.4204/EPTCS.239.2.

4 David Aspinall. Proof general: A generic tool for proof development. In Susanne Graf and Michael I. Schwartzbach, editors, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, pages 38–43. Springer, Springer, 2000. doi:10.1007/3-540-46419-0_3.

5 Edward W. Ayers, Mateja Jamnik, and William T. Gowers. A graphical user interface framework for formal verification. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 4:1–4:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ITP.2021.4.

6 Krzysztof Bar, Aleks Kissinger, and Jamie Vicary. Globular: an online proof assistant for higher-dimensional rewriting. In *Leibniz International Proceedings in Informatics*, volume 52, pages 34:1–34:11, 2016. ncatlab.org/nlab/show/Globular.

7 Jon Barwise and John Etchemendy. Hyperproof: Logical reasoning with diagrams. In *Working Notes of the AAAI Spring Symposium on Reasoning with Diagrammatic Representations*, 1992. URL: https://www.aaai.org/Papers/Symposia/Spring/1992/SS-92-02/SS92-02-016.pdf.

8 Yves Bertot. The ctcoq system: Design and architecture. *Formal Aspects Comput.*, 11(3):225–243, 1999. doi:10.1007/s001650050049.

9 Yves Bertot, Frédérique Guilhot, and Loic Pottier. Visualizing geometrical statements with geoview. In David Aspinall and Christoph Lüth, editors, *Proceedings of the User Interfaces for Theorem Provers Workshop, UITP@TPHOLs 2003, Rome, Italy, September 8, 2003*, volume 103 of *Electronic Notes in Theoretical Computer Science*, pages 49–65. Elsevier, 2003. doi:10.1016/j.entcs.2004.09.013.

10 Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software, International Conference TACS '94, Sendai, Japan, April 19-22, 1994, Proceedings*, volume 789 of *Lecture Notes in Computer Science*, pages 141–160. Springer, 1994. doi:10.1007/3-540-57887-0_94.

11 Yves Bertot and Laurent Théry. A generic approach to building user interfaces for theorem provers. *J. Symb. Comput.*, 25(2):161–194, 1998. doi:10.1006/jsco.1997.0171.

12 Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending sledgehammer with SMT solvers. *J. Autom. Reason.*, 51(1):109–128, 2013. doi:10.1007/s10817-013-9278-5.

13 R. Braden. Requirements for internet hosts - communication layers. RFC 1122, RFC Editor, October 1989. URL: https://www.rfc-editor.org/rfc/rfc1122.txt.

14 Joachim Breitner. Visual theorem proving with the incredible proof machine. In Jasmin Christian Blanchette and Stephan Merz, editors, *International Conference on Interactive Theorem Proving*, pages 123–139. Springer, 2016. doi:10.1007/978-3-319-43144-4_8.

15 Niklas Bülow. Proof visualization for the lean 4 theorem prover, April 2022.

16 David Christiansen, David Darais, and Weixi Ma. The final pretty printer, 2016. URL: https://web.archive.org/web/20230219222209/https://davidchristiansen.dk/drafts/final-pretty-printer-draft.pdf.

17 Eugene Charles Ciccarelli. *Presentation based user interfaces.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1984. URL: https://hdl.handle.net/1721.1/15346.

18 Bob Coecke and Ross Duncan. Interacting quantum observables. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, volume 5126 of *Lecture Notes in Computer Science*, pages 298–310. Springer, 2008. doi:10.1007/978-3-540-70583-3_25.

19 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.

20 Silvia de Toffoli. Chasing the diagram–the use of visualizations in algebraic reasoning. *Review of Symbolic Logic*, 10(1):158–186, 2017. doi:10.1017/s1755020316000277.

21 R.A. Dean. *Elements of Abstract Algebra.* Wiley international edition. Wiley, 1966. URL: https://books.google.com/books?id=kmulxmBgkxoC.

22 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.

23 Douglas C. Engelbart. Augmenting human intellect: A conceptual framework. Technical report, Stanford Research Institute, October 1962. URL: https://web.archive.org/web/20230220110343/https://dougengelbart.org/pubs/augment-3906.html.

24 Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In Amy P. Felty and Aart Middeldorp, editors, *CADE*, volume 9195 of *LNCS*, pages 527–538. Springer, 2015. doi:10.1007/978-3-319-21401-6_36.

25 Adele Goldberg. *Smalltalk-80 - the interactive programming environment.* Addison-Wesley, 1984.

26 André Joyal and Ross Street. The geometry of tensor calculus, i. *Advances in mathematics*, 88(1):55–112, 1991.

27 Joomy Korkut and David Thrane Christiansen. Extensible type-directed editing. In Richard A. Eisenberg and Niki Vazou, editors, *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2018, St. Louis, MO, USA, September 27, 2018*, pages 38–50. ACM, 2018. doi:10.1145/3240719.3241791.

28 Guillaume Lample, Marie-Anne Lachaux, Thibaut Lavril, Xavier Martinet, Amaury Hayat, Gabriel Ebner, Aurélien Rodriguez, and Timothée Lacroix. Hypertree proof search for neural theorem proving. *CoRR*, abs/2205.11491, 2022. doi:10.48550/arXiv.2205.11491.

29 Jill H. Larkin and Herbert A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11(1):65–100, 1987. doi:10.1111/j.1551-6708.1987.tb00863.x.

30 Helen Lowe and David Duncan. Xbarnacle: Making theorem provers more accessible. In William McCune, editor, *Automated Deduction - CADE-14, 14th International Conference on Automated Deduction, Townsville, North Queensland, Australia, July 13-17, 1997, Proceedings*, volume 1249 of *Lecture Notes in Computer Science*, pages 404–407. Springer, 1997. doi:10.1007/3-540-63104-6_39.

31 Zhaohui Luo and Robert Pollack. Lego proof development system: User's manual. Technical report, LFCS, Edinburgh University, 1992. URL: https://www.lfcs.inf.ed.ac.uk/reports/92/ECS-LFCS-92-211/.

32 Christoph Lüth and Martin Ring. A web interface for isabelle: The next generation. In Jacques Carette, David Aspinall, Christoph Lange, Petr Sojka, and Wolfgang Windsteiger, editors, *Intelligent Computer Mathematics - MKM, Calculemus, DML, and Systems and Projects 2013, Held as Part of CICM 2013, Bath, UK, July 8-12, 2013. Proceedings*, volume 7961 of *Lecture Notes in Computer Science*, pages 326–329. Springer, 2013. doi:10.1007/978-3-642-39320-4_22.

33 Hannes Mehnert and David Christiansen. Tool demonstration: An ide for programming and proving in idris, 2014. URL: https://davidchristiansen.dk/pubs/dtp2014-idris-mode.pdf.

34 Guillaume Melquiond. Plotting in a formally verified way. In José Proença and Andrei Paskevich, editors, *Proceedings of the 6th Workshop on Formal Integrated Development Environment, F-IDE@NFM 2021, Held online, 24-25th May 2021*, volume 338 of *EPTCS*, pages 39–45, 2021. doi:10.4204/EPTCS.338.6.

35 Robert B Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 267–277, 1968.

36 Julien Narboux. A graphical user interface for formal proofs in geometry. *J. Autom. Reason.*, 39(2):161–180, 2007. doi:10.1007/s10817-007-9071-4.

37 Chris Okasaki. Red-black trees in a functional setting. *J. Funct. Program.*, 9(4):471–477, 1999. doi:10.1017/s0956796899003494.

38 Lawrence C. Paulson. Thoughts on user interfaces for theorem provers, December 2022. URL: https://web.archive.org/web/20230219221749/https://lawrencecpaulson.github.io/2022/12/14/User_interfaces.html.

39 Clément Pit-Claudel. Untangling mechanized proofs. In Ralf Lämmel, Laurence Tratt, and Juan de Lara, editors, *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020*, pages 155–174. ACM, 2020. doi:10.1145/3426425.3426940.

40 Kurt Reidemeister. *Knot theory*. BCS Associates, 1983.

41 David Reutter and Jamie Vicary. High-level methods for homotopy construction in associative n-categories. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '19. IEEE Press, 2021.

42 Valentin Robert. *Front-end tooling for building and maintaining dependently-typed functional programs*. PhD thesis, University of California, San Diego, USA, 2018. URL: http://www.escholarship.org/uc/item/9q3490fh.

43 Daniel Selsam, Sebastian Ullrich, and Leonardo de Moura. Tabled Typeclass Resolution. *CoRR*, 2020. arXiv:2001.04301v2.

44 Jörg Siekmann, Stephan Hess, Christoph Benzmüller, Lassaad Cheikhrouhou, Armin Fiedler, Helmut Horacek, Michael Kohlhase, Karsten Konrad, Andreas Meier, Erica Melis, Martin Pollet, and Volker Sorge. Loui: Lovely omega user interface. *Formal Aspects of Computing*, 11(3):326–342, 1999. doi:10.1007/s001650050053.

45 Aaron Stockdill, Daniel Raggi, Mateja Jamnik, Grecia Garcia Garcia, and Peter C.-H. Cheng. Considerations in representation selection for problem solving: A review. In Amrita Basu, Gem Stapleton, Sven Linker, Catherine Legg, Emmanuel Manalo, and Petrucio Viana, editors, *Diagrammatic Representation and Inference*, pages 35–51, Cham, 2021. Springer International Publishing.

46 Makarius Wenzel. Asynchronous user interaction and tool integration in isabelle/pide. 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 515–530. Springer, 2014. doi:10.1007/978-3-319-08970-6_33.

47 Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine, and Keenan Crane. Penrose: from mathematical notation to beautiful diagrams. *ACM Trans. Graph.*, 39(4):144, 2020. doi:10.1145/3386569.3392375.

48 Bohua Zhan, Zhenyan Ji, Wenfan Zhou, Chaozhu Xiang, Jie Hou, and Wenhui Sun. Design of point-and-click user interfaces for proof assistants. In Yamine Aït Ameur and Shengchao Qin, editors, *Formal Methods and Software Engineering - 21st International Conference on Formal Engineering Methods, ICFEM 2019, Shenzhen, China, November 5-9, 2019, Proceedings*, volume 11852 of *Lecture Notes in Computer Science*, pages 86–103. Springer, 2019. doi:10.1007/978-3-030-32409-4_6.

49 Jiaje Zhang and Donald A. Norman. Representations in distributed cognitive tasks. *Cognitive Science*, 18(1):87–122, 1994. doi:10.1016/0364-0213(94)90021-3.

Bel-Games: A Formal Theory of Games of Incomplete Information Based on Belief Functions in the Coq Proof Assistant

Pierre Pomeret-Coquot
IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3, Toulouse, France

Hélène Fargier
IRIT, CNRS, Toulouse, France

Érik Martin-Dorel
IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3, Toulouse, France

Abstract

Decision theory and game theory are both interdisciplinary domains that focus on modelling and analyzing decision-making processes. On the one hand, decision theory aims to account for the possible behaviors of an agent with respect to an uncertain situation. It thus provides several frameworks to describe the decision-making processes in this context, including that of belief functions. On the other hand, game theory focuses on multi-agent decisions, typically with probabilistic uncertainty (if any), hence the so-called class of Bayesian games. In this paper, we use the Coq/SSReflect proof assistant to formally prove the results we obtained in [35]. First, we formalize a general theory of belief functions with finite support, and structures and solutions concepts from game theory. On top of that, we extend Bayesian games to the theory of belief functions, so that we obtain a more expressive class of games we refer to as Bel games; it makes it possible to better capture human behaviors with respect to lack of information. Next, we provide three different proofs of an extended version of the so-called Howson–Rosenthal's theorem, showing that Bel games can be casted into games of complete information, i.e., without any uncertainty. We thus embed this class of games into classical game theory, enabling the use of existing algorithms.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Type theory; Theory of computation → Higher order logic; Theory of computation → Algorithmic game theory; Theory of computation → Solution concepts in game theory; Theory of computation → Representations of games and their complexity

Keywords and phrases Game of Incomplete Information, Belief Function Theory, Coq Proof Assistant, SSReflect Proof Language, MathComp Library

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.25

Related Version *Paper-and-pencil proof article*: doi:10.1016/j.ijar.2022.09.010 [35]

Supplementary Material *Software (Formal proofs repo)*: https://github.com/pPomCo/belgames
archived at swh:1:dir:5566e90ea5b3121a0b4f989a7584a251995c297a

Funding *Pierre Pomeret-Coquot*: ANITI, funded by the French "Investing for the Future – PIA3" program under the Grant agreement n°ANR-19-PI3A-0004.

1 Introduction

From a mathematical perspective, measure theory is a fundamental domain to learn and use, notably given its direct application to integration and probability theory. Several works thus focused on formalizing measure theory in type theory, e.g., relying on reference textbooks [16]. Next, probability play a key role in the context of game theory, gathering several multi-agent frameworks that can model situations in many application areas such as economics, politics,


© Pierre Pomeret-Coquot, Hélène Fargier, and Érik Martin-Dorel;
licensed under Creative Commons License CC-BY 4.0

logics, artificial intelligence, biology, and so on. In particular, the framework of Bayesian games (a class of games of incomplete information), has been well-studied by the decision theory community [15, 30]. However, using probability and additive measures appears to be unsatisfactory to model subtle decision-making situations with uncertainty.

In this work, we aim to show that the belief function theory also is amenable to formal proof, and makes it possible to formally verify the correctness of three state-of-the-art algorithms. In [13, 35], we introduced the notion of *Bel games*, which faithfully models games of incomplete information where the uncertainty is expressed within the *Dempster-Shafer theory of belief functions*. This framework naturally encompass Bayesian games, as belief functions generalize probability measures. Also, we generalized the *Howson-Rosenthal theorem* to the framework of Bel games and proposed three transforms which make it possible to cast any Bel game into an equivalent game of complete information (without any uncertainty). Furthermore, these transforms preserve the space complexity of the original Bel game (they produce a game with a succinct representation, corresponding to the class of so-called hypergraphical games).

Contributions. In the present paper, we consolidate the mathematical results previously published in [35], presenting a formal verification of our algorithms using the Coq proof assistant [6]. First, we formalize a general theory of belief functions. Then, we formalize structures and solution concepts for "standard" games, Bayesian games, and Bel games, and we formally prove the correctness of the three transform algorithms, in order to provide strong confidence on these results. The software artifact obtained was released under the MIT license and is available within the official Coq projects OPAM archive. Our formalization effort also resulted in more background lemmas, integrated in the MathComp library. To the best of our knowledge, it is the first time the theory of belief functions is mechanized in a formal proof assistant, and applied to the domain of (formal) game theory of incomplete information.

Related works. Several formalization efforts have been carried out in game theory since 2006, each focusing on a somewhat different fragment: Kaliszyk et al. [33], formalizing foundations of decision making, using Isabelle/HOL; Vestergaard [40] then Le Roux [36], formalizing Kuhn's existence of a Nash equilibrium in finite games in extensive form, using Coq; Lescanne et al. [25], studying rationality of infinite games in extensive form, using Coq; Martin-Dorel et al. [27], studying the probability of existence of winning strategies in Boolean finite games, using Coq; Bagnall et al. [4], formalizing well-known results of algorithmic game theory, using Coq; Dittmann [8], proving the positional determinacy of parity games, using Isabelle/HOL; Le Roux et al. [23], proving that a determinacy assumption implies the existence of Nash equilibrium in 2-player games, using Coq and Isabelle/HOL; this result being combined with that of Dittmann, using Isabelle/HOL [24]. Furthermore, game theory is an important topic in economics: Lange et al. [21], proposing guidelines for formal reasoning; Kaliszyk et al. [20], formalizing microeconomic foundations, using Isabelle/HOL, and Echenim et al. [12], formalizing the Binomial Pricing Model, using Isabelle/HOL. Game theory aside, numerous works have been carried out in proof assistants to formalize probability and/or measure theory. Regarding the Coq proof assistant, we can mention the recent works by Affeldt et al. (on information theory based on discrete probability theory [2]; and measure theory based on MathComp [1]) and Boldo et al. [5], focusing on Lebesgue's integral theory.

Paper outline. We start by introducing a motivating example, for which the Bayesian approach fails but the Dempster-Shafer approach succeeds. Section 3 presents the Dempster-Shafer theory of belief functions, then Section 4 focuses on complete-information games (including hypergraphical games) while Section 5 deals with Bel games; then Section 6 is devoted to our formalization of Howson-Rosenthal's generalized theorem in the scope of n-player Bel games. Finally, Section 7 gives concluding remarks and perspectives. Throughout the paper, we interleave mathematical and formal statements as needed to ensure our formally verified results are well-surveyable: the definitions and results are given first in mathematical syntax, then in Coq syntax when required (i.e., we include the Coq statements of the main theorems and the definitions they depend on, but not those of intermediate lemmas).

2 Motivating Example: the Murder of Mr. Jones

▶ **Example 1** (Inspired by *the Murder of Mr. Jones* [39])**.** Player 1 and Player 2 have to choose a partner which can be either Peter (P), Quentin (Q), or Rose (R). The point is that a crime has been committed, for which these three people only are suspected. Furthermore, a poor-quality surveillance video allows to estimate that there is a 50% chance that the culprit is a man (P or Q), and a 50% chance that it is a woman (R). As to their interest, choosing an innocent people leads to a payoff of \$6k, to be shared between the people making the deal (that is, \$2k or \$3k depending on if the players choose the same partner or not); choosing the culprit yields no payoff (\$0k). Moreover, Player 1 is investigating P and will know whether he is guilty before making the decision. Similarly, Player 2 will know whether R is guilty.

The Bayesian approach claims that any knowledge shall be described by a single subjective probability; it is not well-suited here. Indeed, assume Player 1 learns that Peter is innocent. It should not impact the evidence of 50% chance per sex, so the probability of guilt should become 1/2 for Quentin and 1/2 for Rose. However, in a purely Bayesian view, a prior probability must be made explicit, e.g., by equiprobability assumption: 1/4 for Peter, 1/4 for Quentin, and 1/2 for Rose. After conditioning, the posterior probability would not give 50% chance per sex anymore: equiprobability and conditioning "given Peter is innocent" yields 1/3 for Quentin and 2/3 for Rose. Learning that Peter is innocent would increase the odds against Rose! In the sequel, we will reuse this example to highlight how the framework of belief functions better captures uncertain knowledge.

3 Formalization of Belief Functions for Mono-Agent Decision Making

Modelling mono-agent decision making under uncertainty involves three main tasks. First, knowledge has to be expressed in a well-suited representation, encoding what is known without making extra assumptions. Then, if the agent may learn or observe some event before the decision, one shall identify the relevant conditioning rule. Finally, the agent's preferences must be captured by a compatible decision rule. In this work, we focus on real-valued utility-based decision rules, which evaluate uncertain outcomes so that the agent prefers outcomes with a bigger score. For example, modelling well-known variable phenomena can perfectly be captured in a probabilistic setting: a probability represents the variability; conditional probability updates knowledge; and preferences over uncertain outcomes may be captured by expected utility. Still, this approach may be unsuccessful to model other kinds of uncertainty.

In the sequel, we rely on belief function theory, which generalizes probability theory and enables capturing both variability and ignorance. In this section, we focus on a single decision maker, while the material from Section 4 will deal with multi-agent decision making.

3.1 Belief functions

The theory of belief functions is a powerful toolset from decision theory and statistics. It encompasses two distinct approaches for reasoning under uncertainty: the Dempster-Shafer theory of evidence (DS) [7, 37] and the upper-lower probability theory (ULP) [7, 41]. Both approaches consider a finite set of possible "states of the world" $\Omega = \{\omega_1, \ldots, \omega_n\}$, one of which being the actual state of the world ω^*, and three functions $m : 2^\Omega \to [0,1]$, $\mathrm{Bel} : 2^\Omega \to [0,1]$, and $\mathrm{Pl} : 2^\Omega \to [0,1]$, which all map subsets of Ω to real numbers. Those functions are deducible one from another. In the DS theory, the mass function m is the basic knowledge about the world: $m(A)$ is the part of belief supporting the evidence $\omega^* \in A$, but that does not support smaller claims such as $\omega^* \in B \subset A$. The non-additive continuous measures Bel and Pl indicate how much a proposition is implied by (resp. is compatible with) the knowledge. By contrast, the ULP theory suppose that there is an unknown probability $\Pr^*$ which is bounded by Bel and Pl: $\forall A, \mathrm{Bel}(A) \leq \Pr^*(A) \leq \mathrm{Pl}(A)$; then m is just a concise representation of the family of compatible probabilities.

Example 1 can be understood in both theories. In the DS theory, $m(\{P,Q\}) = m(\{R\}) = 1/2$ directly encodes the given evidences. In the ULP theory, one rather considers the family of probabilities $(\Pr_x)_x$ which satisfy $\Pr_x(\{P,Q\}) = \Pr_x(\{R\}) = 1/2$ (see Table 1).

Table 1 Prior knowledge from Example 1 – in the DS theory, m directly describes the knowledge, in the ULP theory, m describes a family of probability measures $(\Pr_x)_{x \in [0,\, 0.5]}$.

$A \subseteq \Omega$	$\emptyset$	$\{P\}$	$\{Q\}$	$\{R\}$	$\{P,Q\}$	$\{P,R\}$	$\{Q,R\}$	$\{P,Q,R\}$
$m(A)$	0	0	0	0.5	0.5	0	0	0
$\mathrm{Bel}(A)$	0	0	0	0.5	0.5	0.5	0.5	1
$\mathrm{Pl}(A)$	0	0.5	0.5	0.5	0.5	1	1	1
$\Pr_x(A)$	0	x	$0.5 - x$	0.5	0.5	$0.5 + x$	$1 - x$	1

In this work, we follow the DS approach and formalize notions in terms of the function m. We chose to use the Coq proof assistant, along with the SSReflect tactic langage and the MathComp library [26]. This combination offers several features that contribute to facilitate the formalization: dependent types (making it possible to easily grasp usual definitions in game theory, and account for the variability of actions spaces w.r.t. individual agents), reflection prodicates (to easily go back and forth between decidable Boolean predicates and their propositional counterpart), packed classes and structure inference (making it possible to get formal statements as concise and legible as "LaTeX" ones) as well as the availability of comprehensive theories of finite sets or functions with finite support, endowed with decidable equality, and big operators such as $\sum$ or $\prod$.[1] Regarding automation (which is sometimes a criterium to choose a particular theorem prover over another): the formal verification of our results especially involves proofs by rewriting, very often under the binder of a big operator: the `under` tactic was instrumental to this aim [28]. Besides using this tactic, the formalization did not highlight a particular need to automate specific fragments or recurring proof goals.

▶ **Definition 2** (Frame of discernment)**.** *A* frame of discernment *is a finite set Ω, representing the possible states of the world. One of them is* the actual state of the world ω^*.

[1] MathComp notations: `{set X}` denotes finite sets over `(X : finType)`, `A:&:B` $= A \cap B$, `A:|:B` $= A \cup B$, `~:A` $= A^c$, `set0` $= \emptyset$; `{ffun X -> Y}` denotes the type of finite support functions from `(X : finType)` to `(Y : Type)`; and for any `(T : X -> Type)`, `{ffun forall x : X, T x}` denotes the type of finite support functions with a dependently-typed codomain, mapping any `(x : X)` to an element of `(T x)`.

▶ **Definition 3** (Events). *A set $A \subseteq \Omega$ is an* event *which represents the proposition "$\omega^* \in A$". All set functions we consider (m,* Bel, Pl*) map events to real numbers within* $[0, 1]$.

The set Ω is endowed with MathComp's finite type structure, that is, `(W : finType)`. It makes it possible to use set operations and big operators. The carrier of the functions m, Bel, and Pl is `(R : realFieldType)`: only field operations are needed for this work.

▶ **Definition 4** (Basic probability assignment). *A basic probability assignment (bpa), a.k.a. mass function, is a set-function $m : 2^\Omega \to [0, 1]$ such that:*

$$m(\emptyset) = 0 \quad \textit{and} \quad \sum\nolimits_{A \subseteq \Omega} m(A) = 1 \quad \textit{and} \quad \forall A \subseteq \Omega,\ m(A) \geq 0. \quad \text{Formally:} \tag{1}$$

```
Definition bpa_axiom m := [&& m set0 == 0, \sum_A m A == 1 & [∀A, m A >= 0]].
Structure bpa := { bpa_val :> {ffun {set W} -> R} ; bpa_ax : bpa_axiom bpa_val }.
```

▶ **Definition 5** (Belief function, plausibility measure). *Given a bpa m over Ω, the associated belief function* $\mathrm{Bel} : 2^\Omega \to [0, 1]$ *and plausibility measure* $\mathrm{Pl} : 2^\Omega \to [0, 1]$ *are defined by:*

$$\mathrm{Bel}(A) = \sum\nolimits_{B \subseteq A} m(B) \quad \textit{and} \quad \mathrm{Pl}(A) = \sum\nolimits_{B \cap A \neq \emptyset} m(B).$$

▶ **Proposition 6** (Duality). *For any $A \subseteq \Omega$, $\mathrm{Pl}(A) = 1 - \mathrm{Bel}(A^c)$ and $\mathrm{Bel}(A) = 1 - \mathrm{Pl}(A^C)$.*

▶ **Proposition 7** (Super- and sub-additivity). Bel *is super-additive while* Pl *is sub-additive: for disjoint sets $A, B \subseteq \Omega$, $\mathrm{Bel}(A \cup B) \geq \mathrm{Bel}(A) + \mathrm{Bel}(B)$ and $\mathrm{Pl}(A \cup B) \leq \mathrm{Pl}(A) + \mathrm{Pl}(B)$.*

▶ **Proposition 8** (Bounds). *For any $A \subseteq \Omega$, $0 = \mathrm{Bel}(\emptyset) = \mathrm{Pl}(\emptyset) \leq \mathrm{Bel}(A) \leq \mathrm{Pl}(A) \leq \mathrm{Bel}(\Omega) = \mathrm{Pl}(\Omega) = 1$.*

▶ **Definition 9** (Focal elements, focal set). *Given a bpa m over Ω, any subset $A \subseteq \Omega$ with a non-zero mass $m(A)$ is called* focal element*, and the set of focal elements of m is called* the focal set of m *and denoted by $\mathcal{S}_m$. In other words, $A \in \mathcal{S}_m$ iff $m(A) > 0$.*

▶ **Proposition 10** (Focal elements, focal set). *Given a bpa m over Ω, Definition 5 can straightforwardly be rephrased by rewriting the sums over the focal set:*

$$\mathrm{Bel}(A) = \sum_{\substack{B \in \mathcal{S}_m \\ B \subseteq A}} m(B) \quad \textit{and} \quad \mathrm{Pl}(A) = \sum_{\substack{B \in \mathcal{S}_m \\ B \cap A \neq \emptyset}} m(B).$$

Next, we recall a standard "complexity definition" about belief functions, that will prove useful to characterize probability measures:

▶ **Definition 11** (k-additivity). *For any bpa m, let $k = \max_{B \in \mathcal{S}_m} |B|$ be the maximal cardinality of its focal elements. Then, m is said to be k-additive.*

▶ **Definition 12** (Probability measure). *Given a bpa m over Ω, if m is 1-additive, i.e. if all focal elements are singletons, then* Bel $=$ Pl *is a discrete probability measure, associated with the distribution* `dist m` $: \Omega \to [0, 1]$ defined by $x \mapsto m(\{\omega\})$.

```
Structure proba := {proba_val :> bpa; proba_ax: \max_(B in focalset proba_val) #|B| == 1}.
Definition dist (m : proba) := fun w => m [set w]. (*[set w] corresponds to {w} *)
```

3.2 Conditioning in the Belief Function Theory

Conditioning is the operation that captures knowledge revision (fact learning) as well as focusing (hypothesis) [11, 9, 14]. By turning a prior bpa into a posterior "given an event C", one updates the knowledge so it now asserts that C is certain. Several conditioning rules for belief functions have been proposed, depending on the DS or ULP interpretation (cf. Section 3.1) and on the kind of update it involves. Starting from the same prior bpa, they yield distinct posteriors – they indeed capture distinct operations.

Before dealing with conditional events $(\cdot \mid C)$ – read "given C" – a precondition happens to be necessary: on the technical side, it avoids division-by-zero, and on the semantics side, it means one cannot learn that an impossible event holds. Since the definition of this precondition is specific to each conditioning rule, we abstract it away in the form of a `revisable` predicate, which indicates whether an event can be assumed.

▶ **Definition 13** (Conditioning). *Given a bpa m, a predicate* $\texttt{revisable}_m : \Omega \to \{1, 0\}$, *and an event $C \subseteq \Omega$ such that* $\texttt{revisable}_m(C)$ *holds, a conditioning turns m into a bpa $m(\cdot \mid_{cond} C)$ such that the complement C^c of C is impossible, i.e.,* $\mathrm{Bel}(C^c \mid C) = 0$. *Formally:*

```
Definition conditioning_axiom (revisable : bpa -> pred {set W})
                              (cond : ∀m C, revisable m C -> bpa) :=
  ∀m C (Hrev : revisable m C), Bel (cond m C Hrev) (~:C) = 0.
```

In other words, assuming m is `revisable` by the event C implies that if one learns that C holds, then one also learns that no evidence for the complement C^c can hold. Next, we formalize a `conditioning` structure that encapsulates the `revisable` predicate, the conditioning algorithm itself – which turns a `revisable` prior in its posterior "given C" – and a proof of the `conditioning_axiom`:

```
Structure conditioning := { revisable : bpa -> pred {set W} ;
                            cond_val :> ∀m C, revisable m C -> bpa ;
                            cond_ax : conditioning_axiom cond_val }.
```

The most common conditioning is the so-called Dempster's conditioning [7], which captures knowledge revision (i.e., fact learning). In the DS framework, it is understood as a transfer of parts of beliefs: learning that C holds, $m(B)$ is transferred to $B \cap C$ if it is not empty, or discarded otherwise (then the posterior has to be renormalized due to Equation (1)). That is, the evidence now concerns $B \cap C$, the only possible states of the world "given that C holds". In the ULP framework, it is understood as a max-likelihood conditioning: the posterior probability family delimited by $\mathrm{Bel}(\cdot \mid C)$ and $\mathrm{Pl}(\cdot \mid C)$ is the conditioning of those prior probabilities which assign the maximal probability to event C, that we now know for sure.

▶ **Definition 14** (Dempster's conditioning). *For any bpa m and any event C such that $\mathrm{Pl}(C) \neq 0$, Dempster's conditioning defines the bpa: $m(A \mid_D C) = \sum_{B \cap C = A \neq \emptyset} m(B) / \mathrm{Pl}(C)$. Formally:*

```
Definition Dempster_revisable m C := Pl m C != 0.
Definition Dempster_fun (m : bpa) (C : {set W}) := [ffun A : {set W} =>
  if A == set0 then 0
  else \sum_(B : {set W} | (B \in focalset m) && (B :&: C == A)) m B / Pl m C].
Program Definition Dempster_cond m C (Hrev : Dempster_revisable m C) : bpa :=
  {| bpa_val := Dempster_fun m C ; bpa_ax := _ |}.
Program Definition Dempster_conditioning : conditioning :=
  {| cond_val := Dempster_cond ; cond_ax := _ |}.
```

▶ **Example 15** (Knowledge revision, follow-up of Example 1/Table 1). Dempster's conditioning is the conditioning approach fitting our example (see [9] for details). Suppose e.g. the murderer is Q; Player 1 learns $\omega^* \notin \{P\}$, i.e., $\omega^* \in \{Q, R\}$. From this viewpoint, the evidence concerning men now only concerns Q: the knowledge becomes $m(\{Q\}) = m(\{R\}) = 0.5$ (Fig. 1, center). Player 2 learns $\omega^* \notin \{R\}$, i.e., $\omega^* \in \{P, Q\}$. From this viewpoint, the evidence about women is discarded: the knowledge becomes $m(\{P, Q\}) = 1$ (Fig. 1, right).

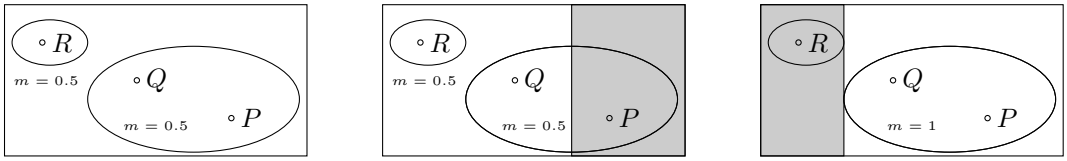


Figure 1 Prior (left) and posteriors given $\{Q, R\}$ (center) and given $\{P, Q\}$ (right). White and gray areas denote possible and impossible events – circles denote focal elements.

▶ **Proposition 16** (Dempster's conditioning, Pl). *For any bpa m and any event C such that* $\mathrm{Pl}(C) \neq 0$, *it holds that* $\mathrm{Pl}(A \mid_D C) = \mathrm{Pl}(A \cap C) / \mathrm{Pl}(C)$.

Two other rules have been proposed and called strong (resp. weak) conditioning [34]; the former, also known as geometrical conditioning [38], is another rule capturing knowledge revision; the latter is seldom used since it yields non-intuitive results (e.g., it may happen that $\mathrm{Bel}(C \mid C) < 1$). We also formalize these two rules below.

▶ **Definition 17** (Strong conditioning). *For any bpa m and any event C s.t.* $\mathrm{Bel}(C) \neq 0$, *the strong conditioning is defined by the bpa* $m(A \mid_S C) = m(A) / \mathrm{Bel}(C)$ *if* $A \subseteq C$, *0 otherwise.*

```
Definition Strong_revisable m C := Bel m C != 0.
Definition Strong_fun (m : bpa) (C : {set W}) := [ffun A : {set W} =>
  if (A != set0) && (A \subset C) then m A / Bel m C else 0].
Program Definition Strong_cond m C (Hrev : Strong_revisable m C) : bpa :=
  {| bpa_val := Strong_fun m C ; bpa_ax := _ |}.
Program Definition Strong_conditioning : conditioning :=
  {| cond_val := Strong_cond ; cond_ax := _ |}.
```

▶ **Proposition 18** (Strong conditioning, Bel). *For any bpa m and any event C such that* $\mathrm{Bel}(C) \neq 0$, *it holds that* $\mathrm{Bel}(A \mid_S C) = \mathrm{Bel}(A \cap C) / \mathrm{Bel}(C)$.

▶ **Definition 19** (Weak conditioning). *For any bpa m and any event C such that $Pl(C) \neq 0$, the weak conditioning is defined by the bpa: $m(A \mid_W C) = m(A)/Pl(B)$ if $A \cap B \neq \emptyset$, 0 otherwise.* Formally:

```
Definition Weak_revisable m C := Pl m C != 0.
Definition Weak_fun (m : bpa) (C : {set W}) := [ffun A : {set W} =>
  if A :&: C != set0 then m A / Pl m C else 0].
Program Definition Weak_cond m C (Hrev : Weak_revisable m C) : bpa :=
  {| bpa_val := Weak_fun m C ; bpa_ax := _ |}.
Program Definition Weak_conditioning : conditioning :=
  {| cond_val := Weak_cond ; cond_ax := _ |}.
```

▶ **Proposition 20** (Weak conditioning, Bel). *For any bpa m and any event C such that* $\mathrm{Pl}(C) \neq 0$, *it holds that* $\mathrm{Bel}(A \mid_W C) = (\mathrm{Bel}(A) - \mathrm{Bel}(A \setminus C)) / \mathrm{Pl}(C)$.

3.3 Decision Making with Belief Functions

Consider a single agent decision involving several actions; let A denote the set of all these actions. Also, assume that the outcome of choosing any $a \in A$ is not certain: it may lead to several oucomes depending on the actual state of the world ω^*. The agent's preferences on outcomes (which are left implicit here) are expressed by a real-valued utility function $u : A \times \Omega \to \mathbb{R}$: $u(a, \omega) > u(a', \omega')$ would mean the agent prefers the outcome of a when $\omega^* = \omega$ to the outcome of a' when $\omega^* = \omega'$. For any action a, let $u_a : \Omega \to \mathbb{R}$ denote the partial application of u: u_a provides the utility of a depending on the state of the world ω.

Preferences under uncertainty are then defined on u_a's: a relation $u_a \succ u_{a'}$ would encode the fact the agent prefers a to a'. In a probabilistic setting, it is meaningful to consider u_a's expectation w.r.t. the probability (hence the name *expected utility*). Using bpa's, several approaches were defined, each modelling various preferences when facing ignorance. In [35], we analyzed three standard functions that generalize expected utility. They provide real values, and thus lead to completely ordered preferences over actions (since every two actions are directly comparable from their score). We denoted them CEU, JEU, and TBEU, respectively, for Choquet–, Jaffray–, and Transferable Belief–Expected Utility. We have shown they are all expressible as the integration of a particular $\varphi_{u_a}^{\mathrm{XEU}}$ function (resp. $\varphi_{u_a}^{\mathrm{CEU}}$, $\varphi_{u_a}^{\mathrm{JEU}}$, and $\varphi_{u_a}^{\mathrm{TBEU}}$) over the powerset 2^Ω. Those $\varphi_{u_a}^{\mathrm{XEU}}$ functions are themselves parameterized by $u_a = \omega \mapsto u(a, \omega)$, that is, by the utility function when a is chosen. As a result, these three scoring functions can be captured by instances of a single higher-order function, which we named XEU.

▶ **Definition 21** (Generalized expected utility). *For any bpa m, any utility function $u : A \times \Omega \to \mathbb{R}$, and any $a \in A$, let us pose $u_a = \omega \mapsto u(a, \omega)$. Let $\varphi : (\Omega \to \mathbb{R}) \to \left(2^\Omega \to \mathbb{R}\right)$ be a parameter function. We then consider the following generalized expected utility of a:*

$$\mathrm{XEU}(m)(\varphi(u_a)) = \sum\nolimits_{B \in \mathcal{S}_m} m(B) \times \varphi(u_a)(B). \quad \text{Formally:}$$

```
Definition XEU (m : bpa) (phi_u_a : {ffun {set W} -> R}) : R :=
  \sum_(B in focalset m) m B * phi_u_a B.
```

Let us review these φ^{XEU} functions, their underlying intuition and formal definition in Coq.

A very common scoring function for belief functions is the Choquet discrete integral (CEU). It models a somehow pessimistic agent. In the ULP interpretation, Bel and Pl delimit a family of probabilities; the CEU computes the minimal expected utility that the family allows. In the DS interpretation, each mass is an evidence supporting an event, for which the CEU only consider its worst-case utility if the considered choice is made.

▶ **Definition 22** (Choquet expected utility). *For any bpa m, any utility function $u : A \times \Omega \to \mathbb{R}$ and any action $a \in A$, the Choquet expected utility of $u_a : \Omega \to \mathbb{R}$ is:*

$$\mathrm{CEU}(m)(u_a) = \sum\nolimits_{B \in \mathcal{S}_m} m(B) \times \min_{\omega \in B} u_a(\omega) = \mathrm{XEU}(m)(\varphi^{\mathrm{CEU}}(u_a)),$$

with $\varphi^{\mathrm{CEU}}(u_a)(B) = \min_{\omega \in B} u_a(\omega)$.

This expression is a weighted sum indexed by the set of focal elements, which is nonempty: using the min operator is legit. Formally, the functions φ^{CEU} and CEU are defined as follows:

```
Definition fCEU (u_a : W -> R) : {set W} -> R :=
  fun B => match minS u_a B with  Some r => r | None => 0  end.
Definition CEU (m : bpa) (u_a : W -> R) := XEU m (fCEU u_a).
```

Another rule, axiomatized by Jaffray [18, 19], is a kind of Hurwicz criterion (i.e., a linear combination over the min. and max. utility reached for each focal element). The parameter coefficients make it possible to locally modulate the pessimism of the modelled agent.

▶ **Definition 23** (Jaffray expected utility). *For any bpa m, any utility function $u : A \times \Omega \to \mathbb{R}$ and any action $a \in A$, the Jaffray expected utility of $u_a : \Omega \to \mathbb{R}$ is parameterized by a family of coefficients $\alpha_{(x_*,x^*)} \in [0,1]$ for each possible utility values $x_* \leq x^*$. For any $B \neq \emptyset$, let us pose $B_* = \min_{\omega\in B} u_a(\omega)$ and $B^* = \max_{\omega\in B} u_a(\omega)$. The Jaffray expected utility of u_a is:*

$$\mathrm{JEU}^\alpha(m)(u_a) = \sum\nolimits_{B\in\mathcal{S}_m} m(B)\times\left(\alpha_{(B_*,B^*)} \times B_* + (1-\alpha_{(B_*,B^*)}) \times B^*\right) = \mathrm{XEU}(m)(\varphi^{\mathrm{JEU}^\alpha}(u_a)),$$

with $\varphi^{\mathrm{JEU}^\alpha}(u_a)(B) = \alpha_{(B_*,B^*)} \times B_* + (1-\alpha_{(B_*,B^*)}) \times B^*$. Formally:

```
Definition fJEU (α : R -> R -> R) (u_a : W -> R) : {set W} -> R :=
  fun B => match minS u_a B, maxS u_a B with
  | Some rmin, Some rmax => let alp := α rmin rmax in alp * rmin + (1-alp) * rmax
  | _, _ => 0   end.
Definition JEU α (m : bpa) (u_a : W -> R) := XEU m (fJEU α u_a).
```

Finally, in the Transferable Belief Model [39], the decision rule is made by recovering a "pignistic" probability distribution[2] BetP that serves only for the choice, at the very moment where the decision is made. So, the equiprobability assumption is made, but after conditionings, if any. The score of an action is then the expected utility w.r.t. $\mathrm{BetP} : \omega \mapsto \sum_{\substack{B\in\mathcal{S}_m \\ \omega\in B}} m(B)/|B|$, that we show to be equivalent to the following definition.

▶ **Definition 24** (Transferable Belief Model expected utility). *For any bpa m, any utility function $u : A \times \Omega \to \mathbb{R}$ and any action $a \in A$, the TBEU of $u_a : \Omega \to \mathbb{R}$ is defined by:*

$$\mathrm{TBEU}(m)(u_a) = \sum\nolimits_{B\in\mathcal{S}_m} m(B) \times \sum\nolimits_{\omega\in B} u_a(\omega) \;/\; |B| = \mathrm{XEU}(m)(\varphi^{\mathrm{TBEU}}(u_a))$$

with $\varphi^{\mathrm{TBEU}}(u_a)(B) = \sum_{\omega\in B} u_a(\omega)/|B|$. *Formally:*

```
Definition fTBEU (u_a : W -> R) := fun B => \sum_(w in B) u_a w / #|B|%:R.
Definition TBEU (m : bpa) (u_a : W -> R) := XEU m (fTBEU u_a).
```

▶ **Proposition 25.** *CEU, JEU, and TBEU all generalize the expected utility criterion. For any probability distribution p, any utility function $u : A \times \Omega \to \mathbb{R}$ and any action $a \in A$,* $\mathrm{CEU}(p)(u_a) = \mathrm{JEU}^\alpha(p)(u_a) = \mathrm{TBEU}(p)(u_a) = \sum_{\omega\in\Omega} p(\omega) \times u_a(\omega)$.

In these formal proofs, the key ingredient is the fact that the criteria satisfy the natural property that $\forall u_a,\ \forall \omega \in \Omega,\ \varphi(u_a)(\{\omega\}) = u_a(\omega)$.

4 Formalization of Several Classes of Games of Complete Information

Game theory is a subdomain of multi-agent decision making [29, 30]. In this paper, we focus on simultaneous games, in which *players* make their choice (called *action* or *pure strategy*) without knowing others' choices in advance; the outcome of an action depends on the choices of other agents. A typical problem amounts to identifying which actions are relevant from the viewpoint of a player, assuming others don't cooperate but strive to increase their own utility. In this section, we consider situations where there is no uncertainty.

[2] The names "pignistic" and BetP are references to classical Bayesian justification in decision theory, where both utilities and beliefs are elicited by considering limits of agent's agreement to a panel of bets.

4.1 Games of Complete Information

▶ **Definition 26** (Game of complete information). *A CGame is a tuple* $G = \big(I, (A_i, u_i)_{i\in I}\big)$ *where* I *is a finite set of players; for each Player* i, A_i *is the set of their actions;* $u_i : A \to \mathbb{R}$ *is an utility function, assigning an utility value to each "action profile", i.e., a vector of actions, also called "pure strategy profile"* $a = (a_1, \ldots, a_n) \in A = A_1 \times \cdots \times A_n$. *Player* i *prefers the outcome of profile* a *to that of* a' *iff* $u_i(a) > u_i(a')$.

We formalize such "profiles-for-CGames" ($a \in \prod_{i\in I} A_i$) using MathComp's dependently-typed finite support functions, hence:

```
Definition cprofile (I : finType) (A : I -> eqType) := {ffun ∀i : I, A i}.
Definition cgame (I : finType) (A : I -> eqType) := cprofile A -> I -> R.
```

One of the most prominent solution concept in game theory is that of Nash equilibrium [31]:

▶ **Definition 27** (Nash equilibrium). *A pure Nash equilibrium is a profile such that no player has any incentive to "deviate". For any pure strategy profile* a *and any Player* i, *let* a_{-i} *be the restriction of* a *to the actions of Players* $j \neq i$, a'_i *an action of Player* i, *then* $a'_i.a_{-i}$ *denotes the profile* a *where the strategy of Player* i *has been switched to* a'_i *(called* `change_strategy a a'_i` *in Coq). A profile* a *is a pure Nash equilibrium iff* $\forall i, \forall a'_i,\ u_i(a) \not< u_i(a_i.a_{-i})$*:*

```
Definition change_strategy (p : cprofile A) (i : I) (a'_i : A i) : cprofile A
Definition Nash_equilibrium (G : cgame) (a : cprofile A) : bool :=
  [∀i : I, [∀a'_i : A i, ~~ (G a i < G (change_strategy a a'_i) i)]].
```

▶ **Example 28.** Consider Example 1 anew; suppose one knows P is the murderer. The situation is captured by the CGame $G = (I, (A_i, u_i)_{i\in I})$ where $I = \{1, 2\}$ is the set of players, $A_i = \{P_i, Q_i, R_i\}$ the set of actions of Player i (choosing P, Q or R) and the u_i's of Table 2. Here, both (Q_1, R_2) and (R_1, Q_2) are Nash equilibria.

Table 2 Utility functions of Example 28 (it is known that P is the murderer). The pair $(u_1(a_1.a_2), u_2(a_1.a_2))$ is read at the intersection of line a_1 and column a_2.

	P_2	Q_2	R_2
P_1	$(0,0)$	$(0,3)$	$(0,3)$
Q_1	$(3,0)$	$(2,2)$	$\boxed{(3,3)}$
R_1	$(3,0)$	$\boxed{(3,3)}$	$(2,2)$

When there is some variability regarding action choices (e.g., for repeated games), it is meaningful to look for mixed strategies. A mixed strategy ρ_i of Player i is a probability over A_i, and a mixed strategy profile $\rho = (\rho_1, \ldots, \rho_n)$ is a vector of mixed strategies:

```
Definition mixed_cprofile := cprofile (fun i => [eqType of proba R (A i)]).
```

A mixed strategy profile ρ defines a probability over the set of pure strategy profiles, namely $p_\rho(a) = \prod_{i\in I} \rho_i(a_i)$. We package this data in a `proba` structure (Definition 12):

```
Definition mk_prod_proba (p : ∀i : X, proba R (A i)) : {ffun cprofile A -> R} :=
  [ffun a : cprofile A => \prod_i dist (p i) (a i)].
Definition prod_proba (p : ∀i : I, proba R (A i)) (i0 : I) : proba R (cprofile A).
```

Last, the utility of a mixed strategy profile is the expected utility w.r.t. the probability over pure strategy profiles, and the notion of Nash equilibrium extends straightforwardly:

```
Definition ms_util (G : cgame R A) (mp : mixed_cprofile) (i : I) : R :=
  \sum_(p : cprofile A) (dist (prod_proba mp witnessI mp) p) * (G p i).
Definition ms_Nash_equilibrium (G : cgame R A) (mp : mixed_cprofile) : Prop :=
  ∀i (si : proba R (A i)), ~ ms_util G mp i < ms_util G (change_strategy mp si) i.
```

A standard reduction [22, Def. 4.6.1] amounts to viewing a mixed equilibrium of a game $(N, (A_i, u_i)_{i\in N})$ as a pure equilibrium in the mixed extension $(N, (\mathcal{A}_i, u_i)_{i\in N})$, where $\mathcal{A}_i$ is the set of mixed strategies over A_i. Formally:

```
Definition mixed_cgame (G:cgame R A) : cgame R (fun i => [eqType of proba R (A i)])
  := fun mp i => ms_util G mp i.
Theorem mixed_cgameE G mp i :  ms_utility G mp i = (mixed_cgame G) mp i.
Theorem ms_NashE (G : cgame R A) (mp : mixed_cprofile) :
  ms_Nash_equilibrium G mp <-> Nash_equilibrium (mixed_cgame G) mp.
```

4.2 Hypergraphical Games

Some games of complete information can be expressed succinctly as hypergraphical games [32, 42], where the utility is not defined globally but locally (namely, split in several "local games"). This yields a hypergraph, where vertices denotes players and hyperedges denote local games. Formally, a hypergraphical game is a tuple $G = \big(I, E, (A_i)_{i\in I}, (u_i^e)_{e\in E, i\in e}\big)$, where I is the set of players, $E \subseteq 2^I$ is the set of local games (in any local game $e = \{a, b, c, \dots\}$, Players a, b, $c\dots$ are playing), A_i is the set of actions of Player i and $u_i^e : A_e \to \mathbb{R}$ is the utility function of Player i in the local game e ($A_e = \prod_{i\in e} A_i$ is the set of local profiles related to e's players). A hypergraphical game with 2-player local games is called a polymatrix.

In our formalization, local games are indexed by the finite type `(localgame : finType)`; players playing a local game `(lg : localgame)` are those who verify the Boolean predicate `(plays_in lg)`; `plays_in` thus formalizes E as a family of sets of players:

```
Variables (localgame : finType) (plays_in : localgame -> pred I).
```

For any local game `lg`, local profiles are profiles that involve only players which `plays\_in lg`:

```
Definition localprof (lg : localgame) :=
  {ffun ∀s : {i : I | plays_in lg i}, A (val s)}.
```

In hypergraphical games, every player chooses one action, and plays it in every local game they are involved in. The global utility of a player is the sum of the locally obtained utilities: $u_i(a) = \sum_{\substack{e\in E\\ i\in e}} u_i^e(a_e)$, where $a_e \in A_e$ is the restriction of a to indices of e. Thus, an hypergraphical game is a CGame that is specified by its local utility functions:

```
Definition hg_game (u : ∀lg, localprof lg -> {i : I & plays_in lg i} -> R) : cgame
    := fun a i => \sum_(s : {lg : localgame | plays_in lg i})
                        u (tag s) [ffun i => a (val i)] (exist _ i (tagged s)).
```

5 Bel Games

Harsanyi has proposed [15] a model for decision-making situations where players may have some uncertainty about other players, their actions, their utility functions, or more generally about any parameter of the game. To model such situations, the partially known parameters

are expressed by so-called types:[3] each Player i has a set of possible types Θ_i. Each type $\theta_i \in \Theta_i$ represents a possible parameter describing Player i's characteristics and knowledge. Every Player i knows (or learns) their own type $\theta_i \in \Theta_i$ at the time of choosing an action. It may or may not be correlated with other players' types, so it is possible to model players that are not aware of other players' type as well as players with some knowledge about them.

Harsanyi defined the model of games of incomplete information where players' knowledge is given by a subjective probability and preferences agree with the expected utility: the so-called class of Bayesian games. In this setting, a probability measure expresses the knowledge on type configurations (the frame of discernment being the cartesian product of all players' types, that is, $\Omega = \prod_{i\in N} \Theta_i = \Theta$). Games of incomplete information were already defined in a possibilistic setting by Ben Amor et al. [3], which makes it possible to represent uncertainty using possibility theory [10]. We further extend this framework to Belief functions (encompassing both Bayesian games and possibilistic games) [13, 35].

▶ **Definition 29** (Bel game). *A Bel game* [35] *is defined by a tuple* $G = \big(I, (A_i, \Theta_i, u_i)_{i\in I}, m\big)$*:*

- I *is the finite set of players;*
- A_i *is the set of actions of Player* i*;* Θ_i *is the finite set of types of Player* i*;*
- $u_i : A \times \Theta \to \mathbb{R}$ *is the utility function of Player* i*; it depends on the joint action* $(a_1, \dots, a_n) \in A := \prod_{i\in I} A_i$ *and on the type configuration* $(\theta_1, \dots, \theta_n) \in \Theta := \prod_{i\in I} \Theta_i$*;*
- $m : 2^\Theta \to [0, 1]$ *is a bpa which describes the prior knowledge.*

Formally speaking, a Bel game is fully defined by two elements: the bpa (prior knowledge) and the utility functions (the players' preferences). This pair is parameterized by three types `I`, the players; `A`, the family of actions $(A_i)_i$; and `T`, the family of types $(\Theta_i)_i$:

```
Definition belgame (I : finType) (A : I -> eqType) (T : I -> finType) :=
  (bpa R (cprofile T) * (cprofile A -> cprofile T -> player -> R)).
```

▶ **Example 30** (Bel game). We now are able to express Example 1 with a Bel game $G = (I, (A_i, \Theta_i, u_i)_{i\in I}, m)$. The set of players is $I = \{1, 2\}$, their action sets are $A_i = \{P_i, Q_i, R_i\}$. Player 1 will learn either that P is the murderer ($\omega^* \in \{P\}$) or that he is not ($\omega^* \in \{Q, R\}$): Player 1's type set is $\Theta_1 = \{\mathtt{P}, \bar{\mathtt{P}}\}$. Similarly, Player 2 will learn either that R is the murderer ($\omega^* \in \{R\}$) or that she is not ($\omega^* \in \{P, Q\}$), so $\Theta_2 = \{\mathtt{R}, \bar{\mathtt{R}}\}$. The knowledge is expressed over $\Theta = \Theta_1 \times \Theta_2$: since $(\mathtt{P}, \bar{\mathtt{R}}) \equiv P$, $(\bar{\mathtt{P}}, \bar{\mathtt{R}}) \equiv Q$, $(\bar{\mathtt{P}}, \mathtt{R}) \equiv R$ and $(\mathtt{P}, \mathtt{R})$ is impossible, the knowledge is $m\big(\{(\mathtt{P}, \bar{\mathtt{R}}), (\bar{\mathtt{P}}, \bar{\mathtt{R}})\}\big) = m\big(\{(\bar{\mathtt{P}}, \mathtt{R})\}\big) = 0.5$. Finally, utility functions are given in Table 3.

Table 3 Utility functions of Example 30 for $\theta = (\mathtt{P}, \bar{\mathtt{R}})$ (left, P is the murderer), $\theta = (\bar{\mathtt{P}}, \bar{\mathtt{R}})$ (center, Q is the murderer) and $\theta = (\bar{\mathtt{P}}, \mathtt{R})$ (right, R is the murderer). Configuration $\theta = (\mathtt{P}, \mathtt{R})$ can't occur.

	P_2	Q_2	R_2
P_1	$(0, 0)$	$(0, 3)$	$(0, 3)$
Q_1	$(3, 0)$	$(2, 2)$	$(3, 3)$
R_1	$(3, 0)$	$(3, 3)$	$(2, 2)$

	P_2	Q_2	R_2
P_1	$(2, 2)$	$(3, 0)$	$(3, 3)$
Q_1	$(0, 3)$	$(0, 0)$	$(2, 2)$
R_1	$(3, 3)$	$(3, 0)$	$(2, 2)$

	P_2	Q_2	R_2
P_1	$(2, 2)$	$(3, 3)$	$(3, 0)$
Q_1	$(3, 3)$	$(2, 2)$	$(3, 0)$
R_1	$(0, 3)$	$(0, 3)$	$(0, 0)$

Since players know their own type before choosing their action, a pure strategy of Player i becomes a function $\sigma_i : \Theta_i \to A_i$: having the type θ_i, Player i will play $\sigma_i(\theta_i) \in A_i$. Next, a strategy profile $\sigma = (\sigma_1, \dots, \sigma_n)$ is a vector of such functions:

[3] Thus, *type* can refer to a type-theory concept or a game-theory one. Context will allow to disambiguate.

```
Definition iprofile I A T := cprofile (fun i => [eqType of {ffun T i -> A i}]).
```

If the actual type configuration is $\theta = (\theta_1, \dots, \theta_n)$, then for any strategy profile σ we denote by $\sigma^\theta = \big(\sigma_1(\theta_1), \dots, \sigma_n(\theta_n)\big) \in A$ the action profile that will actually be played:

```
Definition proj_iprofile I A T (p : iprofile A T) : cprofile A :=
  fun theta => [ffun i => p i (theta i)].
```

In the following, we denote by $u_{i,\sigma} : \Theta \to \mathbb{R}$ the function mapping states of the world ω to the corresponding utility of σ for Player i. It is defined by $u_{i,\sigma}(\theta) = u_i(\sigma^\theta, \theta)$.

In Bayesian games, the global utility of a strategy profile σ for Player i with type θ_i is the expected utility w.r.t. the conditioned probability distribution "given θ_i". *In Bel games,* both expectation and conditioning have to be made explicit, to properly model agents' preferences and knowledge updates. For example, studying a Bel game with Dempster's conditioning and CEU expectation implies that the utility of a given strategy profile σ for Agent i with type θ_i is $\sum_{B\subseteq\Omega} m(B \mid_D \theta_i) \times \min_{\theta' \in B} u_i(\sigma^{\theta'}, \theta')$. Doing so, we need to ensure that conditioning is meaningful and technically possible, i.e., that the bpa is `revisable` given any type of any player. For the sake of readability, we now introduce two shorthands: `Tn`, representing the set Θ gathering all type configurations; and `event_ti` $:= \theta_i \mapsto \{\theta' \in \Theta \mid \theta'_i = \theta_i\}$:

```
Notation Tn := [finType of {dffun ∀i : I, T i}].
Definition event_ti i (ti : T i) := [set t : Tn | t i == ti].
```

A *proper Bel games*, in which conditioning is safe, shall satisfy the predicate:

```
Definition proper_belgame A T (G : belgame A T) (cond : conditioning R Tn) : bool
  := [∀i : player, [∀ti : T i, revisable cond G.1 (event_ti ti)]].
```

▶ **Definition 31** (Utility in a Bel game). *For any Bel game* $G = \big(I, (A_i, \Theta_i, u_i)_{i\in I}, m\big)$*, any conditioning* `cond` *for which* G *is proper and any* XEU *parameter* $\varphi^{\text{XEU}} : (\Theta \to \mathbb{R}) \to 2^\Theta \to \mathbb{R}$*, the utility of the pure strategy profile* σ *for Player* i *having type* $\theta_i \in \Theta_i$*, is the integration of* $u_{i,\sigma} = \theta \mapsto u_i(\sigma^\theta, \theta)$*, i.e.,* $\text{XEU}\,\big(m(\cdot \mid_{cond} \theta_i)\big)\big(\varphi^{\text{XEU}}(u_{i,\sigma})\big)$.

```
Definition belgame_utility A T (G : belgame A T) (cond: conditioning R Tn)
  fXEU (HG : proper_belgame G cond) (p : iprofile A T) (i : player) (ti : T i) : R
  := let kn := cond G.1 (event_ti ti) (is_revisable HG ti) in
     XEU kn (fXEU (fun t => G.2 (proj_iprofile p t) t i)).
```

Also, for Bel games, the definition of Nash equilibrium applies: an `iprofile` is a Nash equilibrium iff no player, whatever is this player's type, has any incentive to deviate:

```
Definition BelG_Nash_equilibrium A T (G : belgame A T) (cond : conditioning R Tn)
    fXEU (H : proper_belgame G cond) (p : iprofile A T) :=
  ∀i : I, ∀ti : T i, ∀ai : A i,
  ~ (belgame_utility u H p ti < belgame_utility u H (change_istrategy p ti ai) ti).
```

▶ **Example 32** (Utility of a strategy). Let $\sigma = (\sigma_1, \sigma_2)$ be defined by $\sigma_1(\mathtt{P}) = Q_1$, $\sigma_1(\bar{\mathtt{P}}) = P_1$, $\sigma_2(\mathtt{R}) = Q_2, \sigma_1(\bar{\mathtt{R}}) = R_2$. σ is a pure strategy asserting that Player 1 will choose Q when learning that P is the murderer, and choose P otherwise, and that Player 2 will choose Q when learning that R is the murderer, and choose R otherwise.
Considering Dempster's conditioning, the Choquet expected utility of σ for Player 1 with type $\bar{\mathtt{P}}$ is the integration of $\varphi^{\text{CEU}}(u_{i,\sigma})$ w.r.t. the posterior bpa $m(\cdot \mid \bar{\mathtt{P}})$. Recall Example 15,

the posterior bpa "given $\bar{\mathtt{P}}$" has two focal elements: $\{Q\}$ and $\{R\}$, both with mass 1/2. Considering type configurations, those focal elements are $\{(\bar{\mathtt{P}}, \bar{\mathtt{R}})\}$ and $\{(\bar{\mathtt{P}}, \mathtt{R})\}$.

$$\begin{aligned}\mathrm{XEU}\left(m(\cdot \mid_D \bar{\mathtt{P}})\right)(\varphi^{\mathrm{CEU}}(u_{1,\sigma})) &= \sum_{B \in \mathcal{S}_{m(\cdot \mid_D \bar{\mathtt{P}})}} m(B \mid_D \bar{\mathtt{P}}) \times \min_{\theta \in B} u_1(\sigma^\theta, \theta) \\ &= 0.5 \times u_1((P_1, Q_2), (\bar{\mathtt{P}}, \mathtt{R})) + 0.5 \times u_1((P_1, R_2), (\bar{\mathtt{P}}, \bar{\mathtt{R}})) = 3.\end{aligned}$$

One may check that for every player and type, σ's CEU equals 3, the best possible score. Since no player, whatever is their type, has incentive to deviate, σ is a Nash equilibrium.

6 Howson-Rosenthal-like transforms

Howson–Rosenthal's theorem asserts the correctness of a transform which casts a 2-player Bayesian game into an equivalent polymatrix game (of complete information) [17]. Bayesian games thus benefit from both theoretical and algorithmic results of classical game theory. In the following, we formally define and prove correct three Howson-Rosenthal-like transforms that we have devised in previous work [13, 35]. We also extend the TBM transform to any conditioning (in the general case, a slight change is necessary, but for Dempster's and Strong conditioning the original statement holds, and so does the complexity of the transform). All these transforms cast n-player Bel games into hypergraphical games; the games so obtained all have the same utility values, though different hypergraphs. These transforms can be applied safely, depending on the conditioning and on the decision rule (cf. Table 4): Dempster's conditioning is hard-coded into the Direct transform while the TBM transform's low complexity comes from properties of the distribution BetP considered by the TBEU.

Table 4 Transforms, conditioning and XEU they are suited for, and their worst-case complexity w.r.t. the k-additivity of the bpa and the size of the input Bel game (taken from [35]).

Transform	Conditioning	XEU	Space	Time
Direct transform	Dempster's c.	any	$O\big(k \times \mathrm{Size}(G)^k\big)$	$O\big(k \times \mathrm{Size}(G)^k\big)$
Conditioned transform	any	any	$O\big(k \times \mathrm{Size}(G)^k\big)$	$O\big(k \times \mathrm{Size}(G)^k\big)$
TBM transform	Dempster's c.	TBEU	$O\big(k \times \mathrm{Size}(G)\big)$	$O\big(\mathrm{Size}(G)\big)$

The three transforms all follow the same approach: starting from a Bel game G, they build the equivalent hypergraphical game $\tilde{G}$, with pairs (i, θ_i) as "abstract" players (i.e., $\tilde{G}$'s vertices), denoting every type of every player of G. The local games correspond to focal elements, so Player (i, θ_i) plays in a local game `lg` iff the type θ_i is possible in the corresponding focal element. Doing so, we benefit from the hypergraphical game structure to compute an XEU (recall that global utility is the sum of local utilities and that the XEU value is the weighted sum of utilities w.r.t. focal elements). For all those transforms, let `(G : belgame A T)` be the input Bel game that has to be turned into a hypergraphical game named $\tilde{G}$. $\tilde{G}$'s players are pairs (i, θ_i), their action sets still are A_i:

```
Definition HR_player : finType := [finType of {i : I & T i}].
Definition HR_action (i_ti : HR_player) : eqType := A (projT1 i_ti).
```

Strategy profiles of G and of $\tilde{G}$ are in one-to-one correspondance. Every strategy profile `(`σ `: iprofile A T)` in G, that is, $\sigma : \prod_{i \in I}(\Theta_i \to A_i)$, is flattened to $\tilde{\sigma}$ `: cprofile (fun i_ti : {i : I & T i} => A (val i))` in $\tilde{G}$, that is, $\tilde{\sigma} : \prod_{(i,\theta_i) \in I \times \Theta_i} A_i$. E.g. in a 2-player game with 2 types per player, $\sigma = (\sigma_1, \sigma_2)$ is flattened to $(\sigma_1(\theta_1), \sigma_1(\theta'_1), \sigma_2(\theta_2), \sigma_2(\theta'_2))$. This "dependent uncurrying" is performed by the following function:

```
Definition flatten I (T : I -> finType) A (sigma : iprofile A T) :=
  [ffun i_ti => sigma (projT1 i_ti) (projT2 i_ti)].
```

6.1 The Direct Transform

The direct transform applies Dempster's conditioning on-the-fly (so this is the only possible conditioning). It is suitable for any XEU. Starting from a Bel game G, we construct a local game e_B for each prior focal element B. Vertex (i, θ_i) plays in B iff θ_i is possible in B, that is, if $\exists \theta' \in B,\ \theta_i = \theta'_i$. Its local utility in e_B is the "part of XEU" computed over B', the subset of B on which the mass shall be transferred during Dempster's conditioning.

▶ **Definition 33** (Direct transform of a Bel game). *The direct transform of a Bel game* $G = \big(I, (A_i, \Theta_i, u_i)_{i\in I}, m\big)$ *is the hypergraphical game* $\tilde{G} = \big(\tilde{I}, \tilde{E}, (\tilde{A}_{(i,\theta_i)})_{(i,\theta_i)\in\tilde{I}}, (\tilde{u}^e_{(i,\theta_i)})_{e\in\tilde{E},(i,\theta_i)\in e}\big)$*:*

- $\tilde{I} = \{(i, \theta_i) \mid i \in I, \theta_i \in \Theta_i\}$, $\tilde{E} = (e_B)_{B\subseteq\mathcal{S}_m}$, $e_B = \{(i, \theta_i) \mid \theta \in B, i \in I\}$, $\tilde{A}_{(i,\theta_i)} = A_i$,
- *for each* $e_B \in \tilde{E}$, $(i, \theta_i) \in e_B$ *and* $\tilde{\sigma} \in \tilde{A}$, *let us pose* $\tilde{v}^{\tilde{\sigma}}_i(\theta) = u_i(\tilde{\sigma}^\theta, \theta)$ *in:* $\tilde{u}^{e_B}_{(i,\theta_i)}(\tilde{\sigma}_{e_B}) = m(B) \times \big(\varphi^{\mathrm{XEU}}(\tilde{v}^{\tilde{\sigma}}_i)(B \cap \{\theta' \mid \theta'_i = \theta_i\})\big) / \mathrm{Pl}(\{\theta' \mid \theta'_i = \theta_i\})$.

Formally, let G be a proper Bel game w.r.t. Dempster's conditioning and `fXEU` a φ function:

```
Variable (proper_G : proper_belgame G (Dempster_conditioning R Tn))
         (fXEU : {ffun Tn -> R} -> {ffun {set Tn} -> R}).
```

Then, let $\tilde{G}$'s local games be indexed by focal elements, i.e., sets of type configurations:

```
Definition HRdirect_localgame := [finType of {set Tn}].
```

A vertex (i, θ_i) plays in the local game e_B iff θ_i is possible in B:

```
Definition HRdirect_plays_in (lg : HRdirect_localgame) (i_ti : HR_player) : bool
  := [∃t : Tn, [&& t \in lg & t (projT1 i_ti) == projT2 i_ti]].
```

Then, local utility functions are given by a function which constructs from a local profile p and a type configuration θ, the `cprofile` $(p_{(1,\theta_1)}, \ldots, p_{(n,\theta_n)})$. This function has type:

```
Definition HRdirect_mkprofile lg i_ti (Hi_ti : HRdirect_plays_in lg i_ti)
    (p : HRdirect_localprof lg) (t : Tn) : profile.
```

Local utility in a local game e_B is the part of the XEU computed from the prior focal element B. Note that Dempster's conditioning transfers masses from B to $B \cap \{\theta' \in \Theta \mid \theta'_i = \theta_i\}$ $= B \cap$ `(event_ti` θ_i`)` so the local utility amounts to an on-the-fly Dempster's conditioning. The resulting HG game is finally built from local utility functions:

```
Definition HRdirect_u : ∀lg, HRdirect_localprof lg -> HRdirect_localplayer lg -> R
  := fun lg p x => let (i_ti, Hi_ti) := x in let (i, ti) := i_ti in
     G.1 lg * fXEU [ffun t => G.2 (HRdirect_mkprofile Hi_ti p t) t i]
                   (lg :&: (event_ti ti)) / Pl G.1 (event_ti ti).
Definition HRdirect : cgame R HR_action := hg_game HRdirect_u.
```

▶ **Theorem 34** (Correctness of the direct transform). *For any proper Bel game* G*, Player* i *with type* θ_i*, XEU function* φ^{XEU}*, and profile* σ*, we have* $\mathrm{XEU}\big(m(\cdot \mid_D \theta_i)\big)\big(\varphi^{\mathrm{XEU}}(u_{i,\sigma})\big) = \tilde{u}_{(i,\theta_i)}(\mathtt{flatten}(\sigma))$*. Thence, Nash equilibria of* G *and* $\tilde{G}$ *are in one-to-one correspondence:*

```
Theorem HRdirect_correct (i : I) (ti : T i) (p : iprofile A T) :
  belgame_utility fXEU properG p ti = HRdirect (flatten p) (existT _ i ti).
Theorem HRdirect_eqNash (p : iprofile A T) :
  BelG_Nash_equilibrium fXEU proper_G p <-> Nash_equilibrium HRdirect (flatten p).
```

6.2 The Conditioned Transform

The conditioned transform holds for any conditioning and XEU. Starting from a Bel game G, all the conditioning "given θ_i" are pre-computed, let $\mathcal{S}^*$ be the union of all posterior focal sets (i.e., the set of all possible focal elements given any θ_i). Each $B \in \mathcal{S}^*$ leads to a local game. As in the direct transform, a vertex (i, θ_i) plays in e_B if θ_i is possible in B. Its utility in e_B is the part of XEU computed over the posterior focal element B. Note that (i, θ_i)'s local utility in B may be 0, if B is not focal in the posterior "given θ_i". Formally speaking:

▶ **Definition 35** (Conditioned transform). *The conditioned transform of a Bel game* $G = \big(I, (A_i, \Theta_i, u_i)_{i\in I}, m\big)$ *is the hypergraphical game* $\tilde{G} = \big(\tilde{I}, \tilde{E}, (\tilde{A}_{(i,\theta_i)})_{(i,\theta_i)\in\tilde{I}}, (\tilde{u}^e_{(i,\theta_i)})_{e\in\tilde{E},(i,\theta_i)\in e}\big)$*:*

- $\tilde{I} = \{(i, \theta_i) \mid i \in I, \theta_i \in \Theta_i\}$, $\tilde{E} = (e_B)_{B\in\mathcal{S}^*}$, $e_B = \{(i, \theta_i) \mid \theta \in B, i \in I\}$, $\tilde{A}_{(i,\theta_i)} = A_i$,
- $\forall e_B \in \tilde{E}$, $(i, \theta_i) \in e_B$, $\tilde{\sigma} \in \tilde{A}$, *let* $\tilde{v}^{\tilde{\sigma}}_i(\theta) = u_i(\tilde{\sigma}^\theta, \theta)$ *in* $\tilde{u}^{e_B}_{(i,\theta_i)}(\tilde{\sigma}_{e_B}) = m(B \mid \theta_i) \times f^{\text{XEU}}_{\tilde{v}^{\tilde{\sigma}}_i}(B)$.

Formally, let `fXEU` be any φ^{XEU}, `cond` be any conditioning, and G be proper w.r.t. `cond`:

```
Variables (fXEU: (Tn -> R) -> {set Tn} -> R)
          (cond : conditioning R Tn) (proper_G : proper_belgame G cond).
```

After similar definitions for `HRcond_localgame` and `HRcond_plays_in`, we define:

```
Definition HRcond_u : ∀lg, HRcond_localprof lg -> HRcond_localplayer lg -> R
  := fun lg p x ⇒ let (i_ti, Hi_ti) := x in let (i, ti) := i_ti in
     let kn := cond G.1 (event_ti ti) (is_revisable proper_G ti) in
     kn lg * fXEU [ffun t ⇒ G.2 (HRcond_mkprofile Hi_ti p t) t i] lg.
Definition HRcond : cgame R HR_action := hg_game HRcond_u.
```

▶ **Theorem 36** (Correctness of the conditioned transform). *For any proper Bel game* G*, Player* i *with type* θ_i*, conditioning* c*, XEU function* φ^{XEU}*, profile* σ*:* $\text{XEU}\big(m(\cdot \mid_c \theta_i)\big)\big(\varphi^{\text{XEU}}(u_{i,\sigma})\big) = \tilde{u}_{(i,\theta_i)}(\texttt{flatten}(\sigma))$*. Thence, Nash equilibria of* G *and* $\tilde{G}$ *are in one-to-one correspondence:*

```
Theorem HRcond_correct (i : I) (ti : T i) (p : iprofile A T):
  belgame_utility fXEU proper_G p ti = HRcond (flatten p) (existT _ i ti).
Theorem HRcond_eqNash (p : iprofile A T),
  BelG_Nash_equilibrium fXEU proper_G p <-> Nash_equilibrium (HRcond) (flatten p).
```

6.3 The TBM Transform

The TBM transform is designed for the Transferable Belief Model [39], in which knowledge is first revised using Dempster's conditioning, then decision is eventually made w.r.t. a probability distribution BetP which is deduced from the bpa m (Definition 24). Here, we benefit from BetP's 1-additivity to produce a low-complexity hypergraph: local games correspond to single states of the world $\theta \in \Theta$. In this work, we generalize the TBM transform to any conditioning, refining the statement defining local games; we show that for Dempster's and the strong conditioning, the original statement suffices, unlike for the weak conditioning.

▶ **Definition 37** (TBM transform). *Let* $G = \big(I, (A_i, \Theta_i, u_i)_{i\in I}, m\big)$ *be a Bel game; it is TBM-transformed into the hypergraphical game* $\tilde{G} = \big(\tilde{I}, \tilde{E}, (\tilde{A}_{(i,\theta_i)})_{(i,\theta_i)\in\tilde{I}}, (\tilde{u}^e_{(i,\theta_i)})_{e\in\tilde{E},(i,\theta_i)\in e}\big)$ *s.t.:*

- $\tilde{I} = \big\{(i, \theta_i) \mid i \in I, \theta_i \in \Theta_i\big\}$, $\tilde{A}_{(i,\theta_i)} = A_i$, $\tilde{E} = (e_\theta)_{\theta\in\Theta}$,
- $e_\theta = \{(i, \theta'_i) \mid \theta'_i = \theta_i \vee (\exists B \in \mathcal{S}_{m(\cdot|\theta'_i)}, \theta \in B \wedge \exists \theta'' \in B, \theta'_i = \theta''_i)\}$,
- $\tilde{u}^{e_\theta}_{(i,\theta_i)}(\tilde{\sigma}_e) = \text{BetP}_{(i,\theta_i)}(\theta) \times u_i(\sigma^\theta, \theta)$.

Formally, let `cond` be a conditioning and G be a proper Bel game w.r.t. `cond`; $\tilde{G}$'s local games are indexed by type configurations, and (i, θ_i) plays in $e_{\theta'}$ if $\theta_i = \theta'_i$ (the original statement, sufficient for Dempster's and the strong conditioning) or if there is a focal element B which contains both θ' and any θ'' such that $\theta_i = \theta''_i$ (necessary for the weak conditioning):

```
Variables (cond : conditioning R Tn) (proper_G : proper_belgame cond).
Definition HRTBM_localgame : finType := Tn.
Definition HRTBM_plays_in : HRTBM_localgame -> pred HR_player := fun lg i_ti =>
  [|| lg (projT1 i_ti) == projT2 i_ti | [∃B, [&& B \in focalset (m_ti i_ti),
    lg \in B & [∃t, (t \in B) && (t (projT1 i_ti) == projT2 i_ti)]]]].
```

Local utilities are computed w.r.t. the "pignistic" distribution `BetP`:

```
Definition HRTBM_u : ∀lg, HRTBM_localprof lg -> HRTBM_localplayer lg -> R :=
  fun lg p x => let (i_ti, _) := x in let (i, ti) := i_ti in
  let betp := BetP (cond G.1 (event_ti ti) (is_revisable proper_G ti)) in
  dist betp lg * G.2 (HRTBM_mkprofile p) lg i.
Definition HRTBM : cgame R HR_action := hg_game HRTBM_u.
```

▶ **Theorem 38** (Correctness of the TBM transform). *For any proper Bel game G, Player i with type θ_i, conditioning c, and profile σ,* $\mathrm{TBEU}\left(m(\cdot \mid_c \theta_i)\right)\left(\varphi^{\mathrm{TBEU}}(u_{i,\sigma})\right) = \tilde{u}_{(i,\theta_i)}(\mathtt{flatten}(\sigma))$. *Thence, Nash equilibria of G and $\tilde{G}$ are in one-to-one correspondence:*

```
Theorem HRTBM_correct (i : I) (ti : T i) (p : iprofile A T) :
  belgame_utility fTBEU proper_G p ti = HRTBM (flatten p) (existT _ i ti).
Theorem HRTBM_eqNash (p : iprofile A T),
  BelG_Nash_equilibrium fTBEU proper_G p <-> Nash_equilibrium HRTBM (flatten p).
```

7 Concluding remarks

In this paper, a 2.5 k LOC Coq/SSReflect formalization of Bel games has been presented. It gathers a theory for Dempster-Shafer belief functions (∼1 k LOC) as well as a generic class of games of incomplete information, built upon the former. This framework makes it possible to capture (lack of) knowledge better than usual game models based on probability. Following Howson's and Rosenthal's approach, three different transforms casting such incomplete games into standard complete-information games [35] have been formalized, one of them being further generalized. We have formally verified that these transforms preserve equilibria. Thus, Bel games are solvable using state-of-the-art, effective algorithms for complete games.

This work provides strong guaranties on the correctness of the transforms, so that game theorists may rely on them without any concern about correctness. Furthermore, the formalization allowed us to identify subtleties that were left implicit in the definitions (e.g., the conditioning pre- and post-conditions), as well as to help improving the proofs, both in their flow and in their prose. Last, generic lemmas that proved useful during our formalization effort have been proposed for integration in the MathComp library.

This work opens several research directions, both on the theoretical side and on the formal verification side. On the one hand, we aim at extending this result with other decision-theoretic approaches, e.g., partially-ordered utility aggregations for belief function and other non-additive-measure approaches (Choquet capacities of order 2, RDU). On the other hand, we would like to focus on complexity proofs which, albeit not safety-critical, play a key role when choosing one transform over the other. Eventually, we would like to encompass this work into a larger library of decision under uncertainty, fostering further developments on related models and proofs.

References

1 Reynald Affeldt and Cyril Cohen. Measure construction by extension in dependent type theory with application to integration, 2022. URL: https://arxiv.org/abs/2209.02345.

2 Reynald Affeldt, Jacques Garrigue, and Takafumi Saikawa. A library for formalization of linear error-correcting codes. *Journal of Automated Reasoning*, 64(6):1123–1164, 2020. doi:10.1007/s10817-019-09538-8.

3 Nahla Ben Amor, Hélène Fargier, Régis Sabbadin, and Meriem Trabelsi. Possibilistic Games with Incomplete Information. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, pages 1544–1550. ijcai.org, 2019.

4 Alexander Bagnall, Samuel Merten, and Gordon Stewart. A Library for Algorithmic Game Theory in SSReflect/Coq. *Journal of Formalized Reasoning*, 10(1):67–95, 2017.

5 Sylvie Boldo, François Clément, Florian Faissole, Vincent Martin, and Micaela Mayero. A Coq formalization of Lebesgue integration of nonnegative functions. *Journal of Automated Reasoning*, 66(2):175–213, 2022. doi:10.1007/s10817-021-09612-0.

6 The Coq Development Team. The Coq Proof Assistant, 2022. URL: https://doi.org/10.5281/zenodo.1003420.

7 Arthur P. Dempster. Upper and Lower Probabilities Induced by a Multivalued Mapping. *The Annals of Mathematical Statistics*, 38:325–339, 1967.

8 Christoph Dittmann. Positional determinacy of parity games. Available at https://www.isa-afp.org/browser_info/devel/AFP/Parity_Game/outline.pdf, 2016.

9 Didier Dubois and Thierry Denoeux. Conditioning in Dempster-Shafer Theory: Prediction vs. Revision. In *Belief Functions: Theory and Applications - Proceedings of the 2nd International Conference on Belief Functions*, pages 385–392. Springer, 2012.

10 Didier Dubois and Henri Prade. *Possibility Theory: An Approach to Computerized Processing of Uncertainty*. Plenum Press, 1988.

11 Didier Dubois and Henri Prade. Focusing vs. belief revision: A fundamental distinction when dealing with generic knowledge. In *Qualitative and quantitative practical reasoning*, pages 96–107. Springer, 1997.

12 Mnacho Echenim and Nicolas Peltier. The binomial pricing model in finance: A formalization in Isabelle. In *CADE*, volume 10395 of *LNCS*, pages 546–562. Springer, 2017.

13 Hélène Fargier, Érik Martin-Dorel, and Pierre Pomeret-Coquot. Games of incomplete information: A framework based on belief functions. In *Symbolic and Quantitative Approaches to Reasoning with Uncertainty - 16th European Conference*, volume 12897 of *LNCS*, pages 328–341. Springer, 2021. doi:10.1007/978-3-030-86772-0_24.

14 Ruobin Gong and Xiao-Li Meng. Judicious judgment meets unsettling updating: Dilation, sure loss and simpson's paradox. *Statistical Science*, 36(2):169–190, 2021.

15 John C Harsanyi. Games with Incomplete Information Played by "Bayesian" Players, I–III. Part I. The Basic Model. *Management Science*, 14(3):159–182, 1967.

16 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.

17 Joseph T Howson Jr and Robert W Rosenthal. Bayesian Equilibria of Finite Two-Person Games with Incomplete Information. *Management Science*, 21(3):313–315, 1974.

18 Jean-Yves Jaffray. Linear Utility Theory for Belief Functions. *Operations Research Letters*, 8(2):107–112, 1989.

19 Jean-Yves Jaffray. Linear Utility Theory and Belief Functions: a Discussion. In *Progress in decision, utility and risk theory*, pages 221–229. Springer, 1991.

20 Cezary Kaliszyk and Julian Parsert. Formal microeconomic foundations and the first welfare theorem. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018*, pages 91–101. ACM, 2018.

21 Christoph Lange, Colin Rowat, and Manfred Kerber. The formare project - formal mathematical reasoning in economics. In *MKM/Calculemus/DML*, volume 7961 of *LNCS*, pages 330–334. Springer, 2013.

22 Rida Laraki, Jérôme Renault, and Sylvain Sorin. *Mathematical foundations of game theory*. Springer, 2019.

23 Stéphane Le Roux, Érik Martin-Dorel, and Jan-Georg Smaus. An Existence Theorem of Nash Equilibrium in Coq and Isabelle. In *Proceedings Eighth International Symposium on Games, Automata, Logics and Formal Verification*, volume 256 of *Electronic Proceedings in Theoretical Computer Science*, pages 46–60, 2017. doi:10.4204/EPTCS.256.4.

24 Stéphane Le Roux, Érik Martin-Dorel, and Jan-Georg Smaus. Existence of Nash equilibria in preference priority games proven in Isabelle. In *Kurt Gödel Day and Czech Gathering of Logicians*, 2021.

25 Pierre Lescanne and Matthieu Perrinel. "backward" coinduction, Nash equilibrium and the rationality of escalation. *Acta Informatica*, 49(3):117–137, 2012.

26 Assia Mahboubi and Enrico Tassi. Mathematical Components, 2022. URL: https://doi.org/10.5281/zenodo.7118596.

27 Érik Martin-Dorel and Sergei Soloviev. A Formal Study of Boolean Games with Random Formulas as Payoff Functions. In *22nd International Conference on Types for Proofs and Programs, TYPES 2016*, volume 97 of *Leibniz International Proceedings in Informatics*, pages 14:1–14:22, 2016.

28 Érik Martin-Dorel and Enrico Tassi. SSReflect in Coq 8.10. In *The Coq Workshop 2019*, Portland State University, OR, USA, 2019. URL: https://staff.aist.go.jp/reynald.affeldt/coq2019/coqws2019-martindorel-tassi.pdf.

29 Oskar Morgenstern and John Von Neumann. *Theory of Games and Economic Behavior*. Princeton University Press, 1953.

30 Roger B Myerson. *Game Theory*. Harvard university press, 2013.

31 John Nash. Non-Cooperative Games. *Annals of Mathematics*, pages 286–295, 1951.

32 Christos H. Papadimitriou and Tim Roughgarden. Computing Correlated Equilibria in Multi-Player Games. *Journal of the Association for Computing Machinery*, 55(3):1–29, 2008.

33 Julian Parsert and Cezary Kaliszyk. Towards formal foundations for game theory. In *Interactive Theorem Proving - 9th International Conference ITP 2018*, volume 10895 of *LNCS*, pages 495–503. Springer, 2018.

34 Bernard Planchet. Credibility and Conditioning. *Journal of Theoretical Probability*, 2(3):289–299, 1989.

35 Pierre Pomeret-Coquot, Hélène Fargier, and Érik Martin-Dorel. Games of incomplete information: A framework based on belief functions. *International Journal of Approximate Reasoning*, 151:182–204, 2022. doi:10.1016/j.ijar.2022.09.010.

36 Stéphane Le Roux. Acyclic preferences and existence of sequential Nash equilibria: A formal and constructive equivalence. In *Proc. Theorem Proving in Higher Order Logics, 22nd International Conference*, volume 5674 of *LNCS*, pages 293–309. Springer, 2009. doi:10.1007/978-3-642-03359-9_21.

37 Glenn Shafer. *A Mathematical Theory of Evidence*. Princeton University Press, 1976.

38 Philippe Smets. Jeffrey's Rule of Conditioning Generalized to Belief Functions. In *Uncertainty in artificial intelligence*, pages 500–505. Elsevier, 1993.

39 Philippe Smets and Robert Kennes. The Transferable Belief Model. *Artificial Intelligence*, 66(2):191–234, 1994.

40 René Vestergaard. A constructive approach to sequential Nash equilibria. *Inormation. Processing Letters*, 97(2):46–51, 2006. doi:10.1016/j.ipl.2005.09.010.

41 Peter Walley. *Statistical Reasoning with Imprecise Probabilities*. Chapman & Hall, 1991.

42 Elena Yanovskaya. Equilibrium Points in Polymatrix Games. *Lithuanian Mathematical Journal*, 8:381–384, 1968.

Proof Repair Infrastructure for Supervised Models: Building a Large Proof Repair Dataset

Tom Reichel
University of Illinois Urbana-Champaign, IL, USA

R. Wesley Henderson
Radiance Technologies, Inc., Ruston, LA, USA

Andrew Touchet
Radiance Technologies, Inc., Ruston, LA, USA

Andrew Gardner[1]
Radiance Technologies, Inc., Ruston, LA, USA

Talia Ringer[1]
University of Illinois Urbana-Champaign, IL, USA

Abstract

We report on our efforts building a new, large proof-repair dataset and benchmark suite for the Coq proof assistant. The dataset is made up of Git commits from open-source projects with old and new versions of definitions and proofs aligned across commits. Building this dataset has been a significant undertaking, highlighting a number of challenges and gaps in existing infrastructure. We discuss these challenges and gaps, and we provide recommendations for how the proof assistant community can address them. Our hope is to make it easier to build datasets and benchmark suites so that machine-learning tools for proofs will move to target the tasks that matter most and do so equitably across proof assistants.

2012 ACM Subject Classification Computing methodologies → Machine learning; Software and its engineering → Software maintenance tools; Security and privacy → Logic and verification

Keywords and phrases proof repair, datasets, benchmarks, machine learning, formal proof

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.26

Supplementary Material
Text (Appendix): `https://dependenttyp.es/pdf/repairdataappendix.pdf`
Dataset (Sample): `https://doi.org/10.5281/zenodo.7935207`

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.

1 Introduction

Machine learning (ML) is coming for proofs. Recent years have seen a surge in interest in ML for proofs – one reflected by the many recent research venues [7, 4, 12], papers [26, 45, 37], tools [2, 13, 35], industrial research groups [47, 5], and funding opportunities [3, 6] centering on or prominently featuring ML for proofs. The surge in interest blurs the line between proofs and data so that any proof development, once released, may itself become data to improve proof automation for future proof developments.

[1] Co-senior authors

Distribution Statement A (Approved for Public Release, Distribution Unlimited).

© Tom Reichel, R. Wesley Henderson, Andrew Touchet, Andrew Gardner, and Talia Ringer;
licensed under Creative Commons License CC-BY 4.0

```
Lemma proc_rspec_crash_refines_op T (p : proc C_Op T)
  (rec : proc C_Op unit) spec (op : A_Op T) :
  (forall sA sC,
-    absr sA sC tt -> proc_rspec c_sem p rec (refine_spec spec sA)) ->
-   (forall sA sC, absr sA sC tt -> (spec sA).(pre)) ->
+    absr sA (Val sC tt) -> proc_rspec c_sem p rec (refine_spec spec sA)) ->
+   (forall sA sC, absr sA (Val sC tt) -> (spec sA).(pre)) ->
  (forall sA sC sA' v,
-    absr sA' sC tt ->
+    absr sA' (Val sC tt) ->
    (spec sA).(post) sA' v -> (op_spec a_sem op sA).(post) sA' v) ->
  (forall sA sC sA' v,
-    absr sA sC tt ->
+    absr sA (Val sC tt) ->
    (spec sA).(alternate) sA' v -> (op_spec a_sem op sA).(alternate) sA' v) ->
  crash_refines absr c_sem p rec (a_sem.(step) op)
    (a_sem.(crash_step) + (a_sem.(step) op;; a_sem.(crash_step))).
```

Figure 1 Changes made to a lemma by a participant in a recent user study of proof engineers, from the REPLica user study paper [52].

We in the proof engineering community have agency in how this surge of interest plays out. We can develop datasets and benchmark suites that steer the ML community toward the tasks that matter most. We can build infrastructure that makes it easy to develop those datasets and benchmark suites, or to work on those tasks. And we can build evaluation methodologies that measure success on those tasks in ways that truly matter, so that state-of-the-art results on benchmarks will transfer smoothly to real-world improvements in proof automation.

This paper takes a step in that direction. In particular, it presents the initial release of Proof Repair Infrastructure for Supervised Models (PRISM) – a dataset and benchmark suite for an important proof automation task in Coq: *proof repair* [49], which comprises the automatic correction of proofs in response to breaking changes in programs or specifications. Proof repair is crucially important for reducing costs in large proof developments and for enabling the application of formal methods to broader and more diverse contexts. Unfortunately, data for proof repair is scarce and challenging to collect [52]. This paper highlights the challenges involved in collecting repair data with an emphasis on how the proof engineering community can adapt to those challenges as ML becomes increasingly relevant. Its contributions are the following:

1. an initial release of a Coq proof repair dataset and benchmark suite accessible to ML experts (Section 3),
2. reusable tools for building and extracting information from Coq projects (Section 4), and
3. a discussion of the challenges we encountered and how to overcome them (Section 5).

Our overarching goal is to build the infrastructure and proof assistant community support we need to steer the ML community toward the tasks that matter most (Section 2).

2 Overview

The necessity and difficulty of proof maintenance has been borne out empirically. A recent user study of eight intermediate through expert proof engineers showed that maintenance happened constantly for participating proof engineers during proof development [52] and that even experts sometimes gave up in the face of change [49].

Consider, for example, the change in Figure 1, in which a user study participant updated a lemma statement in response to a change in a dependency. As noted in the user study paper, this was part of a larger change, with 10 other definitions or lemma statements changing in analogous ways. Furthermore, this change broke at least five proofs, four of which the user study participant – an expert proof engineer – admitted or aborted rather than repair.

The ubiquity of maintenance and the challenges of repair have been largely neglected in ML tools for proofs. ML tools for proofs have instead historically fixated on tasks like predicting tactics or automatically formalizing natural language [13, 37, 45]. The lack of a good dataset and benchmark suite obstructs progress; what currently exists is not sufficient for training and evaluating proof repair models (see Section 6). If the datasets and benchmark suites are not fit for maintenance tasks, the ML community may neglect those tasks entirely, instead chasing state-of-the-art results on tasks for which existing benchmark suites suffice.

Our experiences interacting with ML experts and building datasets ourselves suggest that the choice of datasets and benchmark suites for a domain is not driven solely by what is likely to be useful – it is also driven by barriers imposed by infrastructure, lack of domain expertise, or social factors. Things we may take for granted, like parsing and checking proofs, can become prohibitive infrastructure challenges for ML experts.

In this paper, we describe our efforts to overcome these challenges and build a large proof repair dataset for Coq. We also discuss the barriers we do encounter, and we describe both how we overcome those barriers, and what kind of work the proof assistant community would need to put in to make it so that they cease to be barriers at all.

We find this work especially prudent given that the danger of chasing benchmarks that may not transfer to real life workflows has been realized quite dramatically in other domains, from incorrect patches to programs [46] to incorrect clinical interpretation of x-ray results [67]. Furthermore, these challenges can influence not just the tasks that the ML community chooses to tackle but even the very proof assistants for which the ML community chooses to build supporting tools. It is in the community's best interest to drive strong, practical results for useful tasks in a way that is equitable across proof assistants.

Our hopes are twofold. First, we hope that our dataset will be immediately useful for proof repair. Second, and perhaps more prudently, we hope our discussion of the challenges involved in building it will serve as a call for action to improve infrastructure. The proof engineering community can then ensure that ML experts focus on the automation tasks that matter most to the community, that they measure success on those tasks in ways that transfer smoothly to real-world usefulness, and that they do so equitably across proof assistants.

3 A Proof Repair Dataset

An initial release of the PRISM dataset and benchmark suite that we have assembled is publicly available (see Supplementary Material); we will continue to update the release with later versions as we mine more data. The task that PRISM focuses on is proof repair (Section 3.1). The data comprise aligned Git commits that correspond to existing changes in proof developments found on GitHub (Section 3.2). Success on the resulting benchmark is evaluated in terms of successful proof checking for repaired proofs (Section 3.3).

3.1 The Task: Proof Repair

In ML, a *task* refers to a high-level input/output specification of what is being learned. A dataset and benchmark suite typically organizes itself around a particular task while remaining agnostic to the details of the model implementation.

We define proof repair as an ML task with **inputs** comprising an old theorem statement, proof of the old theorem, and a new theorem statement; and **outputs** comprising a proof of the new theorem. Note that this particular task assumes that we already know how to repair the theorem statement and its dependencies. We could also consider a second task that allows the model to repair the specification itself. This second repair task would be harder to evaluate, so we do not focus our benchmark suite on it at this time, even though PRISM supports it.

Inputs. We aim to provide sufficient context in the data to support a wide range of ML approaches. At a high level, the input to the ML repair model is the entire state of a project where the approach dictates how much of this state (and in what form) actually reaches the model. More precisely, the input comprises the statement of the theorem whose proof should be repaired, any contextual definitions on which it depends, the step-by-step (and subgoals) goals and hypotheses for each sentence in the old proof, and known changes to the project up to and including changes in the theorem statement and its dependencies. For each of these components, we supply raw text representations, abstract syntax trees (ASTs), and identifiers. In the case of goals and hypotheses, serialized Coq kernel representations supply detailed internal proof states. Environmental dependencies such as Coq compiler versions are captured for errors induced by external application programming interface (API) changes.

Outputs. The ultimate output is the repaired proof, which takes the form of text that may be generated one sentence at a time, all at once, or through targeted modifications of existing sentences. In the case of supervised repair learning, we supply ground truth targets in the same form as the inputs: raw text, ASTs, etc. for the entire repaired project's state (compactly represented as a "diff" relative to the input). Sufficient context is provided in the data to programmatically execute up to the error in an interactive REPL such as `coqtop` or `sertop`, where one may apply reinforcement learning akin to CoqGym [65].

3.2 The Data: Aligned Git Commits

The training and testing data comprise aligned Git commits for a selection of realistic Coq projects. The initial release of PRISM spans just a few Coq projects and consists of roughly 200 unique changes. We are working to continue to grow PRISM in the short term to span the 60 Coq projects listed in the appendix, and in the long term to reach even more projects in the long term. An initial versions of the dataset and a summary of the projects we are considering for the next version can be found in supplementary material.

Projects were originally selected by querying GitHub's API for projects that contained Coq source code, had a file called "Makefile" in the project's root directory, and had at least 100 commits from which to mine repair data. Eventually, we also included projects from CoqGym [65] and filtered to projects that were listed in OCaml Package Manager (opam) repositories (opam acts as the primary distributor of Coq projects). We excluded projects that did not contain any proofs, or that had ulterior motives in their builds (e.g., projects that intended to test the performance of the Coq compiler `coqc`). We hope in the future to include additional projects, though this will require us to support more build environments and expand upon the work detailed in Section 4.2.

Repair Examples. Within each Coq project, the data comprises a number of repair examples – that is, changes to definitions or proofs. A repair example is constructed by comparing a definition or proof before and after a change. Since sentences and files may be moved,

renamed, added, deleted, or otherwise altered between commits, they must first be *aligned* to ensure the right changes are compared. This means that Vernacular commands in one commit are assigned one-by-one to commands in another commit, where these assignments may cross file boundaries. Note that each command may not get a partner, indicating that it was either added or deleted. We describe this in more detail in Section 4.5.

After alignment, proof repair examples are constructed by partially applying changes, e.g., by omitting the changes to a proof that accompanied a change to the proposition. Thus, one pair of commits may give rise to multiple examples. The examples are compactly represented by commit hashes and diffs that indicate the state before and after a repair. This compact representation enables dissemination of the dataset without the accompanying projects, although we note supplementary tools for efficiently extracting project data will still be vitally important for eliminating redundant computations and effort in practice.

Data Split. ML datasets and benchmark suites often include a data split between training, validation, and testing data. We do not commit to a split ahead of time, but we consider two different ways of splitting data: across projects and chronologically within projects. These two splits test two different kinds of generalization beyond the training data. We plan to include defaults for both splits in the final release of PRISM.

The first split – across projects – chooses distinct sets of projects to use for training, validation, and test data. This approach measures generalization of the learned model to new projects not seen at training time. The second split – chronologically within projects – uses the same set of projects for training and test data, but splits them by time of commit, so that training data includes earlier commits, and testing data includes later commits for the same projects. This approach measures generalization of the learned model to new changes within a given project, when the model was trained on older data for that project.

3.3 The Metrics: Proof Checking

Changes in proof developments that break proofs can be fixed in two ways: by repairing the proofs themselves or by repairing some other definition such as a program or specification [49]. PRISM includes both kinds of changes. We focus our benchmark suite on the former (repairing proofs), as the metric for success is immediately clear. We hope our benchmark suite will also be useful for the latter (repairing definitions), but we believe the problem of choosing a good metric for success for repairing definitions to be an open research problem.

Repairing Proofs. We focus our benchmarks on the problem of repairing a proof script assuming that the statement of the repaired theorem is already known. In this case, checking the correctness of the repaired proof amounts to using Coq's kernel to proof check the type of the repaired proof against the type that represents the desired repaired theorem statement.

The proof checking metric is the same as that used for the standard CoqGym [65] proof generation benchmark suite for Coq. This metric is sound and complete (up to the correctness of Coq's kernel with nonterminating proof scripts designated as incorrect): any proof that checks with the desired type is a proof of the theorem the type encodes (**soundness**), and all proofs that prove that theorem will check with the desired type (**completeness**).

For this flavor of proof repair, we are able to take advantage of the fact that proof checking is a perfect oracle when the theorem statement is known. Perfect oracles have been hugely beneficial for existing ML work for proof generation [26] and for early symbolic work on proof repair when specifications do not change [53]. They continue to benefit ML for proof repair.

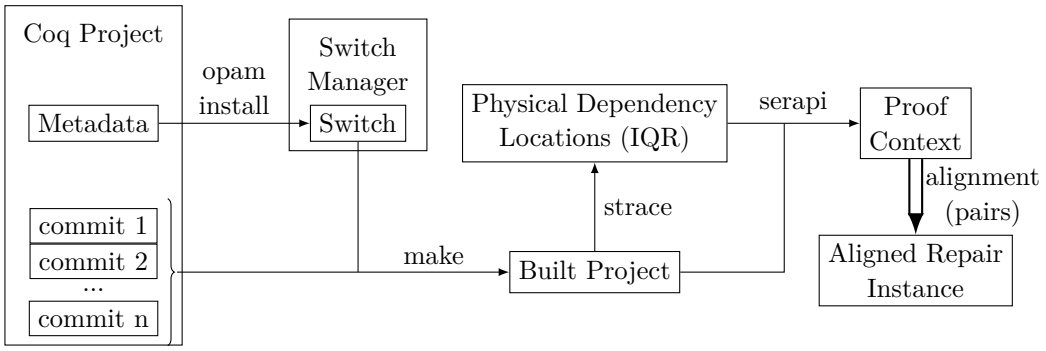


Figure 2 Process of extraction for a Coq project commit.

Repairing Definitions. Helping users fix the definitions that the specification depends on – or the theorem statement itself – is also desirable. The REPLICA user study, for example, found that 75% of the time proof engineers fixed a broken proof, they did so by fixing something else, like a program or specification [52]. Supporting this use case may actually be *more* helpful to proof engineers than supporting the original flavor of proof repair. Unfortunately, existing metrics are insufficient for measuring success on this task:

- The **proof checking** metric is insufficient when the repaired specification is unknown; showing that a proof type checks is not meaningful unless we know its intended type.
- The metric of **exact equality** with an expected repaired definition is too conservative, as there are many equivalent ways to state the same theorems or write the same definitions.
- Common notions of **definitional** or **propositional equality** in Coq are less conservative, but are still too far from complete.
- PUMPKIN PI [51] repairs definitions to be equivalent up to **univalent transport**, but checking this automatically is undecidable.
- Common natural language **distance metrics** like BLEU [42] are poor measures of success in code tasks [24], so we expect them to be inadequate for proofs.

If you choose to evaluate a model for repairing definitions, we recommend a conservative metric like exact or definitional equality to avoid the danger of chasing misleading benchmarks. We hope to eventually develop a suitable less conservative metric, particularly one that captures what makes a change "close to correct" for some suitable notion of correctness.

4 Building the Proof Repair Dataset

We now take a step back and describe the processes behind our data collection efforts. As stated, the foundation of the dataset comprises open-source Coq projects. Mining the commits of these projects eventually yields examples of refactors or repairs. Each project is accompanied by per-commit metadata containing project dependencies, source URL, and build commands that is in parts manually curated and programmatically inferred. The process of generating repair data from a project comprises the following steps (see Figure 2):

- We obtain a *switch* (opam virtual environment) that satisfies as many of the project's dependencies as possible using the **Switch Manager (SwiM)** (Section 4.1).
- Once we have a switch, we run the build command in the generated switch to produce a **Built Project** (Section 4.2).
- We `strace` the build process to scrape the **Physical Dependency Locations (IQR flags)** of each document in the project (Section 4.3).

- Using the IQR flags for each document in a built project along with the Coq serializer SerAPI [8], we extract a **Proof Context** corresponding to each intermediate proof state for the project by querying Coq's state during the execution of proofs (Section 4.4).
- Finally, we align changed proofs across commits and save those along with their intermediate proof contexts to arrive at an **Aligned Repair Instance** (Section 4.5).

Building this infrastructure was a significant undertaking with many challenges encountered along the way; we discuss these challenges in Section 5. Our hope is that the infrastructure we have built will make it easier to collect similar datasets in the future.

4.1 The Switch Manager

To extract information about projects like intermediate proof states, we must build them. This requirement is nontrivial because different projects can depend on different versions of Coq, the Ocaml compiler, or other dependencies.

To resolve dependencies and make it possible to build many different commits of one or more projects, we introduce a novel SwiM capability that works in tandem with opam to model the build environment for a given commit of a given project as a Python object. In particular, the object models an opam switch, which is opam's representation of an isolated collection of installed packages.

This capability subverts the typical manual opam workflow to create and activate a switch. This manual workflow would be intractable at the scale of hundreds of commits for each of dozens of projects. With the SwiM, we can automate this functionality and extract a dataset at scale.

Several benefits arise from the SwiM's design. The SwiM enables build sandboxing by providing switch clones that last for just the duration of the commit's extraction, and it also minimizes the time needed to obtain a clone by maintaining a pool of switches across all threads with pre-installed packages, and choosing the one upon request that is closest to satisfying a commit's requirements. Implementation of this capability required reflection of opam's dependency formula parsing and evaluation logic from OCaml to Python.

As new commits are built, switches containing their dependencies are added to the managed pool of switches. Since switches range in size from hundreds of megabytes to a few gigabytes, a least-recently-used cache maintains the total disk consumption below an implicit limit by deleting stale, infrequently used switches.

4.2 Built Projects

Using the switch provided by the SwiM and a build command from the metadata, we may be able to build the project. Confounding issues that may prevent building include undefined opam variables within dependency formulas. In practice, we have so far seen a build failure rate of about 68%. We attempt to build each commit with seven different major versions of Coq ranging from 8.9 to 8.15 corresponding to versions of SerAPI that support capabilities we deemed necessary. Since Coq releases are rarely backwards compatible, many of the build failures can be explained by the fact that each commit can only be expected to build for the single Coq version for which it was written. Furthermore, since we pin one of the seven Coq versions in the switch supplied by the SwiM, conflicting version requirements may yield an opam command that has no solution. Consequently, some build errors are inevitable.

However, other errors are due to mistakes or missing information in the human-sourced metadata. We plan to address this latter class of build errors over time by fixing problems in the metadata through automated inference mechanisms. If the project build fails, we

hope in the future to be able to recover proofs from the documents that built before the failure as well as subsequent independent proofs. We are also exploring ways to automatically recover from simple build errors such as dependency mismatches between the switch and the project's requirements by using the date the commit was made as a version hint.

4.3 Physical Dependency Locations (IQR Flags)

In order to run any of the Coq or SerAPI tools (e.g., `coqc`, `coqtop`, `sertop`) on a given Coq source file, one or more flags regularly need to be passed to these commands to specify the physical location of dependencies. These flags are described below:

- The `-I` flag allows a directory to be added to the OCaml loadpath.
- The `-Q` flag adds a physical directory to the loadpath and binds it to a given logical path.
- The `-R` flag acts like the `-Q` flag, but also makes subdirectories available recursively.

In publicly available Coq projects, these flags (referred to as "IQR" flags from here on) are specified in one or more build or configuration files. No single standardized approach for specifying IQR flags exists, making it difficult to automatically infer them from configuration and build files alone. While projects will be able to build successfully without our knowledge of these flags, we must infer them to use SerAPI tools in other stages of our framework.

Our solution to this problem builds off an approach developed in IBM's PyCoq [20]. Following PyCoq, we use `strace` to inspect the actual commands run during the build process for a Coq project. Each build command is captured and any present IQR flags are extracted using regular expressions. In some projects, build files may be nested, and IQR flags may specify physical paths that are relative to the nested directories. We need to ensure that the inferred IQR flags are relative to the project root directory, so before we store the inferred IQR flags, their paths are resolved to the project root directory.

4.4 Proof Contexts

Once the project has been built, the individual Coq source files are parsed into sentences and then interactively executed with `sertop` to capture intermediate proof states.

A parser (`sercomp`) is available through SerAPI, but it only works on Coq source files whose dependencies are already compiled, which prohibits its use in recoveries from partial builds. Furthermore, `sercomp` introduces significant redundant computation with respect to `sertop`. As a more efficient alternative, we developed a simple regular-expression-based "heuristic parser" to perform sentence extraction and approximate proof identification.

From `sertop`, we can collect thorough context from the document, like whitespace-normalized text, ASTs, command types, and intermediate proof steps with goals and hypotheses. Each command is accompanied by inferred identifiers of the command itself (e.g., an inductive type's name and constructors) and a list of fully-qualified identifiers referenced within the command, which enables models to more easily incorporate local context or apply graph-based approaches. Accompanying source code locations allow for accurate provenance of data and application of proposed repairs to appropriate destinations for testing.

4.5 Aligned Repair Instances

The last step in our data collection process is extracting proof repair examples from different versions of projects, accounting for the fact that definitions and proofs may be changed, moved, renamed, or deleted between commits. We must establish a robust mapping between Vernacular commands in a pair of commits that preserves some notion of command identity.

Our objective is similar to that of the "diff" utility, which describes changes made between files. Where our objective differs is that we seek to match Vernacular commands rather than lines, and we seek to do so within the entire project directory structure rather than a file.

A traditional order-preserving alignment between two sequences, e.g., the Smith-Waterman algorithm [58], is not quite an appropriate approach to resolve this issue as it cannot correctly align two independent definitions whose order has been reversed during a refactor (perhaps due to an introduced dependency). Therefore, we approach the problem as a bipartite matching or *assignment* between the unordered elements of two sets such that the overall similarity of matched elements is maximized. We can formally specify the desired assignment between two commits X and Y considered as respective sets of commands across one or more files as the solution to the following optimization problem:

$$\begin{aligned} &\underset{U,\,W}{\text{minimize:}} \sum_{u \in U} C(u, f(u)) \\ &\text{subject to: } f : U \leftrightarrow W \ \wedge\ U \subseteq X \ \wedge\ W \subseteq Y \ \wedge\ \|U\| = \min(\|X\|, \|Y\|) \end{aligned} \tag{1}$$

Here, $C(x, y)$ is a non-negative cost function that measures the *distance* between the commands x and y, and f is a bijection between subsets of X and Y– a partial alignment between commands of X and Y. To align as many commands as possible, the domain of f must have at least as many members of the smaller of X and Y, which is our final constraint above. This optimization is an instance of the well-known *assignment problem*, which one can solve exactly in polynomial time, e.g., with the Hungarian algorithm [41].

The optimization is parameterized by the choice of C, for which we choose a normalized variant of the Levenshtein edit distance E:

$$C(x, y) = \frac{2E(x, y)}{\|x\| + \|y\| + E(x, y)}, \tag{2}$$

where $\|x\|$ and $\|y\|$ give the character lengths of x and y considered as text (not including proof bodies). This normalization is an instance of the biotope transform [22], which preserves the metric properties (such as the triangle inequality) of the edit distance. We further threshold the distance by a constant t such that $C_t(x, t) = \min\{C(x, y), t\}$, which also preserves metric properties [43]. After solving for f, commands x and y assigned to one another ($f(x) = y$) with a cost of t are considered to be unassigned (i.e., we determine that x was dropped between commits and y was added). We choose $t = 0.4$, which roughly corresponds to 50% of a command's text being changed before it is considered to have been dropped.

Solving this assignment problem for two entire commits can be costly: solving exactly is cubic complexity, and calculating the edit distance between all pairs of commands from both commits is necessarily quadratic complexity. Furthermore, the assignments produced may be somewhat spurious, especially in the event of multiple global optima. We mitigate these issues by applying the assignment problem only to those commands known to have changed in some manner between the commits according to their intersection with a (Git) "diff". The final resolution of the problem is thus somewhere in between alignment and assignment.

Once we determine an alignment, we create examples of proof errors by leaving out changes to individual proofs one at a time, thus providing the context for each change to a proof that required repair but not the repair itself. The left-out change to the proof then accompanies the error as a ground truth target for supervised learning.

5 Challenges

Collecting and building datasets and benchmark suites for many tasks is still extremely challenging, and it is challenging in a way that is not at all equitable across proof assistants. Here, we discuss our experiences dealing with challenges encountered during the creation of PRISM (Section 5.1), what we believe the Coq community can learn from other proof assistant communities (Section 5.2), and how the proof assistant community at large could address them more sustainably going forward (Section 5.3).

5.1 Our Experiences

The major challenges we faced in building this dataset and benchmark suite chiefly fall into two categories: Project Management (Section 5.1.1) and Parsing & Serialization (Section 5.1.2). For each of these categories, we discuss our experiences dealing with each stated challenge.

5.1.1 Project Management

One of the greatest barriers to building this dataset was the lack of a centralized archive for Coq proof data. In the absence of this centralized archive, we resorted to looser collections of projects organized by package management. The package manager opam gives us a programmatic interface to build compatible environments for the dataset's constituent projects. However, it was designed to service individual developers using a few switches, whereas we must spin up *dozens* of switches efficiently. We thus had to reimplement and expand upon some of opam's capabilities. We faced three challenges in so doing:

1. Significant **build system variation** across different proof developments;
2. **Expressive dependencies** in opam packages that complicate efficient installs;
3. Insufficient caching of opam build artifacts that necessitated **copying switches** to avoid rebuilding the same packages.

Build System Variation. Over the years, the recommended build system for Coq proof developments has been in flux. In 2019, for example, the Coq development team urged proof engineers to move their proof developments to Dune [19]. This effort did not fully succeed, and the documentation for the latest Coq version includes instructions for both Dune and the native Coq build system [19]. The native build system itself has also changed over time, losing compatibility with its previous versions. Because of this fragmented build infrastructure, we had to employ extremely abstract methods to extract arguments for SerAPI tools, namely by using `strace` to grab IQR flags passed to Coq's compiler `coqc` (described in Section 4.3) while making almost no assumptions about the process invoking `coqc`.

Expressive Dependencies. The opam package manager provides a powerful and expressive syntax (package formulae) for packages to specify dependencies. Package formulae allow developers to restrict the versions of dependencies that can be installed, to conjunct and disjunct formulae into more complicated expressions, and to refer to variables declared elsewhere in the environment. This feature benefits the library developer that can precisely specify the environment for running code, but for our purposes it poses a challenge: packages can be picky about their environments and force opam to rebuild existing libraries. Since we need to install many versions of many packages, we need efficient ways to create or select compatible switches, which means interpreting these formulae. As a result, we reimplemented a majority of opam's package formula features, including parsing the custom grammar for package formulae and implementing package version comparison, to reason about which existing switches would require the least time to install a given package with opam.

Listing 1 A notation that breaks `CoqIDE`'s parser. This example was found in the Coq Discourse [17].

```
Notation "( a . b )" := (a, b).
Check (1 . 2).
```

Copying Switches. To work with conflicting packages or different versions of the same packages, we must use different environments. The opam "switch" abstraction allows us to sandbox environments, but creating many switches incurs exorbitant overhead as each new switch rebuilds packages from source. Building a package *once* and deploying it in multiple switches is preferable, but many executables built by opam contain their absolute path as a hardcoded variable, which means they stop working if the name or location of the containing switch changes. That is, a built opam package only necessarily works in one switch. Our workaround is to copy switches and use the `bwrap` utility (which is also used internally in opam) to bind-mount the copied switch over the original such that the clone is in the original hardcoded location from the perspective of the running process. This solution allows copies of switches to act as if they are the original. Of course, handling these cloned switches requires extra bookkeeping and infrastructure, which the SwiM (Section 4.1) ultimately handles.

5.1.2 Parsing & Serialization

No matter how sophisticated the build system, we cannot get detailed data about individual proofs without parsing Coq files and serializing proof state to text. SerAPI [8] is the de facto standard for serializing Coq, providing a query protocol for exposing internal Coq data like definitions in the global environment, syntax trees, goals, types, and more. We used the CoqGym [65] Python wrapper as a starting point for our implementation, taking care to decouple it from CoqGym's custom versions of Coq and SerAPI since we need to support multiple versions of each coinciding with chosen projects' Git histories. This need to support multiple versions of Coq exacerbated challenges arising from gaps in SerAPI's query protocol, requiring us to implement workarounds using the most public and arguably stable interface Coq possesses: its Vernacular query commands. We faced four challenges related to parsing & serialization:

1. Executing a file one Coq sentence at a time requires accurately **parsing sentence boundaries**, but parsing requires execution: a catch-22.
2. **Identifying dependencies between commands** (e.g., which lemmas a theorem uses) is critical to providing locally relevant repairs but is not a capability provided by SerAPI.
3. **Determining the scope of a conjecture** is complicated by the potential presence of nested proofs/definitions and arbitrary grammar extensions.
4. SerAPI is experimental software, which leads to breaking **changes between versions**.

Parsing Sentence Boundaries. A Coq statement or "sentence" ends with a period (.), but Coq also uses the symbol for import paths and module members so that one cannot identify sentences in a file merely by splitting on periods. To further complicate matters, Coq boasts an extensible syntax that enables users to define syntax that allows periods to show up in even more situations. For example, Listing 1 defines syntax using a period that complicates sentence splitting to the point where the latest version of `CoqIDE` – the official editor for Coq – cannot correctly parse and run this code even though Coq can. We did not discover any public or officially supported mechanism to extract the sentences of a Coq document, which led us to develop the Python-based heuristic parser mentioned in Section 4.3 for simplicity and maximal portability between build environments.

Listing 2 A simple example showing that proofs may be interleaved and that multiple proofs (obligations) may be associated with one term.

```
Require Coq.Program.Tactics.
Set Nested Proofs Allowed.
Program Definition foo := let x := _ : unit in _ : x = tt.
Next Obligation. (* Start first obligation of foo *)
  Definition foobar : unit. (* Interject with new conjecture. *)
    exact tt.
    Next Obligation. (* Switch back to first obligation of foo *)
      exact tt.
    Qed. (* Finish proof of foo's first obligation *)
  Defined. (* Finish proof of foobar *)
Next Obligation. (* Start next obligation of foo *)
  simpl; match goal with ⊢ ?a = _ ⇒ now destruct a end.
Qed. (* foo is defined *)
```

Identifying Command Dependencies. Identifying dependencies decomposes into two subproblems: detecting the definitions (if any) introduced by a given command and resolving referenced names unambiguously.

No SerAPI query resolves the first subproblem, nor is there any reliable syntactic clue in the text that generalizes across unforeseen grammar extensions. Instead, we rely upon parsing user-level feedback that notes the introduction of new identifiers (e.g., "X is defined") and Vernacular queries. Since feedback is not guaranteed for all definition types (particularly propositions, depending on the Coq version), we also monitor for changes in the set of all locally defined names yielded from Vernacular `Print All` command. One can thus reliably identify a command with names introduced immediately after its execution.

The second subproblem arises from the fact that identifiers within ASTs yielded from SerAPI are not necessarily fully qualified. Correcting this deficiency requires locating the identifiers within the AST and issuing a Vernacular `Locate` query for each one. Care must be taken to ensure that variables within local binders, patterns, or other sub-expressions do not get mistaken as any top-level definition that they may shadow. Given the lack of insight available into Coq's internal name resolution, the accuracy is ultimately limited by handcrafted scope rules. We also note one restriction on resolving globally bound identifiers: if a definition shadows an existing one, then it cannot also use the shadowed one. Violation of this assumption is possible (consider a recursive function `nat` that expects arguments of type `nat`) but not expected to pose a significant risk as it is unlikely in the first place and would generally be considered poor practice. If the restriction is violated, then the shadowed definition will simply be mistaken for its shadower within the shadower's definition.

Determining Conjecture Scope. Determining conjecture scope decomposes into two subproblems: attribution of proof steps to the correct conjecture and detection of proof (conjecture) completion. Each is complicated by potentially intermingled or nested proof steps as shown in Listing 2 and by the lack of a SerAPI query of the active conjecture's identity.

A Vernacular command – `Show Conjectures` – again provides the solution. This command lists the names of currently stated but unproved conjectures and by all observations is guaranteed to list the conjecture actively being proved first. We rely upon this presumed order to identify the current conjecture, accumulating proof steps in stacks per open conjecture. The method's accuracy depends upon the assumption that each conjecture enters proof mode once its first sentence is executed. The only known exceptions to this rule comprise Programs, which do not enter proof mode until their first Obligation's proof is begun.

Special handling is required to associate each Obligation with the correct Program since `Show Conjectures` reveals a unique name for each Obligation. However, the special handling means any grammar extension that defines its own Obligation or Program equivalents (e.g., multi-block proofs) cannot be serialized to the same level of accuracy. If any extension does so, then each Obligation-equivalent is expected to be serialized as an unrelated theorem.

We rely upon detection of definitions to determine when and if a conjecture was proved, assuming that no conjecture emits an identifier before it is defined (i.e., before it is proved). Only subproofs (generally delimited by bullets and braces) are allowed to violate this rule. However, one cannot assume that the first detected definition in the midst of a proof corresponds to the conjecture, nor can one assume that the name of the conjecture once defined will actually match its name as returned by `Show Conjectures`.

We ultimately detect the completion of a proof by requiring two conditions: a change in the currently detected conjecture and the detection of a new definition. This rule necessarily invokes an additional assumption: a change in the current conjecture implies that either a new proof has begun or the current proof has ended (but not both). Since we assume that conjectures cannot emit identifiers before they are done, we deduce that the emission of an identifier upon the change of the current conjecture implies the completion of the prior one.

Finally, if the conjecture is aborted, then it will never be detected as a definition at all even though its proof has ended. We detect aborted proofs simply by checking the type of the command, assuming that no grammar extension defines `Abort` or `Abort All` equivalents.

Serialization and Version Changes. SerAPI was in theory supposed to help with proof assistant versioning problems. In practice, though, SerAPI itself depends on the version of Coq, and we found we had to break the SerAPI abstraction barrier often as the Coq version changed. In other words, while SerAPI provides a convenient interface to expose certain Coq internals, those internals are not necessarily stable. For example, SerAPI had "can't-fix" bugs involving nested proofs because the serialization errors occur in the Coq codebase itself [28]. SerAPI itself has as of a few days ago been deprecated in favor of a new serializer [29, 18].

5.2 Other Proof Assistants

Here, we discuss features in other proof assistants in the context of our experiences above.

Project Management. In summary, we are not aware of an elegant and effective solution to package management for other proof assistants, but we believe Isabelle's rich archival culture sets a good example to follow. In Isabelle, the Archive of Formal Proofs (AFP) provides a highly centralized, standard host for proof developments and eases their association with metadata that may be useful for ML. The AFP also neatly versions proof developments for every official release of Isabelle and semantically groups them in different folders. At the time of writing, the AFP includes 725 proof developments [1], and it already forms the basis of a static dataset for Isabelle [32]. We suspect the AFP would also make a very strong basis for a proof repair dataset due to its neat versioning.

Agda possesses its own library management system, which it uses in combination with Hackage, the Haskell package repository. Anecdotally, researchers we have spoken to cite installation difficulties as a barrier to learning Agda. Lean also has its own package manager but lacks advanced features to address the problems we faced. Isabelle in general takes an IDE-centric approach to builds and other tooling [50, 60], but does include a notion of sessions that can inherit from other sessions. The underlying functionality the IDE is based on is also accessible in Scala and by command line. However, one of the authors has found that students learning Isabelle/HOL in a proof automation course struggle to understand how to build dependencies.

Parsing & Serialization. A particularly successful example of an interoperable proof system is MetaMath [39], whose syntax and semantics are so simple that its verifier has been reimplemented in under 1000 lines of Python [61]. This and its centralized proof database has made it a popular choice for ML experts as a benchmark for ML applications in theorem proving [37, 45] as all barriers to serialization can be avoided by modifying a very small parser/verifier. As a trade off, MetaMath does not have a comparable feature set to ITPs like Coq, Lean or Isabelle.

More complicated and featureful ITPs have more varied methods: PISA [33] is a bleeding edge Isabelle interaction and proof serialization tool written to support an ML experiment [32]. This complements `scala-isabelle` [57], an earlier, less ML-oriented tool which is also actively maintained. As for Lean, PACT [31] presented a dataset (LeanStep) aimed at ML applications that uses Lean's meta-programming facilities to serialize Lean. LeanStep's tools weigh under 1500 lines of code, which is light compared to line-counts for other serialization efforts.

5.3 Recommendations

Project Management. A strong archival effort is the best way forward, though even the best archival infrastructures for proof assistants fall short in multiple ways. For example, while Isabelle's AFP makes a natural data source for ML tools for proofs, it does not include any processes for informed consent – the datasets that build on it assume that all publicly available data is fair game. While this assumption is standard in ML for programs and proofs, it is not ideal; archival is a natural place to consider it.

In addition, though archival makes it possible to associate metadata with proof developments, little consideration is given to metadata that associates definitions and proofs *across versions* of a proof development. Presently, we rely on package management tools such as opam, which also posed challenges. Though a legitimate argument can be made that package management targets a very different use case from ours and that existing tools are sufficient for that use case, shared high-level libraries and tools on top of existing package managers and in support of bulk efforts like our own would be especially advantageous since such efforts are common when building ML datasets and tools. Nonetheless, package managers themselves warrant some improvement. For example, the problem we encountered of copying switches was due to poor caching of build dependencies, which itself was due to some degree to hard-coding of paths.

Parsing & Serialization. The great flexibility afforded Coq by its extensible grammar allows documents to be more human-friendly and readable, but the lack of syntactic assurances introduces major headaches for automated systems. Ideally, future languages will be structured to be machine-readable human/compiler-out-of-the-loop or to at least provide a public parsing API. For Coq, exposing the classification of a Vernacular command[1] in SerAPI would help substantially and obviate the need for many of the workarounds detailed above.

We also recommend a greater emphasis on backwards compatibility and backporting as several useful and even critical features that exist in newer versions of Coq or SerAPI were not suitable for our use. To this end, SerAPI is "still a research, experimental project, and it is expected to evolve considerably" [27] For instance, future plans rebase SerAPI on the language server protocol standard [29], which exposes features like document overviews that appear to list all the definitions in the file and the ability to fold proofs, implying that it has the capability to list theorems and gather the associated lines – one of our current challenges.

[1] See the `vernac_classification` type.

Overall. Based on our observations, we make the following broad recommendations for the proof assistant community going forward:

1. Work with the Isabelle community to learn how to build **centralized archives** as successful as the AFP for other proof assistants.
2. Include an **informed consent form** in any centralized archive that allows proof engineers to opt in or out of their developments being used for ML tools.
3. Determine what **kinds of metadata** within and across proof developments would ease the creation of ML tools for the tasks that matter most, make it easy to **track that metadata** inside of any centralized archive, and create standard ways of **associating that metadata** with proof developments even outside of centralized archives.
4. Establish or adopt **open standards** for managing proof developments and interfacing with external tools like modern IDEs.
5. Develop tools based on these standards to enable **extraction of metadata** relevant to ML tools from non-archival proof developments.
6. Help proof engineers **port legacy proof developments** to meet those standards, and continue to work on tools for **proof repair and reuse** that ease this burden.
7. Consider building **shared libraries and tools** optimized for the problems of bulk builds.
8. Consider limiting the scope of possible **dependency complexity**.
9. Improve **build caching** across multiple or bulk builds, for example by avoiding hard-coded paths seen in opam.
10. Consider opening conversations with **language developers and companies** inside and outside of verification about their solutions for package management, distribution, and release management, as these problems are pervasive across all software.

6 Related Work

Datasets & Benchmark Suites. The REPLica [52] user study collected incremental edit data from eight proof engineers over the course of a month. Due to difficulties recruiting participants, the dataset is too small for data-hungry ML tools. PRISM is less incremental, but we expect the final version of the dataset to be much larger. The REPLica data may make a useful supplement to PRISM.

A number of datasets and benchmark suites target *autoformalization*: the automatic translation of natural language mathematics to formal mathematics. Autoformalization datasets consisting of aligned natural and formal language include ProofNet [9] and the Isabelle Parallel Corpus [15]. MiniF2F [68] includes math Olympiad problems formalized in different proof assistants and is used as a benchmark for autoformalization and synthesis.

A few datasets and benchmark suites exist for proof synthesis, including CoqGym [65] for Coq and HOList [11] for HOL Light. These datasets include static data from fixed project versions. The distinguishing feature of PRISM is that it describes the project's history, which is necessary to produce repair examples.

We expect there is much that we can learn from ML for code, given the similarities between code and proofs. A summary of recent work in this space can be found in a survey paper on neurosymbolic programming [16]. Of particular interest for our work is the question of whether code distance metrics like CodeBLEU [48] will work well for formal proof.

In the field of software engineering, accessible datasets facilitate new research. For example, Defects4 [34] is a collection of bugs and patches in Java that is frequently used as a benchmark for program repair [23, 59, 38]. We hope that PRISM will spur new research in proof repair. We also hope that, by focusing on good benchmarks and metrics for success early on, we can avoid some of the methodology challenges faced in program repair [46].

Proof Repair. The ML task that our dataset focuses on is proof repair, which is summarized in the namesake thesis [49]. There is not yet published work we are aware of for ML for proof repair, though we are aware of ongoing work by other teams in proof assistants other than Coq. We plan to train and evaluate at least two distinct proof repair models in Coq using PRISM, and we hope that PRISM makes it easy for others to do the same.

Proof repair is closely related to work in proof reuse [25, 54, 14], proof refactoring [63, 62, 55], and proof transformation [44]. These and other related topics in proof engineering have a long history, described in detail in the proof engineering survey paper QED at Large [50], as well as in the proof repair namesake thesis [49].

Proof repair can be viewed as program repair [40, 30] for proofs. There is a large amount of work on learning to repair programs, both symbolically (for example, in Getafix [10]) and neurally (for example, in Break-It-Fix-It [66]). This work may provide useful insights when building ML datasets, benchmark suites, and models for proof repair, though care must be taken to consider the differences between typical programs and formal proof developments [49].

Machine Learning for Proofs. Advances in ML have had a transformative effect on many fields, and theorem provers are not excluded. Examples of recent work on ML for synthesizing formal proofs include GPT-f [45] and HTPS [36] for Metamath and Lean; Proverbot9001 [56], ASTactic [65], Tactician [13], and DIVA [26] for Coq; and DeepHOL [11] for HOL Light. Also of note is recent work on autoformalization in Isabelle/HOL [64], Lean [9], and Coq [21]. More ML work for proofs can be found in QED at Large [50]. Our main goal is to expand the scope of tasks covered in ML for proofs, reaching important tasks not previously explored.

7 Conclusions & Future Work

We have described the initial version of a novel dataset and benchmark suite for the ML task of proof repair centered around the Coq Proof Assistant. We expect later versions of the data to be significantly larger than any existing alternative, spanning years-long developments collected from a corpus of open-source Github repositories. We discussed challenges that we encountered during the creation of the dataset and ramifications for the proof engineering community going forward. The tools developed to overcome these challenges enable subsequent expansion of the dataset with supplementary Coq projects and are likely to be useful for creating and interfacing with datasets for other proof-related ML tasks including, for example, proof synthesis.

Moving forward, our immediate plans are to continue to grow the dataset, and to release the infrastructure we built for more general use. We also plan to use the dataset to build ML models for proof repair in Coq. We would also like to develop better metrics for measuring success at repairing definitions. Finally, we hope to work with the rest of the proof assistant community to address the many challenges we have highlighted, so that we may steer ML for proofs in the right direction.

References

1 Statistics - archive of formal proofs. URL: `https://www.isa-afp.org/statistics/`.

2 Arpan Agrawal, Emily First, Zhanna Kaufman, Tom Reichel, Shizhuo Zhang, Timothy Zhou, Alex Sanchez-Stern, and Talia Ringer. Proofster. URL: `https://www.alexsanchezstern.com/papers/proofster.pdf`.

3 Europroofnet. URL: `https://europroofnet.github.io/`.

4 2nd MATH-AI Workshop at NeurIPS'22, 2021-2022. URL: `https://mathai2022.github.io/`.

5 Openai. URL: https://openai.com/.

6 Proof engineering, adaptation, repair, and learning for software (pearls). URL: https://sam.gov/opp/da84366306554cc981f37f703a78c698/view.

7 AI for Theorem Proving, 2016-2022. URL: http://aitp-conference.org/.

8 Emilio Jesús Gallego Arias. SerAPI: Machine-Friendly, Data-Centric Serialization for COQ. Technical Report hal-01384408, HAL, 2016. URL: http://dml.mathdoc.fr/item/hal-01384408/.

9 Zhangir Azerbayev, Bartosz Piotrowski, and Jeremy Avigad. ProofNet: A benchmark for autoformalizing and formally proving undergraduate-level mathematics problems. In *Second MATH-AI Workshop*, 2022. URL: https://mathai2022.github.io/.

10 Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to fix bugs automatically. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi:10.1145/3360585.

11 Kshitij Bansal, Sarah M Loos, Markus N Rabe, Christian Szegedy, and Stewart Wilcox. Holist: An environment for machine learning of higher order logic theorem proving. In *ICML*, 2019.

12 Beyond Bayes: Paths Towards Universal Reasoning Systems, 2022. URL: https://beyond-bayes.github.io/.

13 Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. The tactician: A seamless, interactive tactic learner and prover for coq. In *Intelligent Computer Mathematics: 13th International Conference, CICM 2020, Bertinoro, Italy, July 26–31, 2020, Proceedings*, pages 271–277, Berlin, Heidelberg, 2020. Springer-Verlag. doi:10.1007/978-3-030-53518-6_17.

14 Olivier Boite. Proof reuse with extended inductive types. In *Theorem Proving in Higher Order Logics: 17th International Conference, TPHOLs 2004, Park City, Utah, USA, September 14-17, 2004. Proceedings*, pages 50–65, Berlin, Heidelberg, 2004. Springer. doi:10.1007/978-3-540-30142-4_4.

15 Anthony Bordg, Yiannos A Stathopoulos, and Lawrence C Paulson. A parallel corpus of natural language and isabelle artefacts. In *7th Conference on Artificial Intelligence and Theorem Proving (AITP)*, 2022. URL: http://aitp-conference.org/2022/abstract/AITP_2022_paper_8.pdf.

16 Swarat Chaudhuri, Kevin Ellis, Oleksandr Polozov, Rishabh Singh, Armando Solar-Lezama, Yisong Yue, et al. Neurosymbolic programming. *Foundations and Trends® in Programming Languages*, 7(3):158–243, 2021.

17 Is there a full documentation of coq's grammar? URL: https://coq.discourse.group/t/is-there-a-full-documentation-of-coqs-grammar/647/10.

18 coq_lsp. URL: https://github.com/ejgallego/coq-lsp.

19 Proposal: a custom build tool for coq projects. URL: https://coq.discourse.group/t/proposal-a-custom-build-tool-for-coq-projects/239/2.

20 pycoq. URL: https://github.com/IBM/pycoq.

21 Garett Cunningham, Razvan C. Bunescu, and David Juedes. Towards autoformalization of mathematics and code correctness: Experiments with elementary proofs, 2023. doi:10.48550/ARXIV.2301.02195.

22 Michel Marie Deza and Elena Deza. *Encyclopedia of Distances*. Springer Berlin Heidelberg, Berlin, 3 edition, October 2014. URL: https://link.springer.com/book/10.1007/978-3-642-30958-8.

23 Thomas Durieux, Matias Martinez, Martin Monperrus, Romain Sommerard, and Jifeng Xuan. Automatic repair of real bugs: An experience report on the defects4j dataset. *CoRR*, abs/1505.07002, 2015. arXiv:1505.07002.

24 Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. Out of the bleu: how should we assess quality of the code generation models? *arXiv preprint arXiv:2208.03133*, 2022.

25 Amy Felty and Douglas Howe. Generalization and reuse of tactic proofs. In *Logic Programming and Automated Reasoning: 5th International Conference*, LPAR '94, pages 1–15, Berlin, Heidelberg, 1994. Springer. doi:10.1007/3-540-58216-9_25.

26 Emily First and Yuriy Brun. Diversity-driven automated formal verification. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)(22–27). Pittsburgh, PA, USA. https://doi. org/10.1145/3510003.3510138*, 2022.

27 General roadmap. URL: https://github.com/ejgallego/coq-serapi/issues/252.

28 Query ast returns empty result. URL: https://github.com/ejgallego/coq-serapi/issues/117.

29 Serapi 'classic mode' final release notice. URL: https://github.com/ejgallego/coq-serapi/issues/252#issuecomment-1365510329.

30 Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 1219, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3180155.3182526.

31 Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W. Ayers, and Stanislas Polu. Proof artifact co-training for theorem proving with language models. *CoRR*, abs/2102.06203, 2021. arXiv:2102.06203.

32 Albert Jiang, Wenda Li, Jesse Han, and Wu Yuhuai. Lisa: Language models of isabelle proofs. In *6th Conference on Artificial Intelligence and Theorem Proving (AITP)*, 2021.

33 Portal-to-isabelle. URL: https://github.com/albertqjiang/Portal-to-ISAbelle.

34 René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, San Jose, CA, USA, July 2014. Tool demo.

35 Daniel Kühlwein, Jasmin Christian Blanchette, Cezary Kaliszyk, and Josef Urban. Mash: Machine learning for sledgehammer. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, pages 35–50, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

36 Guillaume Lample, Marie-Anne Lachaux, Thibaut Lavril, Xavier Martinet, Amaury Hayat, Gabriel Ebner, Aurélien Rodriguez, and Timothée Lacroix. Hypertree proof search for neural theorem proving. *arXiv preprint arXiv:2205.11491*, 2022.

37 Guillaume Lample, Marie-Anne Lachaux, Thibaut Lavril, Xavier Martinet, Amaury Hayat, Gabriel Ebner, Aurélien Rodriguez, and Timothée Lacroix. Hypertree proof search for neural theorem proving, 2022. doi:10.48550/ARXIV.2205.11491.

38 Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. Coconut: Combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, pages 101–114, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3395363.3397369.

39 Norman D. Megill and David A. Wheeler. *Metamath: A Computer Language for Mathematical Proofs*. Lulu Press, Morrisville, North Carolina, 2019. http://us.metamath.org/downloads/metamath.pdf.

40 Martin Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1), January 2018. doi:10.1145/3105906.

41 James Munkres. Algorithms for the Assignment and Transportation Problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957. Publisher: Society for Industrial and Applied Mathematics. URL: https://www.jstor.org/stable/2098689.

42 Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, pages 311–318, USA, 2002. Association for Computational Linguistics. doi:10.3115/1073083.1073135.

43 Ofir Pele and Michael Werman. Fast and robust earth mover's distances. In *2009 IEEE 12th International Conference on Computer Vision*, pages 460–467, 2009. doi:10.1109/ICCV.2009.5459199.

44 Frank Pfenning. *Proof Transformations in Higher-Order Logic*. PhD thesis, Carnegie Mellon University Pittsburgh, 1987.

45 Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *CoRR*, abs/2009.03393, 2020. arXiv:2009.03393.

46 Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 24–36, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2771783.2771791.

47 Markus N. Rabe and Christian Szegedy. Towards the automatic mathematician. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 25–37, Cham, 2021. Springer International Publishing.

48 Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *CoRR*, abs/2009.10297, 2020. arXiv:2009.10297.

49 Talia Ringer. *Proof Repair*. PhD thesis, University of Washington, 2021.

50 Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. QED at large: A survey of engineering of formally verified software. *CoRR*, abs/2003.06458, 2020. arXiv:2003.06458.

51 Talia Ringer, RanDair Porter, Nathaniel Yazdani, John Leo, and Dan Grossman. Proof repair across type equivalences. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 112–127, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454033.

52 Talia Ringer, Alex Sanchez-Stern, Dan Grossman, and Sorin Lerner. REPLica: REPL instrumentation for Coq analysis. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 99–113, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372885.3373823.

53 Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Adapting proof automation to adapt proofs. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 115–129, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3167094.

54 Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Ornaments for proof reuse in coq. In *Interactive Theorem Proving*, 2019.

55 Valentin Robert. *Front-end tooling for building and maintaining dependently-typed functional programs*. PhD thesis, UC San Diego, 2018.

56 Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. Generating correctness proofs with neural networks, 2019. doi:10.48550/ARXIV.1907.07794.

57 scala-isabelle. URL: https://dominique-unruh.github.io/scala-isabelle/.

58 T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981. doi:10.1016/0022-2836(81)90087-5.

59 Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 1–11, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3180155.3180233.

60 Makarius Wenzel. Isabelle/jedit — a prover IDE within the PIDE framework. *CoRR*, abs/1207.3441, 2012. arXiv:1207.3441.

61 mmverify.py. URL: https://github.com/david-a-wheeler/mmverify.py.

62 Iain Johnston Whiteside. *Refactoring proofs*. PhD thesis, University of Edinburgh, November 2013. URL: http://hdl.handle.net/1842/7970.

63 Karin Wibergh. Automatic refactoring for agda. Master's thesis, Chalmers University of Technology and University of Gothenburg, 2019.

64 Yuhuai Wu, Albert Q Jiang, Wenda Li, Markus N Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. *arXiv preprint arXiv:2205.12615*, 2022.

65 Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. In *International Conference on Machine Learning (ICML)*, Long Beach, CA, USA, 2019. URL: `http://proceedings.mlr.press/v97/yang19a/yang19a.pdf`.

66 Michihiro Yasunaga and Percy Liang. Break-it-fix-it: Unsupervised learning for program repair. In *International Conference on Machine Learning*, pages 11941–11952. PMLR, 2021.

67 John R. Zech, Marcus A. Badgeley, Manway Liu, Anthony B. Costa, Joseph J. Titano, and Eric Karl Oermann. Variable generalization performance of a deep learning model to detect pneumonia in chest radiographs: A cross-sectional study. *PLOS Medicine*, 15(11):e1002683, November 2018. `doi:10.1371/journal.pmed.1002683`.

68 Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. minif2f: a cross-system benchmark for formal olympiad-level mathematics. In *International Conference on Learning Representations*, 2022. URL: `https://openreview.net/forum?id=9ZPegFuFTFv`.

POSIX Lexing with Bitcoded Derivatives

Chengsong Tan
Imperial College London, UK

Christian Urban
King's College London, UK

Abstract

Sulzmann and Lu describe a lexing algorithm that calculates Brzozowski derivatives using bitcodes annotated to regular expressions. Their algorithm generates POSIX values which encode the information of *how* a regular expression matches a string – that is, which part of the string is matched by which part of the regular expression. This information is needed in the context of lexing in order to extract and to classify tokens. The purpose of the bitcodes is to generate POSIX values incrementally while derivatives are calculated. They also help with designing an "aggressive" simplification function that keeps the size of derivatives finitely bounded. Without simplification the size of some derivatives can grow arbitrarily big, resulting in an extremely slow lexing algorithm. In this paper we describe a variant of Sulzmann and Lu's algorithm: Our variant is a recursive functional program, whereas Sulzmann and Lu's version involves a fixpoint construction. We *(i)* prove in Isabelle/HOL that our variant is correct and generates unique POSIX values (no such proof has been given for the original algorithm by Sulzmann and Lu); we also *(ii)* establish finite bounds for the size of our derivatives.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Theory of computation → Formal languages and automata theory

Keywords and phrases POSIX matching and lexing, derivatives of regular expressions, Isabelle/HOL

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.27

Supplementary Material *Software (Source Code)*: `https://github.com/urbanchr/posix`

1 Introduction

In the last fifteen or so years, Brzozowski's derivatives of regular expressions have sparked quite a bit of interest in the functional programming and theorem prover communities. Derivatives of a regular expressions, written $r\backslash c$, give a simple solution to the problem of matching a string s with a regular expression r: if the derivative of r w.r.t. (in succession) all the characters of the string matches the empty string, then r matches s (and *vice versa*). The beauty of Brzozowski's derivatives [4] is that they are neatly expressible in any functional language, and easily definable and reasoned about in theorem provers – the definitions just consist of inductive datatypes and simple recursive functions. Another attractive feature of derivatives is that they can be easily extended to *bounded* regular expressions, such as $r^{\{n\}}$ or $r^{\{..n\}}$, where numbers or intervals of numbers specify how many times a regular expression should be used during matching.

However, there are two difficulties with derivative-based matchers: First, Brzozowski's original matcher only generates a yes/no answer for whether a regular expression matches a string or not. This is too little information in the context of lexing where separate tokens must be identified and also classified (for example as keywords or identifiers). Sulzmann and Lu [15] overcome this difficulty by cleverly extending Brzozowski's matching algorithm. Their extended version generates additional information on *how* a regular expression matches a string following the POSIX rules for regular expression matching. They achieve this by adding a second "phase" to Brzozowski's algorithm involving an injection function. In our own earlier work we provided the formal specification of what POSIX matching means and proved in Isabelle/HOL the correctness of Sulzmann and Lu's extended algorithm accordingly [2].


© Chengsong Tan and Christian Urban;
licensed under Creative Commons License CC-BY 4.0

The second difficulty is that Brzozowski's derivatives can grow to arbitrarily big sizes. For example if we start with the regular expression $(a + aa)^*$ and take successive derivatives according to the character a, we end up with a sequence of ever-growing derivatives like

$$\begin{array}{lll}
(a + aa)^* & \stackrel{\backslash a}{\longrightarrow} & (\mathbf{1} + \mathbf{1}a) \cdot (a + aa)^* \\
& \stackrel{\backslash a}{\longrightarrow} & (\mathbf{0} + \mathbf{0}a + \mathbf{1}) \cdot (a + aa)^* \;+\; (\mathbf{1} + \mathbf{1}a) \cdot (a + aa)^* \\
& \stackrel{\backslash a}{\longrightarrow} & (\mathbf{0} + \mathbf{0}a + \mathbf{0}) \cdot (a + aa)^* + (\mathbf{1} + \mathbf{1}a) \cdot (a + aa)^* + \\
& & \quad (\mathbf{0} + \mathbf{0}a + \mathbf{1}) \cdot (a + aa)^* + (\mathbf{1} + \mathbf{1}a) \cdot (a + aa)^* \\
& \stackrel{\backslash a}{\longrightarrow} & \ldots \qquad \text{(regular expressions of sizes 98, 169, 283, 468, 767, } \ldots\text{)}
\end{array}$$

where after around 35 steps we run out of memory on a typical computer (we shall define shortly the precise details of our regular expressions and the derivative operation). Clearly, the notation involving $\mathbf{0}$s and $\mathbf{1}$s already suggests simplification rules that can be applied to regular regular expressions, for example $\mathbf{0}\,r \Rightarrow \mathbf{0}$, $\mathbf{1}\,r \Rightarrow r$, $\mathbf{0} + r \Rightarrow r$ and $r + r \Rightarrow r$. While such simple-minded simplifications have been proved in our earlier work to preserve the correctness of Sulzmann and Lu's algorithm [2], they unfortunately do *not* help with limiting the growth of the derivatives shown above: the growth is slowed, but some derivatives can still grow rather quickly beyond any finite bound.

Sulzmann and Lu address this "growth problem" in a second algorithm [15] where they introduce bitcoded regular expressions. In this version, POSIX values are represented as bitsequences and such sequences are incrementally generated when derivatives are calculated. The compact representation of bitsequences and regular expressions allows them to define a more "aggressive" simplification method that keeps the size of the derivatives finitely bounded no matter what the length of the string is. They make some informal claims about the correctness and linear behaviour of this version, but do not provide any supporting proof arguments, not even "pencil-and-paper" arguments. They write about their bitcoded *incremental parsing method* (that is the algorithm to be fixed and formalised in this paper):

> *"Correctness Claim: We further claim that the incremental parsing method [..] in combination with the simplification steps [..] yields POSIX parse trees. We have tested this claim extensively [..] but yet have to work out all proof details." [15, Page 14]*

Contributions. We fill this gap by implementing in Isabelle/HOL our version of the derivative-based lexing algorithm of Sulzmann and Lu [15] where regular expressions are annotated with bitsequences. We define the crucial simplification function as a recursive function, without the need of a fixpoint operation. One objective of the simplification function is to remove duplicates of regular expressions. For this Sulzmann and Lu use in their paper the standard *nub* function from Haskell's list library, but this function does not achieve the intended objective with bitcoded regular expressions. The reason is that in the bitcoded setting, each copy generally has a different bitcode annotation – so *nub* would never "fire". Inspired by Scala's library for lists, we shall instead use a *distinctWith* function that finds duplicates under an "erasing" function that deletes bitcodes before comparing regular expressions. We shall also introduce our *own* arguments and definitions for establishing the correctness of the bitcoded algorithm when simplifications are included. Finally we establish that the size of derivatives can be finitely bounded.

In this paper, we shall first briefly introduce the basic notions of regular expressions and describe the definition of POSIX lexing from our earlier work [2]. This serves as a reference point for what correctness means in our Isabelle/HOL proofs. We shall then prove the correctness for the bitcoded algorithm without simplification, and after that extend the proof to include simplification. Our Isabelle code including the results from Sec. 5 is available from `https://github.com/urbanchr/posix`.

2 Background

In our Isabelle/HOL formalisation strings are lists of characters with the empty string being represented by the empty list, written [], and list-cons being written as _::_; string concatenation is _@_. We often use the usual bracket notation for lists also for strings; for example a string consisting of just a single character c is written $[c]$. Our regular expressions are defined as the following inductive datatype:

$$r ::= \mathbf{0} \mid \mathbf{1} \mid c \mid r_1 + r_2 \mid r_1 \cdot r_2 \mid r^* \mid r^{\{n\}}$$

where $\mathbf{0}$ stands for the regular expression that does not match any string, $\mathbf{1}$ for the regular expression that matches only the empty string and c for matching a character literal. The constructors $+$ and $\cdot$ represent alternatives and sequences, respectively. We sometimes omit the $\cdot$ in a sequence regular expression for brevity. The *language* of a regular expression, written $L(r)$, is defined as usual and we omit giving the definition here (see for example [2]).

In our work here we also add to the usual "basic" regular expressions the *bounded* regular expression $r^{\{n\}}$ where the n specifies that r should match exactly n-times (it is not included in Sulzmann and Lu's original work). For brevity we omit the other bounded regular expressions $r^{\{..n\}}$, $r^{\{n..\}}$ and $r^{\{n..m\}}$ which specify intervals for how many times r should match. The results presented in this paper extend straightforwardly to them, too. The importance of the bounded regular expressions is that they are often used in practical applications, such as Snort (a system for detecting network intrusions) and also in XML Schema definitions. According to Björklund et al [3], bounded regular expressions occur frequently in the latter and can have counters of up to ten million. The problem is that tools based on the classic notion of automata need to expand $r^{\{n\}}$ into n connected copies of the automaton for r. This leads to very inefficient matching algorithms or algorithms that consume large amounts of memory. A classic example is the regular expression $(a + b)^* \cdot a \cdot (a + b)^{\{n\}}$ where the minimal DFA requires at least 2^{n+1} states (see [16]). Therefore regular expression matching libraries that rely on the classic notion of DFAs often impose adhoc limits for bounded regular expressions: For example in the regular expression matching library in the Go language and also in Google's RE2 library the regular expression $a^{\{1001\}}$ is not permitted, because no counter can be above 1000; and in the regular expression library in Rust expressions such as $a^{\{1000\}\{100\}\{5\}}$ give an error message for being too big. Up until recently,[1] Rust however happily generated automata for regular expressions such as $a^{\{0\}\{4294967295\}}$. This was due to a bug in the algorithm that decides when a regular expression is acceptable or too big according to Rust's classification (it did not account for the fact that $a^{\{0\}}$ and similar examples can match the empty string). We shall come back to this example later in the paper. These problems can of course be solved in matching algorithms where automata go beyond the classic notion and for instance include explicit counters (e.g. [16]). The point here is that Brzozowski derivatives and the algorithms by Sulzmann and Lu can be straightforwardly extended to deal with bounded regular expressions and moreover the resulting code still consists of only simple recursive functions and inductive datatypes. Finally, bounded regular expressions do not destroy our finite boundedness property, which we shall prove later on.

Central to Brzozowski's regular expression matcher are two functions called *nullable* and *derivative*. The latter is written $r\backslash c$ for the derivative of the regular expression r w.r.t. the character c. Both functions are defined by recursion over regular expressions.

[1] up until version 1.5.4 of the regex library in Rust; see also CVE-2022-24713.

$$
\begin{array}{rcl}
\mathbf{0}\backslash c & \stackrel{\text{def}}{=} & \mathbf{0} \\
\mathbf{1}\backslash c & \stackrel{\text{def}}{=} & \mathbf{0} \\
d\backslash c & \stackrel{\text{def}}{=} & \mathit{if}\ c = d\ \mathit{then}\ \mathbf{1}\ \mathit{else}\ \mathbf{0} \\
(r_1 + r_2)\backslash c & \stackrel{\text{def}}{=} & r_1\backslash c + r_2\backslash c \\
(r_1 \cdot r_2)\backslash c & \stackrel{\text{def}}{=} & \mathit{if}\ \mathit{nullable}\ r_1 \\
 & & \mathit{then}\ (r_1\backslash c) \cdot r_2 + r_2\backslash c \\
 & & \mathit{else}\ (r_1\backslash c) \cdot r_2 \\
(r^*)\backslash c & \stackrel{\text{def}}{=} & (r\backslash c) \cdot r^* \\
(r^{\{n\}})\backslash c & \stackrel{\text{def}}{=} & \mathit{if}\ n = 0\ \mathit{then}\ \mathbf{0}\ \mathit{else}\ (r\backslash c) \cdot r^{\{n-1\}}
\end{array}
\qquad
\begin{array}{rcl}
\mathit{nullable}\ (\mathbf{0}) & \stackrel{\text{def}}{=} & \mathit{False} \\
\mathit{nullable}\ (\mathbf{1}) & \stackrel{\text{def}}{=} & \mathit{True} \\
\mathit{nullable}\ (c) & \stackrel{\text{def}}{=} & \mathit{False} \\
\mathit{nullable}\ (r_1 + r_2) & \stackrel{\text{def}}{=} & \mathit{nullable}\ r_1 \vee \mathit{nullable}\ r_2 \\
\mathit{nullable}\ (r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \mathit{nullable}\ r_1 \wedge \mathit{nullable}\ r_2 \\
\mathit{nullable}\ (r^*) & \stackrel{\text{def}}{=} & \mathit{True} \\
\mathit{nullable}\ (r^{\{n\}}) & \stackrel{\text{def}}{=} & \mathit{if}\ n = 0 \\
 & & \mathit{then}\ \mathit{True} \\
 & & \mathit{else}\ \mathit{nullable}\ r
\end{array}
$$

We can extend this definition to give derivatives w.r.t. strings, namely as $r\backslash [] \stackrel{\text{def}}{=} r$ and $r\backslash (c :: s) \stackrel{\text{def}}{=} (r\backslash c)\backslash s$. Using *nullable* and the derivative operation, we can define a simple regular expression matcher, namely $\mathit{match}\ s\ r \stackrel{\text{def}}{=} \mathit{nullable}(r\backslash s)$. This is essentially Brzozowski's algorithm from 1964. Its main virtue is that the algorithm can be easily implemented as a functional program (either in a functional programming language or in a theorem prover). The correctness of *match* amounts to establishing the property:

▶ **Proposition 1.** $\mathit{match}\ s\ r \quad \mathit{if\ and\ only\ if} \quad s \in L(r)$

It is a fun exercise to formally prove this property in a theorem prover. We are aware of a mechanised correctness proof of Brzozowski's derivative-based matcher in HOL4 by Owens and Slind [12]. Another one in Isabelle/HOL is part of the work by Krauss and Nipkow [8]. And another one in Coq is given by Coquand and Siles [5]. Also Ribeiro and Du Bois give one in Agda [14].

The novel idea of Sulzmann and Lu is to extend this algorithm for lexing, where it is important to find out which part of the string is matched by which part of the regular expression. For this Sulzmann and Lu presented two lexing algorithms in their paper [15]. The first algorithm consists of two phases: first a matching phase (which is Brzozowski's algorithm) and then a value construction phase. The values encode *how* a regular expression matches a string. *Values* are defined as the inductive datatype

$$v ::= \mathit{Empty} \mid \mathit{Char}\ c \mid \mathit{Left}\ v \mid \mathit{Right}\ v \mid \mathit{Seq}\ v_1\ v_2 \mid \mathit{Stars}\ vs$$

where we use vs to stand for a list of values. The string underlying a value can be calculated by a *flat* function, written $|\_|$. It traverses a value and collects the characters contained in it (see [2]).

Sulzmann and Lu also define inductively an inhabitation relation that associates values to regular expressions. Our version of this relation is defined by the following six rules:

$$
\frac{}{\vdash \mathit{Empty} : \mathbf{1}}
\qquad
\frac{\vdash v_1 : r_1}{\vdash \mathit{Left}\ v_1 : r_1 + r_2}
\qquad
\frac{\vdash v_2 : r_2}{\vdash \mathit{Right}\ v_2 : r_1 + r_2}
\qquad
\frac{\vdash v_1 : r_1 \qquad \vdash v_2 : r_2}{\vdash \mathit{Seq}\ v_1\ v_2 : r_1 \cdot r_2}
$$

$$
\frac{}{\vdash \mathit{Char}\ c : c}
\qquad
\frac{\forall v \in vs.\ \vdash v : r \wedge |v| \neq []}{\vdash \mathit{Stars}\ vs : r^*}
\qquad
\frac{\begin{array}{l}\forall v \in vs_1.\ \vdash v : r \wedge |v| \neq [] \\ \forall v \in vs_2.\ \vdash v : r \wedge |v| = [] \quad \mathit{len}\ (vs_1\ @\ vs_2) = n\end{array}}{\vdash \mathit{Stars}\ (vs_1\ @\ vs_2) : r^{\{n\}}}
$$

Note that no values are associated with the regular expression $\mathbf{0}$, since it cannot match any string. Interesting is our version of the rule for r^* where we require that each value in vs flattens to a non-empty string. This means if r^* matches the empty string, the related value must be of the form *Stars* []. But if r^* "fires" one or more times, then each copy in *Stars* vs needs to match a non-empty string. Similarly, in the rule for $r^{\{n\}}$ we require that the

$$\frac{}{([],\, \mathbf{1}) \to \mathit{Empty}}\, P\mathbf{1} \qquad \frac{}{([c],\, c) \to \mathit{Char}\ c}\, Pc \qquad \frac{(s,\, r_1) \to v}{(s,\, r_1 + r_2) \to \mathit{Left}\ v}\, P{+}L \qquad \frac{(s,\, r_2) \to v \qquad s \notin L\ r_1}{(s,\, r_1 + r_2) \to \mathit{Right}\ v}\, P{+}R$$

$$\frac{(s_1,\, r_1) \to v_1 \qquad (s_2,\, r_2) \to v_2 \qquad \nexists s_3\ s_4.\ s_3 \neq [] \wedge s_3 \mathbin{@} s_4 = s_2 \wedge s_1 \mathbin{@} s_3 \in L\ r_1 \wedge s_4 \in L\ r_2}{(s_1 \mathbin{@} s_2,\, r_1 \cdot r_2) \to \mathit{Seq}\ v_1\ v_2}\, PS$$

$$\frac{}{([],\, r^*) \to \mathit{Stars}\ []}\, P[] \qquad \frac{(s_1,\, r) \to v \qquad (s_2,\, r^*) \to \mathit{Stars}\ vs \qquad |v| \neq [] \qquad \nexists s_3\ s_4.\ s_3 \neq [] \wedge s_3 \mathbin{@} s_4 = s_2 \wedge s_1 \mathbin{@} s_3 \in L\ r \wedge s_4 \in L\ (r^*)}{(s_1 \mathbin{@} s_2,\, r^*) \to \mathit{Stars}\ (v :: vs)}\, P\star$$

$$\frac{\forall v \in vs.\ ([],\, r) \to v \qquad \mathit{len}\ vs = n}{([],\, r^{\{n\}}) \to \mathit{Stars}\ vs}\, Pn[] \qquad \frac{(s_1,\, r) \to v \qquad (s_2,\, r^{\{n\}}) \to \mathit{Stars}\ vs \qquad |v| \neq [] \qquad \nexists s_3\ s_4.\ s_3 \neq [] \wedge s_3 \mathbin{@} s_4 = s_2 \wedge s_1 \mathbin{@} s_3 \in L\ r \wedge s_4 \in L\ (r^{\{n\}})}{(s_1 \mathbin{@} s_2,\, r^{\{n\,+\,1\}}) \to \mathit{Stars}\ (v :: vs)}\, Pn{+}$$

Figure 1 The inductive definition of POSIX values taken from our earlier paper [2]. The ternary relation, written $(s, r) \to v$, formalises the notion of given a string s and a regular expression r what is the unique value v that satisfies the informal POSIX constraints for regular expression matching.

length of the list $vs_1 \mathbin{@} vs_2$ equals n (meaning the regular expression r matches n-times) and that the first segment of this list contains values that flatten to non-empty strings followed by a segment that only contains values that flatten to the empty string. It is routine to establish how values "inhabiting" a regular expression correspond to the language of a regular expression, namely $L\ r = \{|v| \mid \vdash v : r\}$.

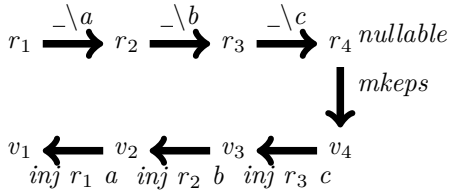
In general there is more than one value inhabiting a regular expression (meaning regular expressions can typically match more than one string). But even when fixing a string from the language of the regular expression, there are generally more than one way of how the regular expression can match this string. POSIX lexing is about identifying the unique value for a given regular expression and a string that satisfies the informal POSIX rules (see [13, 9, 11, 15, 17]). Sometimes these informal rules are called *maximal munch rule* and *rule priority*. One contribution of our earlier paper is to give a convenient specification for what POSIX values are (the inductive rules are shown in Figure 1).

The clever idea by Sulzmann and Lu [15] in their first algorithm is to define an injection function on values that mirrors (but inverts) the construction of the derivative on regular expressions. Essentially it injects back a character into a value. For this they define two functions called *mkeps* and *inj*:

$$\begin{array}{lcl}
\mathit{mkeps}\ \mathbf{1} & \stackrel{\text{def}}{=} & \mathit{Empty} \\
\mathit{mkeps}\ (r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \mathit{Seq}\ (\mathit{mkeps}\ r_1)\ (\mathit{mkeps}\ r_2) \\
\mathit{mkeps}\ (r_1 + r_2) & \stackrel{\text{def}}{=} & \mathit{if}\ \mathit{nullable}\ r_1\ \mathit{then}\ \mathit{Left}\ (\mathit{mkeps}\ r_1)\ \mathit{else}\ \mathit{Right}\ (\mathit{mkeps}\ r_2) \\
\mathit{mkeps}\ (r^*) & \stackrel{\text{def}}{=} & \mathit{Stars}\ [] \\
\mathit{mkeps}\ (r^{\{n\}}) & \stackrel{\text{def}}{=} & \mathit{Stars}\ (\mathit{replicate}\ n\ (\mathit{mkeps}\ r))
\end{array}$$

$$\begin{array}{lcl}
\mathit{inj}\ d\ c\ (\mathit{Empty}) & \stackrel{\text{def}}{=} & \mathit{Char}\ c \\
\mathit{inj}\ (r_1 + r_2)\ c\ (\mathit{Left}\ v_1) & \stackrel{\text{def}}{=} & \mathit{Left}\ (\mathit{inj}\ r_1\ c\ v_1) \\
\mathit{inj}\ (r_1 + r_2)\ c\ (\mathit{Right}\ v_2) & \stackrel{\text{def}}{=} & \mathit{Right}\ (\mathit{inj}\ r_2\ c\ v_2) \\
\mathit{inj}\ (r_1 \cdot r_2)\ c\ (\mathit{Seq}\ v_1\ v_2) & \stackrel{\text{def}}{=} & \mathit{Seq}\ (\mathit{inj}\ r_1\ c\ v_1)\ v_2 \\
\mathit{inj}\ (r_1 \cdot r_2)\ c\ (\mathit{Left}\ (\mathit{Seq}\ v_1\ v_2)) & \stackrel{\text{def}}{=} & \mathit{Seq}\ (\mathit{inj}\ r_1\ c\ v_1)\ v_2 \\
\mathit{inj}\ (r_1 \cdot r_2)\ c\ (\mathit{Right}\ v_2) & \stackrel{\text{def}}{=} & \mathit{Seq}\ (\mathit{mkeps}\ r_1)\ (\mathit{inj}\ r_2\ c\ v_2) \\
\mathit{inj}\ (r^*)\ c\ (\mathit{Seq}\ v\ (\mathit{Stars}\ vs)) & \stackrel{\text{def}}{=} & \mathit{Stars}\ (\mathit{inj}\ r\ c\ v :: vs) \\
\mathit{inj}\ (r^{\{n\}})\ c\ (\mathit{Seq}\ v\ (\mathit{Stars}\ vs)) & \stackrel{\text{def}}{=} & \mathit{Stars}\ (\mathit{inj}\ r\ c\ v :: vs)
\end{array}$$

The function *mkeps* is run when the last derivative is nullable, that is the string to be matched is in the language of the regular expression. It generates a value for how the last derivative can match the empty string. In case of $r^{\{n\}}$ we use the function *replicate* in order to generate a list of exactly n copies, which is the length of the list we expect in this case. The injection function then calculates the corresponding value for each intermediate derivative until a value for the original regular expression is generated. Graphically the algorithm by Sulzmann and Lu can be illustrated by the following picture where the path from the left to the right involving *derivatives*/*nullable* is the first phase of the algorithm (calculating successive Brzozowski's derivatives) and *mkeps*/*inj*, the path from right to left, the second phase.



The picture shows the steps required when a regular expression, say r_1, matches the string $[a, b, c]$. The first lexing algorithm by Sulzmann and Lu can be defined as:

$$\begin{array}{lcl} \mathit{lexer}\ r\ [] & \stackrel{\text{def}}{=} & \mathit{if}\ \mathit{nullable}\ r\ \mathit{then}\ \mathit{Some}\ (\mathit{mkeps}\ r)\ \mathit{else}\ \mathit{None} \\ \mathit{lexer}\ r\ (c :: s) & \stackrel{\text{def}}{=} & \mathit{case}\ \mathit{lexer}\ (r\backslash c)\ s\ \mathit{of}\ \mathit{None} \Rightarrow \mathit{None} \mid \mathit{Some}\ v \Rightarrow \mathit{Some}\ (\mathit{inj}\ r\ c\ v) \end{array}$$

We have shown in our earlier paper [2] that this algorithm is correct, that is it generates POSIX values. The central property we established relates the derivative operation to the injection function.

▶ **Proposition 2.** *If* $(s,\ r\backslash c) \to v$ *then* $(c :: s,\ r) \to \mathit{inj}\ r\ c\ v$.

With this in place we were able to prove:

▶ **Proposition 3.** *(1)* $s \notin L\ r$ *if and only if* $\mathit{lexer}\ r\ s = \mathit{None}$.
(2) $s \in L\ r$ *if and only if* $\exists v.\ \mathit{lexer}\ r\ s = \mathit{Some}\ v \wedge (s,\ r) \to v$.

In fact we have shown that, in the success case, the generated POSIX value v is unique and in the failure case that there is no POSIX value v that satisfies $(s, r) \to v$. While the algorithm is correct, it is excruciatingly slow in cases where the derivatives grow arbitrarily (recall the example from the Introduction). However it can be used as a convenient reference point for the correctness proof of the second algorithm by Sulzmann and Lu, which we shall describe next.

3 Bitcoded Regular Expressions and Derivatives

In the second part of their paper [15], Sulzmann and Lu describe another algorithm that also generates POSIX values but dispenses with the second phase where characters are injected "back" into values. For this they annotate bitcodes to regular expressions, which we define in Isabelle/HOL as the datatype

$$\mathit{breg} ::= \mathit{ZERO} \mid \mathit{ONE}\ \mathit{bs} \mid \mathit{CHAR}\ \mathit{bs}\ c \mid \mathit{ALTs}\ \mathit{bs}\ \mathit{rs} \mid \mathit{SEQ}\ \mathit{bs}\ r_1\ r_2 \mid \mathit{STAR}\ \mathit{bs}\ r \mid \mathit{NT}\ \mathit{bs}\ r\ n$$

$$
\begin{array}{lcl}
\mathit{decode}'\,bs\,(\mathbf{1}) & \stackrel{\text{def}}{=} & (\mathit{Empty}, bs)\\
\mathit{decode}'\,bs\,(c) & \stackrel{\text{def}}{=} & (\mathit{Char}\,c, bs)\\
\mathit{decode}'\,(Z::bs)\,(r_1 + r_2) & \stackrel{\text{def}}{=} & \mathit{let}\,(v, bs_1) = \mathit{decode}'\,bs\,r_1\ \mathit{in}\ (\mathit{Left}\,v, bs_1)\\
\mathit{decode}'\,(S::bs)\,(r_1 + r_2) & \stackrel{\text{def}}{=} & \mathit{let}\,(v, bs_1) = \mathit{decode}'\,bs\,r_2\ \mathit{in}\ (\mathit{Right}\,v, bs_1)\\
\mathit{decode}'\,bs\,(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \mathit{let}\,(v_1, bs_1) = \mathit{decode}'\,bs\,r_1\ \mathit{in}\\
 & & \mathit{let}\,(v_2, bs_2) = \mathit{decode}'\,bs_1\,r_2 \quad \mathit{in}\ (\mathit{Seq}\,v_1\,v_2, bs_2)\\
\mathit{decode}'\,(S::bs)\,(r^*) & \stackrel{\text{def}}{=} & (\mathit{Stars}\,[], bs)\\
\mathit{decode}'\,(Z::bs)\,(r^*) & \stackrel{\text{def}}{=} & \mathit{let}\,(v, bs_1) = \mathit{decode}'\,bs\,r\ \mathit{in}\\
 & & \mathit{let}\,(\mathit{Stars}\,vs, bs_2) = \mathit{decode}'\,bs_1\,(r^*) \quad \mathit{in}\ (\mathit{Stars}\,v::vs, bs_2)\\
\mathit{decode}'\,bs\,(r^{\{n\}}) & \stackrel{\text{def}}{=} & \mathit{decode}'\,bs\,(r^*)\\
\\
\mathit{decode}\,bs\,r & \stackrel{\text{def}}{=} & \mathit{let}\,(v, bs') = \mathit{decode}'\,bs\,r\ \mathit{in}\ \ \mathit{if}\,bs' = []\ \mathit{then}\ \mathit{Some}\,v\ \mathit{else}\ \mathit{None}
\end{array}
$$

Figure 2 Two functions, called *decode′* and *decode*, for decoding a value from a bitsequence with the help of a regular expression. The first function terminates because in each call the size of the regular expression decreases or stays the same (the size of $r^{\{n\}}$ is assumed to be bigger than the size of r^*). In the star-case where the size stays the same in the second recursive call, the length of the decoded bitsequence is shorter. Therefore a lexicographic measure of the size of the regular expression and the length of the bitsequence decreases in every recursive call.

where bs stands for bitsequences; r, r_1 and r_2 for bitcoded regular expressions; and rs for lists of bitcoded regular expressions. The binary alternative $ALT\ bs\ r_1\ r_2$ is just an abbreviation for $ALTs\ bs\ [r_1, r_2]$. The NT-regular expression, called *N-Times*, contains an integer n, which corresponds to how many times $r^{\{n\}}$ should fire. For bitsequences we use lists made up of the constants Z and S. The idea with bitcoded regular expressions is to incrementally generate the value information (for example *Left* and *Right*) as bitsequences. For this Sulzmann and Lu follow Nielsen and Henglein [10] and define a coding function for how values can be coded into bitsequences.

$$
\begin{array}{lcl@{\qquad}lcl}
\mathit{code}\ (\mathit{Empty}) & \stackrel{\text{def}}{=} & [] & \mathit{code}\ (\mathit{Seq}\ v_1\ v_2) & \stackrel{\text{def}}{=} & \mathit{code}\ v_1\ @\ \mathit{code}\ v_2\\
\mathit{code}\ (\mathit{Char}\ c) & \stackrel{\text{def}}{=} & [] & \mathit{code}\ (\mathit{Stars}\ []) & \stackrel{\text{def}}{=} & [S]\\
\mathit{code}\ (\mathit{Left}\ v) & \stackrel{\text{def}}{=} & Z::\mathit{code}\ v & \mathit{code}\ (\mathit{Stars}\ (v::vs)) & \stackrel{\text{def}}{=} & Z::\mathit{code}\ v\ @\ \mathit{code}\ (\mathit{Stars}\ vs)\\
\mathit{code}\ (\mathit{Right}\ v) & \stackrel{\text{def}}{=} & S::\mathit{code}\ v & & &
\end{array}
$$

As can be seen, this coding is "lossy" in the sense that it does not record explicitly character values and also not sequence values (for them it just appends two bitsequences). However, the different alternatives for *Left*, respectively *Right*, are recorded as Z and S followed by some bitsequence. Similarly, we use Z to indicate if there is still a value coming in the list of *Stars*, whereas S indicates the end of the list. The lossiness makes the process of decoding a bit more involved, but the point is that if we have a regular expression *and* a bitsequence of a corresponding value, then we can always decode the value accurately (see Fig. 2). The function *decode* checks whether all of the bitsequence is consumed and returns the corresponding value as *Some* v; otherwise it fails with *None*. We can establish that for a value v inhabiting a regular expression r, the decoding of its bitsequence never fails (see also [10]).

▶ **Lemma 4.** *If* $\vdash v : r$ *then* $\mathit{decode}\,(\mathit{code}\,v)\,r = \mathit{Some}\,v$.

Proof. This follows from the property that $\mathit{decode}'\,((\mathit{code}\,v) \mathbin{@} bs)\,r = (v, bs)$ holds for any bit-sequence bs and $\vdash v : r$. This property can be easily proved by induction on $\vdash v : r$. ◀

Sulzmann and Lu define the function *internalise* in order to transform (standard) regular expressions into annotated regular expressions. We write this operation as $r^{\uparrow}$. This internalisation uses the following *fuse* function.

$$
\begin{array}{lcl}
\mathit{fuse}\, bs\, (\mathit{ZERO}) & \stackrel{\text{def}}{=} & \mathit{ZERO} \\
\mathit{fuse}\, bs\, (\mathit{ONE}\, bs') & \stackrel{\text{def}}{=} & \mathit{ONE}\, (bs \mathbin{@} bs') \\
\mathit{fuse}\, bs\, (\mathit{CHAR}\, bs'\, c) & \stackrel{\text{def}}{=} & \mathit{CHAR}\, (bs \mathbin{@} bs')\, c
\end{array}
\qquad
\begin{array}{lcl}
\mathit{fuse}\, bs\, (\mathit{ALTs}\, bs'\, rs) & \stackrel{\text{def}}{=} & \mathit{ALTs}\, (bs \mathbin{@} bs')\, rs \\
\mathit{fuse}\, bs\, (\mathit{SEQ}\, bs'\, r_1\, r_2) & \stackrel{\text{def}}{=} & \mathit{SEQ}\, (bs \mathbin{@} bs')\, r_1\, r_2 \\
\mathit{fuse}\, bs\, (\mathit{STAR}\, bs'\, r) & \stackrel{\text{def}}{=} & \mathit{STAR}\, (bs \mathbin{@} bs')\, r \\
\mathit{fuse}\, bs\, (\mathit{NT}\, bs'\, r\, n) & \stackrel{\text{def}}{=} & \mathit{NT}\, (bs \mathbin{@} bs')\, r\, n
\end{array}
$$

This function "fuses" a bitsequence to the topmost constructor of a bitcoded regular expressions. A regular expression can then be *internalised* into a bitcoded regular expression as follows:

$$
\begin{array}{lcl}
(\mathbf{0})^{\uparrow} & \stackrel{\text{def}}{=} & \mathit{ZERO} \\
(\mathbf{1})^{\uparrow} & \stackrel{\text{def}}{=} & \mathit{ONE}\, [] \\
(c)^{\uparrow} & \stackrel{\text{def}}{=} & \mathit{CHAR}\, []\, c \\
(r^*)^{\uparrow} & \stackrel{\text{def}}{=} & \mathit{STAR}\, []\, r^{\uparrow}
\end{array}
\qquad
\begin{array}{lcl}
(r_1 + r_2)^{\uparrow} & \stackrel{\text{def}}{=} & \mathit{ALT}\, []\, (\mathit{fuse}\, [Z]\, r_1^{\uparrow})\, (\mathit{fuse}\, [S]\, r_2^{\uparrow}) \\
(r_1 \cdot r_2)^{\uparrow} & \stackrel{\text{def}}{=} & \mathit{SEQ}\, []\, r_1^{\uparrow}\, r_2^{\uparrow} \\
(r^{\{n\}})^{\uparrow} & \stackrel{\text{def}}{=} & \mathit{NT}\, []\, r^{\uparrow}\, n
\end{array}
$$

There is also an *erase*-function, written $r^{\downarrow}$, which transforms a bitcoded regular expression into a (standard) regular expression by just erasing the annotated bitsequences. We omit the straightforward definition. For defining the algorithm, we also need the functions *bnullable* and *bmkeps*(s), which are the "lifted" versions of *nullable* and *mkeps* acting on bitcoded regular expressions.

$$
\begin{array}{lcl}
\mathit{bnullable}\, (\mathit{ZERO}) & \stackrel{\text{def}}{=} & \mathit{False} \\
\mathit{bnullable}\, (\mathit{ONE}\, bs) & \stackrel{\text{def}}{=} & \mathit{True} \\
\mathit{bnullable}\, (\mathit{CHAR}\, bs\, c) & \stackrel{\text{def}}{=} & \mathit{False} \\
\mathit{bnullable}\, (\mathit{ALTs}\, bs\, rs) & \stackrel{\text{def}}{=} & \\
\quad \exists\, r \in rs.\, \mathit{bnullable}\, r & & \\
\mathit{bnullable}\, (\mathit{SEQ}\, bs\, r_1\, r_2) & \stackrel{\text{def}}{=} & \\
\quad \mathit{bnullable}\, r_1 \wedge \mathit{bnullable}\, r_2 & & \\
\mathit{bnullable}\, (\mathit{STAR}\, bs\, r) & \stackrel{\text{def}}{=} & \mathit{True} \\
\mathit{bnullable}\, (\mathit{NT}\, bs\, r\, n) & \stackrel{\text{def}}{=} & \\
\quad \mathit{if}\, n = 0\ \mathit{then}\ \mathit{True}\ \mathit{else}\ \mathit{bnullable}\, r & &
\end{array}
\qquad
\begin{array}{lcl}
\mathit{bmkeps}\, (\mathit{ONE}\, bs) & \stackrel{\text{def}}{=} & bs \\
\mathit{bmkeps}\, (\mathit{ALTs}\, bs\, rs) & \stackrel{\text{def}}{=} & bs \mathbin{@} \mathit{bmkepss}\, rs \\
\mathit{bmkeps}\, (\mathit{SEQ}\, bs\, r_1\, r_2) & \stackrel{\text{def}}{=} & \\
\quad bs \mathbin{@} \mathit{bmkeps}\, r_1 \mathbin{@} \mathit{bmkeps}\, r_2 & & \\
\mathit{bmkeps}\, (\mathit{STAR}\, bs\, r) & \stackrel{\text{def}}{=} & bs \mathbin{@} [S] \\
\mathit{bmkeps}\, (\mathit{NT}\, bs\, r\, n) & \stackrel{\text{def}}{=} & \\
\quad \mathit{if}\, n = 0\ \mathit{then}\ bs \mathbin{@} [S] & & \\
\quad \mathit{else}\, bs \mathbin{@} [Z] \mathbin{@} \mathit{bmkeps}\, r \mathbin{@} \mathit{bmkeps}\, (\mathit{NT}\ []\ r\ (n - 1)) & & \\
\mathit{bmkepss}\, (r :: rs) & \stackrel{\text{def}}{=} & \\
\quad \mathit{if}\, \mathit{bnullable}\, r\ \mathit{then}\ \mathit{bmkeps}\, r\ \mathit{else}\ \mathit{bmkepss}\, rs & &
\end{array}
$$

The key function in the bitcoded algorithm is the derivative of a bitcoded regular expression. This derivative function calculates the derivative but at the same time also the incremental part of the bitsequences that contribute to constructing a POSIX value.

$$
\begin{array}{lcl}
(\mathit{ZERO})\backslash c & \stackrel{\text{def}}{=} & \mathit{ZERO} \\
(\mathit{ONE}\, bs)\backslash c & \stackrel{\text{def}}{=} & \mathit{ZERO} \\
(\mathit{CHAR}\, bs\, d)\backslash c & \stackrel{\text{def}}{=} & \mathit{if}\, c = d\ \ \mathit{then}\ \mathit{ONE}\, bs\ \mathit{else}\ \mathit{ZERO} \\
(\mathit{ALTs}\, bs\, rs)\backslash c & \stackrel{\text{def}}{=} & \mathit{ALTs}\, bs\, (\mathit{map}\, (\_\backslash c)\, rs) \\
(\mathit{SEQ}\, bs\, r_1\, r_2)\backslash c & \stackrel{\text{def}}{=} & \mathit{if}\, \mathit{bnullable}\, r_1 \\
 & & \mathit{then}\ \mathit{ALT}\, bs\ (\mathit{SEQ}\, []\, (r_1\backslash c)\, r_2)\ \ (\mathit{fuse}\, (\mathit{bmkeps}\, r_1)\, (r_2\backslash c)) \\
 & & \mathit{else}\ \mathit{SEQ}\, bs\, (r_1\backslash c)\, r_2 \\
(\mathit{STAR}\, bs\, r)\backslash c & \stackrel{\text{def}}{=} & \mathit{SEQ}\ (bs \mathbin{@} [Z])\ (r\backslash c)\ (\mathit{STAR}\, []\, r) \\
(\mathit{NT}\, bs\, r\, n)\backslash c & \stackrel{\text{def}}{=} & \mathit{if}\, n = 0\ \mathit{then}\ \mathit{ZERO}\ \mathit{else}\ \mathit{SEQ}\ (bs \mathbin{@} [Z])\ (r\backslash c)\ (\mathit{NT}\, []\, r\, (n - 1))
\end{array}
$$

This function can also be extended to strings, written $r\backslash s$, just like the standard derivative. We omit the details. Finally we can define Sulzmann and Lu's bitcoded lexer, which we call *blexer*:

$$blexer\, r\, s \stackrel{\text{def}}{=} let\, r_{der} = (r^{\uparrow})\backslash s\ in\ \ if\, bnullable(r_{der})\ \ then\, decode\,(bmkeps\, r_{der})\, r\ \ else\, None$$

This bitcoded lexer first internalises the regular expression r and then builds the bitcoded derivative according to s. If the derivative is (b)nullable the string is in the language of r and it extracts the bitsequence using the *bmkeps* function. Finally it decodes the bitsequence into a value. If the derivative is *not* nullable, then *None* is returned. We can show that this way of calculating a value generates the same result as *lexer*.

Before we can proceed we need to define a helper function, called *retrieve*, which Sulzmann and Lu introduced for the correctness proof.

$$\begin{array}{lcl}
retrieve\ (ONE\ bs)\ (Empty) & \stackrel{\text{def}}{=} & bs \\
retrieve\ (CHAR\ bs\ c)\ (Char\ d) & \stackrel{\text{def}}{=} & bs \\
retrieve\ (ALTs\ bs\ [r])\ v & \stackrel{\text{def}}{=} & bs\ @\ retrieve\ r\ v \\
retrieve\ (ALTs\ bs\ (r :: rs))\ (Left\ v) & \stackrel{\text{def}}{=} & bs\ @\ retrieve\ r\ v \\
retrieve\ (ALTs\ bs\ (r :: rs))\ (Right\ v) & \stackrel{\text{def}}{=} & bs\ @\ retrieve\ (ALTs\ []\ rs)\ v \\
retrieve\ (SEQ\ bs\ r_1\ r_2)\ (Seq\ v_1\ v_2) & \stackrel{\text{def}}{=} & bs\ @\ retrieve\ r_1\ v_1\ @\ retrieve\ r_2\ v_2 \\
retrieve\ (STAR\ bs\ r)\ (Stars\ []) & \stackrel{\text{def}}{=} & bs\ @\ [S] \\
retrieve\ (STAR\ bs\ r)\ (Stars\ (v :: vs)) & \stackrel{\text{def}}{=} & \\
\multicolumn{3}{r}{bs\ @\ [Z]\ @\ retrieve\ r\ v\ @\ retrieve\ (STAR\ []\ r)\ (Stars\ vs)} \\
retrieve\ (NT\ bs\ r\ 0)\ (Stars\ []) & \stackrel{\text{def}}{=} & bs\ @\ [S] \\
retrieve\ (NT\ bs\ r\ (n + 1))\ (Stars\ (v :: vs)) & \stackrel{\text{def}}{=} & \\
\multicolumn{3}{r}{bs\ @\ [Z]\ @\ retrieve\ r\ v\ @\ retrieve\ (NT\ []\ r\ n)\ (Stars\ vs)}
\end{array}$$

The idea behind this function is to retrieve a possibly partial bitsequence from a bitcoded regular expression, where the retrieval is guided by a value. For example if the value is *Left* then we descend into the left-hand side of an alternative in order to assemble the bitcode. Similarly for *Right*. The property we can show is that for a given v and r with $\vdash v : r$, the retrieved bitsequence from the internalised regular expression is equal to the bitcoded version of v.

▶ **Lemma 5.** *If* $\vdash v : r$ *then* $code\, v = retrieve\,(r^{\uparrow})\, v$.

We also need some auxiliary facts about how the bitcoded operations relate to the "standard" operations on regular expressions. For example if we build a bitcoded derivative and erase the result, this is the same as if we first erase the bitcoded regular expression and then perform the "standard" derivative operation.

▶ **Lemma 6.** *(1)* $(r\backslash s)^{\downarrow} = (r^{\downarrow})\backslash s$
(2) $bnullable(r)$ *iff* $nullable(r^{\downarrow})$
(3) $bmkeps(r) = retrieve\, r\, (mkeps\,(r^{\downarrow}))$ *provided* $nullable(r^{\downarrow})$

Proof. All properties are by induction on annotated regular expressions. ◀

The only difficulty left for the correctness proof is that the bitcoded algorithm has only a "forward phase" where POSIX values are generated incrementally. We can achieve the same effect with *lexer* (which has two phases) by stacking up injection functions during the forward phase. An auxiliary function, called *flex*, allows us to recast the rules of *lexer* in terms of a single phase and stacked up injection functions.

$$flex\, r\, f\, [] \stackrel{\text{def}}{=} f \qquad flex\, r\, f\, (c :: s) \stackrel{\text{def}}{=} flex\,(r\backslash c)\,(\lambda v.\, f\,(inj\, r\, c\, v))\, s$$

The point of this function is that when reaching the end of the string, we just need to apply the stacked up injection functions to the value generated by *mkeps*. Using this function we can recast the success case in *lexer* as follows:

▶ **Lemma 7.** *If* $lexer\ r\ s = Some\ v$ *then* $v = flex\, r\, id\, s\, (mkeps\, (r \backslash s))$.

Note we did not redefine *lexer*, we just established that the value generated by *lexer* can also be obtained by a different method. While this different method is not efficient (we essentially need to traverse the string s twice, once for building the derivative $r\backslash s$ and another time for stacking up injection functions), it helps us with proving that incrementally building up values in *blexer* generates the same result.

This brings us to our main lemma in this section: if we calculate a derivative, say $r\backslash s$, and have a value, say v, inhabiting this derivative, then we can produce the result *lexer* generates by applying this value to the stacked-up injection functions that *flex* assembles. The lemma establishes that this is the same value as if we build the annotated derivative $r^\uparrow\backslash s$ and then retrieve the bitcoded version of v, followed by a decoding step.

▶ **Lemma 8** (Main Lemma). *If* $\vdash v : r\backslash s$ *then* $Some\, (flex\, r\, id\, s\, v) = decode(retrieve\, (r^\uparrow\backslash s)\, v)\, r$

Proof. This can be proved by induction on s and generalising over v. The interesting point is that we need to prove this in the reverse direction for s. This means instead of cases [] and $c :: s$, we have cases [] and $s \,@\, [c]$ where we unravel the string from the back.[2] ◀

We can then prove the correctness of *blexer* – it indeed produces the same result as *lexer*.

▶ **Theorem 9.** $blexer\, r\, s = lexer\, r\, s$

This establishes that the bitcoded algorithm *without* simplification produces correct results. This was only conjectured by Sulzmann and Lu in their paper [15]. The next step is to add simplifications.

4 Simplification

Derivatives as calculated by Brzozowski's method are usually more complex regular expressions than the initial one; the result is that derivative-based matching and lexing algorithms are often abysmally slow if the "growth problem" is not addressed. As Sulzmann and Lu wrote, various optimisations are possible, such as the simplifications $\mathbf{0}\, r \Rightarrow \mathbf{0}$, $\mathbf{1}\, r \Rightarrow r$, $\mathbf{0} + r \Rightarrow r$ and $r + r \Rightarrow r$. While these simplifications can considerably speed up the two algorithms in many cases, they do not solve fundamentally the growth problem with derivatives. To see this let us return to the example from the Introduction that shows the derivatives for $(a + aa)^*$. If we delete in the 3rd step all $\mathbf{0}s$ and $\mathbf{1}$s according to the simplification rules shown above we obtain

$$(a + aa)^* \xrightarrow{\_\backslash [a,a,a]} \underbrace{(\mathbf{1} + a) \cdot (a + aa)^*}_{r} + ((a + aa)^* + \underbrace{(\mathbf{1} + a) \cdot (a + aa)^*}_{r}) \tag{1}$$

This is a simpler derivative, but unfortunately we cannot make any further simplifications. This is a problem because the outermost alternatives contains two copies of the same regular expression (underlined with r). These copies will spawn new copies in later derivative

[2] Isabelle/HOL provides an induction principle for this way of performing the induction.

steps and they in turn even more copies. This destroys any hope of taming the size of the derivatives. But the second copy of r in (1) will never contribute to a value, because POSIX lexing will always prefer matching a string with the first copy. So it could be safely removed without affecting the correctness of the algorithm. The issue with the simple-minded simplification rules above is that the rule $r + r \Rightarrow r$ will never be applicable because as can be seen in this example the regular expressions are not next to each other but separated by another regular expression.

But here is where Sulzmann and Lu's representation of generalised alternatives in the bitcoded algorithm shines: in $ALTs\ bs\ rs$ we can define a more aggressive simplification by recursively simplifying all regular expressions in rs and then analyse the resulting list and remove any duplicates. Another advantage with the bitsequences in bitcoded regular expressions is that they can be easily modified such that simplification does not interfere with the value constructions. For example we can "flatten", or de-nest, or spill out, $ALTs$ as follows

$$ALTs\ bs_1\ ((ALTs\ bs_2\ rs_2) :: rs_1) \quad \xrightarrow{bsimp} \quad ALTs\ bs_1\ ((map\ (fuse\ bs_2)\ rs_2) \mathbin{@} rs_1)$$

where we just need to fuse the bitsequence that has accumulated in bs_2 to the alternatives in rs_2. As we shall show below this will ensure that the correct value corresponding to the original (unsimplified) regular expression can still be extracted.

However there is one problem with the definition for the more aggressive simplification rules described by Sulzmann and Lu. Recasting their definition with our syntax they define the step of removing duplicates as

$$bsimp\ (ALTs\ bs\ rs) \stackrel{\text{def}}{=} ALTs\ bs\ (nub\ (map\ bsimp\ rs))$$

where they first recursively simplify the regular expressions in rs (using map) and then use Haskell's nub-function to remove potential duplicates in lists. Nub decides if an element is a duplicate by checking whether it is an *exact* copy of an earlier element in the list. While this makes sense when considering the example shown in (1), nub is the inappropriate function in the case of bitcoded regular expressions. The reason is that in general the elements in rs will have a different annotated bitsequence and in this way nub will never find a duplicate to be removed. One correct way to handle this situation is to first *erase* the regular expressions when comparing potential duplicates. This is inspired by Scala's list functions of the form $distinctWith\ rs\ eq\ acc$ where eq is an user-defined equivalence relation that compares two elements in rs. We define this function in Isabelle/HOL as

$$\begin{array}{lcl} distinctWith\ []\ eq\ acc & \stackrel{\text{def}}{=} & [] \\ distinctWith\ (x :: xs)\ eq\ acc & \stackrel{\text{def}}{=} & if\ (\exists\ y \in acc.\ eq\ x\ y)\ then\ distinctWith\ xs\ eq\ acc \\ & & else\ x :: distinctWith\ xs\ eq\ (\{x\} \cup acc) \end{array}$$

where we scan the list from left to right (because we have to remove later copies). In $distinctWith$, eq is intended to be an equivalence relation for bitcoded regular expressions and acc is an accumulator for bitcoded regular expressions – essentially a set of regular expressions that we have already seen while scanning the list. Therefore we delete an element, say x, from the list provided a y with y being equivalent to x is already in the accumulator; otherwise we keep x and scan the rest of the list but add x as another "seen" element to acc. We will use $distinctWith$ where eq is an equivalence that deletes bitsequences from bitcoded regular expressions before comparing the components. One way to define this in Isabelle/HOL is by the following recursive function from bitcoded regular expressions to $bool$:

$$
\begin{array}{lll}
ZERO \approx ZERO & \stackrel{\text{def}}{=} & True \\
ONE\ \_ \approx ONE\ \_ & \stackrel{\text{def}}{=} & True \\
STAR\ \_\ r_1 \approx STAR\ \_\ r_2 & \stackrel{\text{def}}{=} & r_1 \approx r_2 \\
ALTs\ \_\ [] \approx ALTs\ \_\ [] & \stackrel{\text{def}}{=} & True
\end{array}
\qquad
\begin{array}{lll}
CHAR\ \_\ c \approx CHAR\ \_\ d & \stackrel{\text{def}}{=} & c = d \\
SEQ\ \_\ r_{11}\ r_{12} \approx SEQ\ \_\ r_{21}\ r_{22} & \stackrel{\text{def}}{=} & \\
\multicolumn{3}{r}{r_{11} \approx r_{21} \wedge r_{12} \approx r_{22}} \\
NT\ \_\ r_1\ n_1 \approx NT\ \_\ r_2\ n_2 & \stackrel{\text{def}}{=} & r_1 \approx r_2 \wedge n_1 = n_2
\end{array}
$$

$$ALTs\ \_\ (r_1 :: rs_1) \approx ALTs\ \_\ (r_2 :: rs_2) \stackrel{\text{def}}{=} r_1 \approx r_2 \wedge ALTs\ \_\ rs_1 \approx ALTs\ \_\ rs_2$$

where all other cases are set to *False*. This equivalence is clearly a computationally more expensive operation than *nub*, but is needed in order to make the removal of unnecessary copies to work properly.

Our simplification function depends on three more helper functions, one is called *flts* and analyses lists of regular expressions coming from alternatives. It is defined by four clauses as follows:

$$
\begin{array}{lll}
flts\ [] & \stackrel{\text{def}}{=} & [] \\
flts\ (ZERO :: rs) & \stackrel{\text{def}}{=} & flts\ rs \\
flts\ ((ALTs\ bs'\ rs') :: rs & \stackrel{\text{def}}{=} & map\ (fuse\ bs')\ rs' \ @\ flts\ rs \\
flts\ (r :: rs) & \stackrel{\text{def}}{=} & r :: flts\ rs \quad \text{(otherwise)}
\end{array}
$$

The second clause of *flts* removes all instances of *ZERO* in alternatives and the third "de-nests" alternatives (but retains the bitsequence bs' accumulated in the inner alternative). There are some corner cases to be considered when the resulting list inside an alternative is empty or a singleton list. We take care of those cases in the *bsimpALTs* function; similarly we define a helper function that simplifies sequences according to the usual rules about *ZERO*s and *ONE*s:

$$
\begin{array}{lll}
bsimpALTs\ bs\ [] & \stackrel{\text{def}}{=} & ZERO \\
bsimpALTs\ bs\ [r] & \stackrel{\text{def}}{=} & fuse\ bs\ r \\
bsimpALTs\ bs\ rs & \stackrel{\text{def}}{=} & ALTs\ bs\ rs
\end{array}
\qquad
\begin{array}{lll}
bsimpSEQ\ bs\ \_\ ZERO & \stackrel{\text{def}}{=} & ZERO \\
bsimpSEQ\ bs\ ZERO\ \_ & \stackrel{\text{def}}{=} & ZERO \\
bsimpSEQ\ bs_1\ (ONE\ bs_2)\ r_2 & \stackrel{\text{def}}{=} & fuse\ (bs_1\ @\ bs_2)\ r_2 \\
bsimpSEQ\ bs\ r_1\ r_2 & \stackrel{\text{def}}{=} & SEQ\ bs\ r_1\ r_2
\end{array}
$$

With this in place we can define our simplification function as

$$
\begin{array}{lll}
bsimp\ (SEQ\ bs\ r_1\ r_2) & \stackrel{\text{def}}{=} & bsimpSEQ\ bs\ (bsimp\ r_1)\ (bsimp\ r_2) \\
bsimp\ (ALTs\ bs\ rs) & \stackrel{\text{def}}{=} & bsimpALTs\ bs\ (distinctWith\ (flts\ (map\ bsimp\ rs))\ \approx\ \varnothing) \\
bsimp\ r & \stackrel{\text{def}}{=} & r
\end{array}
$$

We believe our recursive function *bsimp* simplifies bitcoded regular expressions as intended by Sulzmann and Lu with the small addition of removing "useless" *ONE*s in sequence regular expressions. There is no point in applying the *bsimp* function repeatedly (like the simplification in their paper which needs to be applied until a fixpoint is reached) because we can show that *bsimp* is idempotent, that is

▶ **Proposition 10.** $bsimp\ (bsimp\ r) = bsimp\ r$

This can be proved by induction on r but requires a detailed analysis that the de-nesting of alternatives always results in a flat list of regular expressions. We omit the details since it does not concern the correctness proof.

Next we can include simplification after each derivative step leading to the following notion of bitcoded derivatives:

$$r\backslash_{bsimp}\ [] \stackrel{\text{def}}{=} r \qquad r\backslash_{bsimp}\ (c :: s) \stackrel{\text{def}}{=} bsimp\ (r\backslash c)\backslash_{bsimp}\ s$$

and use it in the improved lexing algorithm defined as

$$blexer^{+}\ r\ s \stackrel{\text{def}}{=} let\ r_{der} = (r^{\uparrow})\backslash_{bsimp}\ s\ in\quad if\ bnullable(r_{der})\ \ then\ decode\,(bmkeps\,r_{der})\,r\ \ else\ None$$

Note that in $blexer^{+}$ the derivative r_{der} is calculated using the simplifying derivative $\_\backslash_{bsimp}\_$. The remaining task is to show that *blexer* and $blexer^{+}$ generate the same answers.

When we first attempted this proof we encountered a problem with the idea in Sulzmann and Lu's paper where the argument seems to be to appeal again to the *retrieve*-function defined for the unsimplified version of the algorithm. But this does not work, because desirable properties such as

$$retrieve\ r\ v = retrieve\ (bsimp\ r)\ v$$

do not hold under simplification – this property essentially purports that we can retrieve the same value from a simplified version of the regular expression. To start with *retrieve* depends on the fact that the value v corresponds to the structure of the regular expression r – but the whole point of simplification is to "destroy" this structure by making the regular expression simpler. To see this consider the regular expression $r = r' + \mathbf{0}$ and a corresponding value $v = Left\ v'$. If we annotate bitcodes to r, then we can use *retrieve* with r and v in order to extract a corresponding bitsequence. The reason that this works is that r is an alternative regular expression and v a corresponding *Left*-value. However, if we simplify r, then v does not correspond to the shape of the regular expression anymore. So unless one can somehow synchronise the change in the simplified regular expressions with the original POSIX value, there is no hope of appealing to *retrieve* in the correctness argument for $blexer^{+}$.

For our proof we found it more helpful to introduce the rewriting systems shown in Fig 3. The idea is to generate simplified regular expressions in small steps (unlike the *bsimp*-function which does the same in a big step), and show that each of the small steps preserves the bitcodes that lead to the POSIX value. The rewrite system is organised such that $\rightsquigarrow$ is for bitcoded regular expressions and $\stackrel{s}{\rightsquigarrow}$ for lists of bitcoded regular expressions. The former essentially implements the simplifications of *bsimpSEQ* and *flts*; while the latter implements the simplifications in *bsimpALTs*. We can show that any bitcoded regular expression reduces in zero or more steps to the simplified regular expression generated by *bsimp*:

▶ **Lemma 11.** $r \rightsquigarrow^{*} bsimp\ r$

Proof. By induction on r. To establish the property we can use the properties $rs \stackrel{s}{\rightsquigarrow}^{*} flts\ rs$ and $rs \stackrel{s}{\rightsquigarrow}^{*} distinctWith\ rs \approx \varnothing$. ◀

We can also show that this rewrite system preserves *bnullable*, that is simplification does not affect nullability:

▶ **Lemma 12.** *If* $r_1 \rightsquigarrow r_2$ *then* $bnullable\ r_1 = bnullable\ r_2$.

Proof. Straightforward mutual induction on the definition of $\rightsquigarrow$ and $\stackrel{s}{\rightsquigarrow}$. The only interesting case is the rule LD where the property holds since by the side-conditions of that rule the empty string will be in both $L\ (rs_a\ @\ [r_1]\ @\ rs_b\ @\ [r_2]\ @\ rs_c)$ and $L\ (rs_a\ @\ [r_1]\ @\ rs_b\ @\ rs_c)$. ◀

From this, we can show that *bmkeps* will produce the same bitsequence as long as one of the bitcoded regular expressions in $\rightsquigarrow$ is nullable (this lemma establishes the missing fact we were not able to establish using *retrieve*, as suggested in the paper by Sulzmannn and Lu).

▶ **Lemma 13.** *If* $r_1 \rightsquigarrow r_2$ *and* $bnullable\ r_1 \wedge bnullable\ r_2$ *then* $bmkeps\ r_1 = bmkeps\ r_2$.

$$\frac{}{(SEQ\ bs\ ZERO\ r_2) \rightsquigarrow (ZERO)}S\mathbf{0}_l \quad \frac{}{(SEQ\ bs\ r_1\ ZERO) \rightsquigarrow (ZERO)}S\mathbf{0}_r \quad \frac{r_1 \rightsquigarrow r_2}{(SEQ\ bs\ r_1\ r_3) \rightsquigarrow (SEQ\ bs\ r_2\ r_3)}\text{SL}$$

$$\frac{}{(SEQ\ bs_1\ (ONE\ bs_2)\ r) \rightsquigarrow fuse\ (bs_1\ @\ bs_2)\ r}S\mathbf{1} \quad \frac{r_3 \rightsquigarrow r_4}{(SEQ\ bs\ r_1\ r_3) \rightsquigarrow (SEQ\ bs\ r_1\ r_4)}\text{SR}$$

$$\frac{}{(ALTs\ bs\ []) \rightsquigarrow (ZERO)}A0 \quad \frac{}{(ALTs\ bs\ [r]) \rightsquigarrow fuse\ bs\ r}A1 \quad \frac{rs_1 \overset{s}{\rightsquigarrow} rs_2}{(ALTs\ bs\ rs_1) \rightsquigarrow (ALTs\ bs\ rs_2)}AL$$

$$\frac{rs_1 \overset{s}{\rightsquigarrow} rs_2}{r :: rs_1 \overset{s}{\rightsquigarrow} r :: rs_2}LT \quad \frac{r_1 \rightsquigarrow r_2}{r_1 :: rs \overset{s}{\rightsquigarrow} r_2 :: rs}LH \quad \frac{}{ALTs\ bs\ rs_1 :: rs_2 \overset{s}{\rightsquigarrow} (map\ (fuse\ bs)\ rs_1\ @\ rs_2)}LS$$

$$\frac{}{ZERO :: rs \overset{s}{\rightsquigarrow} rs}L\mathbf{0} \quad \frac{L\ (r_2{}^{\downarrow}) \subseteq L\ (r_1{}^{\downarrow})}{(rs_1\ @\ [r_1]\ @\ rs_2\ @\ [r_2]\ @\ rs_3) \overset{s}{\rightsquigarrow} (rs_1\ @\ [r_1]\ @\ rs_2\ @\ rs_3)}LD$$

Figure 3 The rewrite rules that generate simplified regular expressions in small steps: $r_1 \rightsquigarrow r_2$ is for bitcoded regular expressions and $rs_1 \overset{s}{\rightsquigarrow} rs_2$ for *lists* of bitcoded expressions. Interesting is the LD rule that allows copies of regular expressions to be removed provided a regular expression earlier in the list can match the same strings.

Proof. By straightforward mutual induction on the definition of $\rightsquigarrow$ and $\overset{s}{\rightsquigarrow}$. Again the only interesting case is the rule LD where we need to ensure that $bmkeps\ (rs_a\ @\ [r_1]\ @\ rs_b\ @\ [r_2]\ @\ rs_c) = bmkeps\ (rs_a\ @\ [r_1]\ @\ rs_b\ @\ rs_c)$ holds. This is indeed the case because according to the POSIX rules the generated bitsequence is determined by the first alternative that can match the string (in this case being nullable). ◀

Crucial is also the fact that derivative steps and simplification steps can be interleaved, which is shown by the fact that $\rightsquigarrow$ is preserved under derivatives.

▶ **Lemma 14.** *If* $r_1 \rightsquigarrow r_2$ *then* $r_1\backslash c \rightsquigarrow^* r_2\backslash c$.

Proof. By straightforward mutual induction on the definition of $\rightsquigarrow$ and $\overset{s}{\rightsquigarrow}$. The case for LD holds because $L\ ((r_2\backslash c)^{\downarrow}) \subseteq L\ ((r_1\backslash c)^{\downarrow})$ if and only if $L\ (r_2{}^{\downarrow}) \subseteq L\ (r_1{}^{\downarrow})$. ◀

Using this fact together with Lemma 11 allows us to prove the central lemma that the unsimplified derivative (with a string s) reduces to the simplified derivative (with the same string).

▶ **Lemma 15.** $r\backslash s \rightsquigarrow^* r\backslash_{bsimp}\ s$

Proof. By reverse induction on s generalising over r. ◀

With these lemmas in place we can finally establish that $blexer^+$ and $blexer$ generate the same value, and using Theorem 9 from the previous section that this value is indeed the POSIX value as generated by *lexer*.

▶ **Theorem 16.** $blexer^+\ r\ s = blexer\ r\ s \quad (= lexer\ r\ s \ \ by\ Thm.\ 9)$

Proof. By unfolding the definitions and using Lemmas 15 and 13. ◀

This means that if the algorithm is called with a regular expression r and a string s with $s \in L(r)$, it will return *Some* v for the unique v we defined by the POSIX relation $(s, r) \rightarrow v$; otherwise the algorithm returns *None* when $s \notin L(r)$ and no such v exists. This completes the correctness proof for the second POSIX lexing algorithm by Sulzmann and Lu. The interesting point of this algorithm is that the sizes of derivatives do not grow arbitrarily big but can be finitely bounded, which we shall show next.

5 Finite Bound for the Size of Derivatives

In this section let us sketch our argument for why the size of the simplified derivatives with the aggressive simplification function can be finitely bounded. Suppose we have a size function for bitcoded regular expressions, written $[\![r]\!]$, which counts the number of nodes if we regard r as a tree (we omit the precise definition; ditto for lists $[\![rs]\!]$). For this we show that for every r there exists a bound N such that

$$\forall s.\ [\![r\backslash_{bsimp}\ s]\!] \leq N$$

Note that the bound N is a bound for *all* strings, no matter how long they are. We establish this bound by induction on r. The base cases for $ZERO$, $ONE\ bs$ and $CHAR\ bs\ c$ are straightforward. The interesting case is for sequences of the form $SEQ\ bs\ r_1\ r_2$. In this case our induction hypotheses state $\exists N_1.\forall s.\ [\![r_1\backslash_{bsimp}\ s]\!] \leq N_1$ and $\exists N_2.\forall s.\ [\![r_2\backslash_{bsimp}\ s]\!] \leq N_2$. We can reason as follows

$$\begin{array}{lll}
 & [\![(SEQ\ bs\ r_1\ r_2)\backslash_{bsimp}\ s]\!] & \\
= & [\![bsimp\,(ALTs\,bs\,((SEQ\ []\ (r_1\backslash_{bsimp}\ s)\ r_2) :: [r_2\backslash_{bsimp}\ s' \mid s' \in Suf(r_1, s)]))]\!] & (1)\\
\leq & [\![distinctWith\,((SEQ\ []\ (r_1\backslash_{bsimp}\ s)\ r_2) :: [r_2\backslash_{bsimp}\ s' \mid s' \in Suf(r_1, s)])\ \approx\ \varnothing]\!] + 1 & (2)\\
\leq & [\![SEQ\ []\ (r_1\backslash_{bsimp}\ s)\ r_2]\!] + [\![distinctWith\,[r_2\backslash_{bsimp}\ s' \mid s' \in Suf(r_1, s)]\ \approx\ \varnothing]\!] + 1 & (3)\\
\leq & N_1 + [\![r_2]\!] + 2 + [\![distinctWith\,[r_2\backslash_{bsimp}\ s' \mid s' \in Suf(r_1, s)]\ \approx\ \varnothing]\!] & (4)\\
\leq & N_1 + [\![r_2]\!] + 2 + l_{N_2} * N_2 & (5)
\end{array}$$

where in (1) the set $Suf(r_1, s)$ are all the suffixes of s where $r_1\backslash_{bsimp}\ s'$ is nullable (s' being a suffix of s). In (3) we know that $[\![SEQ\ []\ (r_1\backslash_{bsimp}\ s)\ r_2]\!]$ is bounded by $N_1 + [\![r_2]\!] + 1$. In (5) we know the list comprehension contains only regular expressions of size smaller than N_2. The list length after $distinctWith$ is bounded by a number, which we call l_{N_2}. It stands for the number of distinct regular expressions smaller than N_2 (there can only be finitely many of them). We reason similarly for $STAR$ and NT.

Clearly we give in this finiteness argument (Step (5)) a very loose bound that is far from the actual bound we can expect. We can do better than this, but this does not improve the finiteness property we are proving. If we are interested in a polynomial bound, one would hope to obtain a similar tight bound as for partial derivatives introduced by Antimirov [1]. After all the idea with $distinctWith$ is to maintain a "set" of alternatives (like the sets in partial derivatives). Unfortunately to obtain the exact same bound would mean we need to introduce simplifications, such as $(r_1 + r_2) \cdot r_3 \longrightarrow (r_1 \cdot r_3) + (r_2 \cdot r_3)$, which exist for partial derivatives. However, if we introduce them in our setting we would lose the POSIX property of our calculated values. For example given the regular expressions $(a + ab) \cdot (b + \mathbf{1})$ and the string $[a, b]$, then our algorithm generates the following correct POSIX value

$$Seq\ (Right\ (Seq\ (Char\ a)\ (Char\ b)))\ (Right\ Empty)$$

Essentially it matches the string with the longer $Right$-alternative in the first sequence (and then the "rest" with the empty regular expression $\mathbf{1}$ from the second sequence). If we add the simplification above, then we obtain the following value $Seq\ (Left\ (Char\ a))\ (Left\ (Char\ b))$ where the $Left$-alternatives get priority. However, this violates the POSIX rules and we have not been able to reconcile this problem. Therefore we leave better bounds for future work.

Note also that while Antimirov was able to give a bound on the *size* of his partial derivatives [1], Brzozowski gave a bound on the *number* of derivatives, provided they are quotient via ACI rules [4]. Brzozowski's result is crucial when one uses his derivatives for obtaining a DFA (it essentially bounds the number of states). However, this result does *not*

transfer to our setting where we are interested in the *size* of the derivatives. For example it is *not* true that the set of our derivatives $r\backslash s$ for a given r and all strings s is finite (even when simplified). This is because for our annotated regular expressions the bitcode annotation is dependent on the number of iterations that are needed for $STAR$-regular expressions. This is not a problem for us: Since we intend to do lexing by calculating (as fast as possible) derivatives, the bound on the size of the derivatives is important, not their number.

6 Conclusion

We set out in this work to prove in Isabelle/HOL the correctness of the second POSIX lexing algorithm by Sulzmann and Lu [15]. This follows earlier work where we established the correctness of the first algorithm [2]. In the earlier work we needed to introduce our own specification for POSIX values, because the informal definition given by Sulzmann and Lu did not stand up to formal proof. Also for the second algorithm we needed to introduce our own definitions and proof ideas in order to establish the correctness. Our interest in the second algorithm lies in the fact that by using bitcoded regular expressions and an aggressive simplification method there is a chance that the derivatives can be kept universally small (we established in this paper that for a given r they can be kept finitely bounded for *all* strings). Our formalisation is approximately 7500 lines of Isabelle code. A little more than half of this code concerns the finiteness bound obtained in Section 5. This slight "bloat" in the latter part is because we had to introduce an intermediate datatype for annotated regular expressions and repeat many definitions for this intermediate datatype. But overall we think this made our formalisation work smoother.

Having proved the correctness of the POSIX lexing algorithm, which lessons have we learned? Well, we feel this is a very good example where formal proofs give further insight into the matter at hand. For example it is very hard to see a problem with *nub* vs *distinctWith* with only experimental data – one would still see the correct result but find that simplification does not simplify in well-chosen, but not obscure, examples.

With the results reported here, we can of course only make a claim about the correctness of the algorithm and the sizes of the derivatives, not about the efficiency or runtime of our version of Sulzmann and Lu's algorithm. But we found the size is an important first indicator about efficiency: clearly if the derivatives can grow to arbitrarily big sizes and the algorithm needs to traverse the derivatives possibly several times, then the algorithm will be slow – excruciatingly slow that is. Other works seem to make stronger claims, but during our formalisation work we have developed a healthy suspicion when for example experimental data is used to back up efficiency claims. For instance Sulzmann and Lu write about their equivalent of $blexer^+$ *"...we can incrementally compute bitcoded parse trees in linear time in the size of the input"* [15, Page 14]. Given the growth of the derivatives in some cases even after aggressive simplification, this is a hard to believe claim. A similar claim about a theoretical runtime of $O(n^2)$ for one specific list of regular expressions is made for the Verbatim lexer, which calculates tokens according to POSIX rules [6]. For this, Verbatim uses Brzozowski's derivatives like in our work. About their empirical data, the authors write: *"The results of our empirical tests [..] confirm that Verbatim has $O(n^2)$ time complexity."* [6, Section VII]. While their correctness proof for Verbatim is formalised in Coq, the claim about the runtime complexity is only supported by some empirical evidence obtained by using the code extraction facilities of Coq. Given our observation with the "growth problem" of derivatives, this runtime bound is unlikely to hold universally: indeed we tried out their extracted OCaml code with the example $(a + aa)^*$ as a single lexing rule, and it took for

us around 5 minutes to tokenise a string of 40 a's and that increased to approximately 19 minutes when the string is 50 a's long. Taking into account that derivatives are not simplified in the Verbatim lexer, such numbers are not surprising. Clearly our result of having finite derivatives might sound rather weak in this context but we think such efficiency claims really require further scrutiny. The contribution of this paper is to make sure derivatives do not grow arbitrarily big (universally). In the example $(a + aa)^*$, *all* derivatives have a size of 17 or less. The result is that lexing a string of, say, 50 000 a's with the regular expression $(a + aa)^*$ takes approximately 10 seconds with our Scala implementation of the presented algorithm.[3]

Finally, let us come back to the point about bounded regular expressions. We have in this paper only shown that $r^{\{n\}}$ can be included, but all our results extend straightforwardly also to the other bounded regular expressions. We find bounded regular expressions fit naturally into the setting of Brzozowski derivatives and the bitcoded regular expressions by Sulzmann and Lu. In contrast bounded regular expressions are often the Achilles' heel in regular expression matchers that use the traditional automata-based approach to lexing, primarily because they need to expand the counters of bounded regular expressions into n-connected copies of an automaton. This is not needed in Sulzmann and Lu's algorithm. To see the difference consider for example the regular expression $a^{\{1001\}} \cdot a^*$, which is not permitted in the Go language because the counter is too big. In contrast we have no problem with matching this regular expression with, say 50 000 a's, because the counters can be kept compact. In fact, the overall size of the derivatives is never greater than 5 in this example. Even in the example from Section 2, where Rust raises an error message, namely $a^{\{1000\}\{100\}\{5\}}$, the maximum size for our derivatives is a moderate 14.

Let us also return to the example $a^{\{0\}\{4294967295\}}$ which until recently Rust deemed acceptable. But this was due to a bug. It turns out that it took Rust more than 11 minutes to generate an automaton for this regular expression and then to determine that a string of just one(!) a does *not* match this regular expression. Therefore it is probably a wise choice that in newer versions of Rust's regular expression library such regular expressions are declared as "too big" and raise an error message. While this is clearly a contrived example, the safety guaranties Rust wants to provide necessitate this conservative approach. However, with the derivatives and simplifications we have shown in this paper, the example can be solved with ease: it essentially only involves operations on integers and our Scala implementation takes only a few seconds to find out that this string, or even much larger strings, do not match.

Let us also compare our work to the verified Verbatim++ lexer where the authors of the Verbatim lexer introduced a number of improvements and optimisations, for example memoisation [7]. However, unlike Verbatim, which works with derivatives like in our work, Verbatim++ compiles first a regular expression into a DFA. While this makes lexing fast in many cases, with examples of bounded regular expressions like $a^{\{100\}\{5\}}$ one needs to represent them as sequences of $a \cdot a \cdot \ldots \cdot a$ (500 a's in sequence). We have run their extracted code with such a regular expression as a single lexing rule and a string of 50 000 a's – lexing in this case takes approximately 5 minutes. We are not aware of any better translation using the traditional notion of DFAs so that we can improve on this. Therefore we prefer to stick with calculating derivatives, but attempt to make this calculation (in the future) as fast as possible. What we can guarantee with the presented work is that the maximum size of the derivatives for $a^{\{100\}\{5\}} \cdot a^*$ is never bigger than 9. This means our Scala implementation again only needs a few seconds for this example and matching 50 000 a's, say.

[3] Our Scala implementation is "hand-crafted" and *not* generated via Isabelle's code extraction mechanism.

References

1 V. Antimirov. Partial Derivatives of Regular Expressions and Finite Automata Constructions. *Theoretical Computer Science*, 155:291–319, 1995.

2 F. Ausaf, R. Dyckhoff, and C. Urban. POSIX Lexing with Derivatives of Regular Expressions (Proof Pearl). In *Proc. of the 7th International Conference on Interactive Theorem Proving (ITP)*, volume 9807 of *LNCS*, pages 69–86, 2016.

3 H. Björklund, W. Martens, and T. Timm. Efficient Incremental Evaluation of Succinct Regular Expressions. In *Proc. of the 24th ACM Conf. on Information and Knowledge Management (CIKM)*, pages 1541–1550, 2015.

4 J. A. Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11(4):481–494, 1964.

5 T. Coquand and V. Siles. A Decision Procedure for Regular Expression Equivalence in Type Theory. In *Proc. of the 1st International Conference on Certified Programs and Proofs (CPP)*, volume 7086 of *LNCS*, pages 119–134, 2011.

6 D. Egolf, S. Lasser, and K. Fisher. Verbatim: A Verified Lexer Generator. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 92–100, 2021.

7 D. Egolf, S. Lasser, and K. Fisher. Verbatim++: Verified, Optimized, and Semantically Rich Lexing with Dderivatives. In *Proc. of the 11th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP)*, pages 27–39. ACM, 2022.

8 A. Krauss and T. Nipkow. Proof Pearl: Regular Expression Equivalence and Relation Algebra. *Journal of Automated Reasoning*, 49:95–106, 2012.

9 C. Kuklewicz. Regex Posix. `https://wiki.haskell.org/Regex_Posix`.

10 L. Nielsen and F. Henglein. Bit-Coded Regular Expression Parsing. In *Proc. of the 5th International Conference on Language and Automata Theory and Applications (LATA)*, volume 6638 of *LNCS*, pages 402–413, 2011.

11 S. Okui and T. Suzuki. Disambiguation in Regular Expression Matching via Position Automata with Augmented Transitions. In *Proc. of the 15th International Conference on Implementation and Application of Automata (CIAA)*, volume 6482 of *LNCS*, pages 231–240, 2010.

12 S. Owens and K. Slind. Adapting Functional Programs to Higher Order Logic. *Higher-Order and Symbolic Computation*, 21(4):377–409, 2008.

13 The Open Group Base Specification Issue 6 IEEE Std 1003.1 2004 Edition, 2004. URL: `http://pubs.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html`.

14 R. Ribeiro and A. Du Bois. Certified Bit-Coded Regular Expression Parsing. In *Proc. of the 21st Brazilian Symposium on Programming Languages*, pages 4:1–4:8, 2017.

15 M. Sulzmann and K. Lu. POSIX Regular Expression Parsing with Derivatives. In *Proc. of the 12th International Conference on Functional and Logic Programming (FLOPS)*, volume 8475 of *LNCS*, pages 203–220, 2014.

16 L. Turoňová, L. Holík, O. Lengál, O. Saarikivi, M. Veanes, and T. Vojnar. Regex Matching with Counting-Set Automata. *Proceedings of the ACM on Programming Languages (PACMPL)*, 4:218:1–218:30, 2020.

17 S. Vansummeren. Type Inference for Unique Pattern Matching. *ACM Transactions on Programming Languages and Systems*, 28(3):389–428, 2006.

A Sound and Complete Projection for Global Types

Dawit Tirore
IT University of Copenhagen, Denmark

Jesper Bengtson
IT University of Copenhagen, Denmark

Marco Carbone
IT University of Copenhagen, Denmark

Abstract

Multiparty session types is a typing discipline used to write specifications, known as global types, for branching and recursive message-passing systems. A necessary operation on global types is projection to abstractions of local behaviour, called local types. Typically, this is a computable partial function that given a global type and a role erases all details irrelevant to this role.

Computable projection functions in the literature are either unsound or too restrictive when dealing with recursion and branching. Recent work has taken a more general approach to projection defining it as a coinductive, but not computable, relation. Our work defines a new computable projection function that is sound and complete with respect to its coinductive counterpart and, hence, equally expressive. All results have been mechanised in the Coq proof assistant.

2012 ACM Subject Classification Theory of computation → Type theory; Computing methodologies → Distributed programming languages; Theory of computation → Program verification

Keywords and phrases Session types, Mechanisation, Projection, Coq

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.28

Supplementary Material *Software (Coq Source Code)*: `https://github.com/Tirore96/projection` archived at `swh:1:dir:c4f38245d064fc65ed000d62ebd15826ef8da337`

Funding This work was supported by DFF-Research Project 1 Grant no. 1026-00385A, from The Danish Council for Independent Research for the Natural Sciences (FNU).

1 Introduction

Session types are types for abstracting the behaviour of communicating processes. First proposed by Honda et al. [15] for binary protocols, they specify the sequence of possible actions processes need to follow when sending and receiving messages over a channel. Session types provide a clear language for describing protocols that are guaranteed to not deadlock or contain communication errors, e.g., never receive an integer when expecting a boolean. A decade after their conception, Honda et al. [16] proposed a generalisation, called *multiparty session types*, that specifies how an arbitrary but fixed number of processes should interact with each other. Multiparty session types are based on the concept of *global types* which provide a global description of the multiparty protocol being abstracted. Recently, multiparty session types have gained interest from several communities, resulting in their integration into several mainstream programming languages [2].

Multiparty session types follow a precise approach to designing and implementing communicating processes: from global types that specify the protocols, we can automatically generate *local types*, the local specifications of the behaviour of each *role* in the protocol; then, each local type specification is (type) checked against the local code being written by the programmer. The automatic generation of local types from global types, called *projection*, is key for relating global types to implementations. Given a role, projection is an operation that erases the parts of the global type irrelevant for the role. When projection is defined the


© Dawit Tirore, Jesper Bengtson, and Marco Carbone;
licensed under Creative Commons License CC-BY 4.0

output is a local type specifying the behaviour of this role. As an example, let us consider a global type where Carl can ask Dave to either go `Left` or `Right` over some channel k:

$$\mathsf{Carl} \to \mathsf{Dave} : k \left\{ \begin{array}{l} \texttt{Left} : \ \mathsf{Carl} \to \mathsf{Dave} : k'\langle \texttt{Int} \rangle.\mathsf{Alice} \to \mathsf{Bob} : k''\langle \texttt{Int} \rangle.\mathsf{end} \\ \texttt{Right} : \ \mathsf{Carl} \to \mathsf{Dave} : k'\langle \texttt{String} \rangle.\mathsf{Alice} \to \mathsf{Bob} : k''\langle \texttt{Int} \rangle.\mathsf{end} \end{array} \right\}$$

Above, if Carl chooses `Left`, he will also send an integer (`Int`) over some other channel k'; otherwise, he will send a string (`String`). No matter what branch Carl chooses, all roles must collectively follow the description of that branch.

Nested in both branches, there is a communication over k'' of an integer `Int` between Alice and Bob. The projections of Carl and Alice are:

$$\mathsf{Carl} : \quad k \oplus \left\{ \begin{array}{l} \texttt{Left} : !k'\langle \texttt{Int} \rangle.\mathsf{end} \\ \texttt{Right} : !k'\langle \texttt{String} \rangle.\mathsf{end} \end{array} \right\} \qquad\qquad \mathsf{Alice} : \quad !k''\langle \texttt{Int} \rangle.\ \mathsf{end}$$

Above, Carl makes a choice (denoted by $\oplus$), and then outputs on channel k' either something of type `Int` or something of type `String`. Alice is instead sending over channel k''. An important observation is that, since neither Alice nor Bob are informed of the choice made by Carl, their behaviour should be independent from Carl's choice. In fact, a *restriction* that projection usually imposes is that all those roles not participating to a branching communication behave the same on all branches.

In order to be able to express repetitive behaviour, global types (and local types) are usually equipped with recursion, expressed as *μ-types* [22]. For example, consider

$$\mu \mathbf{t}.\ \mathsf{Alice} \to \mathsf{Bob} : k\langle \texttt{String} \rangle.\ \mu \mathbf{t}'.\ \mathsf{Carl} \to \mathsf{Dave} : k' \left\{ \begin{array}{l} \texttt{Left} : \ \mathbf{t} \\ \texttt{Right} : \ \mathsf{Alice} \to \mathsf{Bob} : k\langle \texttt{String} \rangle.\mathbf{t}' \end{array} \right\} \quad (1)$$

The example above poses some questions on how projection should work. For Alice, should it be undefined since we cannot syntactically see her behaviour on the first branch? Or, can the projection first unfold on $\mathbf{t}$ and then generate a local type? We observe that the following global type is equivalent to (1) but does not violate our constraint on branches:

$$\mu \mathbf{t}.\ \mathsf{Alice} \to \mathsf{Bob} : k\langle \texttt{String} \rangle.\ \mathsf{Carl} \to \mathsf{Dave} : k' \{ \texttt{Left} : \ \mathbf{t}, \quad \texttt{Right} : \ \mathbf{t} \} \quad (2)$$

Since both recursive global types (1) and (2) seem to specify the same behaviour, we would assume that Alice is projected to $\mu \mathbf{t}.\ !k\langle \texttt{String} \rangle.\ \mathbf{t}$, i.e., she repeatedly sends something of type `String`. The bad news is that the projection algorithms available in the literature do not allow global types like (1) to be projected while the equivalent type (2) can be projected.

The most common way of defining projection is as a structurally recursive partial function on global types, which we call *standard projection*. Recent work [13] defines projection as a coinductive relation on coinductive types, which intuitively are a complete (possibly infinite) unfolding of recursive protocols. Both approaches come with trade-offs. Standard projection is a computable procedure, which is necessary for multiparty session types to support decidable type checking, but it has limits as pointed out above. The coinductive approach is more general but, to the best of our knowledge, there are no equivalent computable algorithms available in the literature. The discrepancy between standard and coinductive projection was initially pointed out by Ghilezan et al. [13]. They correctly show that the canonical way partial projection treats the binders of μ-types causes standard projection to be undefined for some μ-types that have a coinductive projection, such as global type (1). In this paper, we define a procedure on μ-types that implements the projection on coinductive types.

Contributions and Structure. The main contribution of this paper is the definition of a computable projection function that is sound and complete with respect to a coinductive projection relation. All our proofs have been mechanised in the Coq [21] proof assistant[1].

We structure the paper as follows. Section 2 walks through existing variants of standard projection and their pitfalls. Section 3 introduces global and local coinductive types as well as a coinductive projection relation from the former to the latter. Section 4 introduces a projection function from global to local μ-types, proves that it is sound and complete with respect to its coinductive counterpart, and Section 5 proves that it is decidable. Section 6 describes key insights from our Coq mechanisation, Section 7 covers related and future work, and Section 8 concludes.

2 Global Types, Local Types, and Standard Projection

The purpose of this section is two-fold: introducing the syntax of global and local types and a walk through computable definitions of projection found in the literature.

Syntax. Let $\mathcal{P}$ be a set of roles (ranged over by $\mathsf{p}, \mathsf{q}, \mathsf{r}, \mathsf{s}, \mathsf{t}$), $\mathcal{L}$ a totally ordered set of labels (ranged over by l), and $\mathcal{X}$ a set of recursion variables ranged over by $\mathbf{t}$.

▶ **Definition 1** (Inductive Types [17]). *Global types G^μ and local types T^μ are μ-types generated inductively by the following grammars, where U represents primitive types:*

$$G^\mu ::= \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu \mid \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G_j^\mu\}_{j\in J} \mid \mu\mathbf{t}.G^\mu \mid \mathbf{t} \mid \mathsf{end}^\mu$$
$$T^\mu ::= !^\mu k\langle U\rangle.T^\mu \mid ?^\mu k\langle U\rangle.T^\mu \mid k \oplus^\mu \{l_j : T_j^\mu\}_{j\in J} \mid k \,\&^\mu\, \{l_j : T_j^\mu\}_{j\in J} \mid \mu\mathbf{t}.T^\mu \mid \mathbf{t} \mid \mathsf{end}^\mu$$

The type $\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu$ denotes a communication between roles p_1 and p_2 via channel k of a message of type U, which then proceeds as G^μ. Similarly, the type $\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G_j^\mu\}_{j\in J}$ denotes a communication between two roles where, given the set of indices J, role p_1 selects a branch with label l_i, and then proceeds as G_i^μ. Types $\mu\mathbf{t}.G^\mu$ and $\mathbf{t}$ model recursive protocols. Finally, end^μ denotes the successful termination of a protocol. A message type U is just a basic value type: extensions of this are irrelevant for the focus of this paper.

For local types, the type $!^\mu k\langle U\rangle.T^\mu$ outputs a message of type U over channel k, while its dual, $?^\mu k\langle U\rangle.T^\mu$ receives a message of type U over k. Types $k \oplus^\mu \{l_j : T_j^\mu\}_{j\in J}$ and $k \,\&^\mu\, \{l_j : T_j^\mu\}_{j\in J}$ implement branching where the former is the type of a process that internally selects a branch l_i and communicates it over channel k, while the latter is the type of a process that offers choices $l_1, \ldots, l_n$ (for $J = \{1, \ldots, n\}$ with $n \geq 1$) over channel k. We overload the type end^μ and use it also for local types.

We deal with recursive variables in a standard way and write capture-avoiding substitution as $G_1^\mu[G_2^\mu/\mathbf{t}]$. Moreover, types can be contractive: a μ-type G^μ (or T^μ) is contractive if, for any of its subexpressions with shape $\mu\mathbf{t}_0.\mu\mathbf{t}_1 ... \mu\mathbf{t}_n.\mathbf{t}$, the body $\mathbf{t}$ is not $\mathbf{t}_0$ [22]. We allow non-contractive μ-types and will in the next section show how to enforce contractiveness by requiring that a μ-type is related to a coinductive type.

Overview of projections. For each role, we use projection to relate global and local types. We start our overview with the projection proposed by Castro-Perez et al. [7] which can be found in Figure 1. The projection $\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.\ G^\mu \upharpoonright^\mu_{\mathsf{p}}$ produces either a sending or a

[1] https://github.com/Tirore96/projection

$$(\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu)\upharpoonright^\mu_{\mathsf{p}} = \begin{cases} !^\mu k\langle U\rangle.(G^\mu\upharpoonright^\mu_{\mathsf{p}}) & \text{if } \mathsf{p}=\mathsf{p}_1 \text{ and } \mathsf{p}_1\neq\mathsf{p}_2 \\ ?^\mu k\langle U\rangle.(G^\mu\upharpoonright^\mu_{\mathsf{p}}) & \text{if } \mathsf{p}=\mathsf{p}_2 \text{ and } \mathsf{p}_1\neq\mathsf{p}_2 \\ G^\mu\upharpoonright^\mu_{\mathsf{p}} & \text{if } \mathsf{p}\notin\{\mathsf{p}_1,\mathsf{p}_2\} \end{cases}$$

$$(\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G^\mu_j\}_{j\in J})\upharpoonright^\mu_{\mathsf{p}} = \begin{cases} k\oplus^\mu\{l_j : (G^\mu_j\upharpoonright^\mu_{\mathsf{p}})\}_{j\in J} & \text{if } \mathsf{p}=\mathsf{p}_1 \text{ and } \mathsf{p}_1\neq\mathsf{p}_2 \\ k\,\&^\mu\{l_j : (G^\mu_j\upharpoonright^\mu_{\mathsf{p}})\}_{j\in J} & \text{if } \mathsf{p}=\mathsf{p}_2 \text{ and } \mathsf{p}_1\neq\mathsf{p}_2 \\ (G^\mu_1\upharpoonright^\mu_{\mathsf{p}}) & \text{if } \mathsf{p}\notin\{\mathsf{p}_1,\mathsf{p}_2\} \text{ and } \forall i,j\in J.\ G^\mu_i\upharpoonright^\mu_{\mathsf{p}} = G^\mu_j\upharpoonright^\mu_{\mathsf{p}} \\ \bot & \textit{otherwise} \end{cases}$$

$$(\mu\mathbf{t}.G^\mu)\upharpoonright^\mu_{\mathsf{p}} = \begin{cases} \mu\mathbf{t}.(G^\mu\upharpoonright^\mu_{\mathsf{p}}) & \text{if } \mathsf{guardedVar}(\mathbf{t}, G^\mu\upharpoonright^\mu_{\mathsf{p}}) \\ \mathsf{end}^\mu & \text{otherwise} \end{cases} \qquad \mathbf{t}\upharpoonright^\mu_{\mathsf{p}} = \mathbf{t} \qquad \mathsf{end}^\mu\upharpoonright^\mu_{\mathsf{p}} = \mathsf{end}^\mu.$$

$$\mathsf{guardedVar}(\mathbf{t}, G^\mu) = \begin{cases} \mathsf{guardedVar}(\mathbf{t}, G^\mu_1) & \text{if } G^\mu = \mu\mathbf{t}'.G^\mu_1 \\ \mathbf{t}\neq\mathbf{t}' & \text{if } G^\mu = \mathbf{t}' \\ \mathsf{true} & \text{otherwise} \end{cases}$$

Figure 1 The *standard projection of G onto* p, written $G\upharpoonright^\mu_{\mathsf{p}}$ [7].

receiving action if the role p is equal to p_1 or p_2 respectively, otherwise the action is deleted. The projection of branching $\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G^\mu_j\}_{j\in J}\upharpoonright^\mu_{\mathsf{p}}$ works similarly but, when role p is not involved, all branches must project to exactly the same type. This requirement is known as *plain merge*. *Full merge*, used for example by Ghilezan et al. [13], is a more permissive operation which merges local types with distinct external choices. We discuss an extension of our work to full merge in Section 7. For recursion $\mu\mathbf{t}.G^\mu$, G^μ is projected only if the result is a contractive local type (checked by the $\mathsf{guardedVar}$ predicate). Finally, variable $\mathbf{t}$ and the type end^μ project directly to their local counterparts.

The use of $\mathsf{guardedVar}$ formally fixes a problem with the original projection [17] that could generate non-contractive types, which is unsound (informally fixed by forbidding non-contractive types). Alternatively, Demangeon and Yoshida [11] fix this issue by replacing the side condition with $G^\mu\upharpoonright^\mu_{\mathsf{p}} \neq \mathbf{t}$. However, all these projections invite the counterexample:

$$\mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mu\mathbf{t}.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\{l_1 : \mathsf{end}^\mu,\ l_2 : \mathbf{t}\}\upharpoonright^\mu_{\mathsf{p}} \tag{3}$$

which is undefined because the branch condition fails. Since p is not a role in the branch, the desired result of this projection should be $!^\mu k\langle U\rangle.\ \mathsf{end}^\mu$. Bejleri and Yoshida [5] solve this with a recursion condition testing participation in the body

$$(\mu\mathbf{t}.G^\mu)\upharpoonright^\mu_{\mathsf{p}} = \begin{cases} \mu\mathbf{t}.(G^\mu\upharpoonright^\mu_{\mathsf{p}}) & \text{if } \mathsf{p}\in G^\mu \\ \mathsf{end}^\mu \end{cases} \tag{4}$$

This function always generates contractive types, but the projection of

$$\mu\mathbf{t}.\ \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mu\mathbf{t}'.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\langle U'\rangle.\ \mathbf{t}\upharpoonright^\mu_{\mathsf{p}} \tag{5}$$

incorrectly results in the local type $\mu\mathbf{t}.!^\mu k\langle U\rangle.\mathsf{end}^\mu$ rather than the desired $\mu\mathbf{t}.!^\mu k\langle U\rangle.\ \mathbf{t}$. Glabbeek et al. [27] fix it by adding a variable constraint to the recursion condition:

$$(\mu\mathbf{t}.G^\mu)\upharpoonright^\mu_{\mathsf{p}} = \begin{cases} \mu\mathbf{t}.(G^\mu\upharpoonright^\mu_{\mathsf{p}}) & \\ \mathsf{end}^\mu & \text{if } \mathsf{p}\notin G^\mu \text{ and } \mu\mathbf{t}.G^\mu \text{ is closed} \end{cases} \tag{6}$$

This way, the projection in (5) correctly results in the type $\mu\mathbf{t}.!^{\mu}k\langle U\rangle.\mathbf{t}$. To the best of our knowledge, this is the most general and sound version of projection, but it still does not capture certain global types whose infinite unfolding is intuitively projectable. One such example is equivalent to (1), modulo renaming, from the introduction:

$$\mu\mathbf{t}.\ \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mu\mathbf{t}'.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\{l_1 : \mathbf{t},\quad l_2 : \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mathbf{t}'\} \upharpoonright^{\mu}_{\mathsf{p}} \tag{7}$$

Here, the branching condition fails because $\mathbf{t}$ is syntactically not the same type as $!^{\mu}k\langle U\rangle.\mathbf{t}'$. But how can we recognise that $\mathbf{t}$ and $!^{\mu}k\langle U\rangle.\mathbf{t}'$ are equivalent in this case? Our main insight is that standard projection can be performed in two steps: first, a boolean predicate tests projectability by unfolding μ-operators; and, when the check is passed, a translation function generates the local type by instead structurally recursing under the μ-operators. Checking projectability by unfolding μ-operators makes termination non-trivial and we explore this in Section 5. This approach lets us recognise (7) as projectable.

3 Projection on Coinductive Types

In this section, we define what an ideal projection is. The inductive definition of global types uses μ-types in order to represent infinite behaviour which, as shown by our examples, can create issues with projection. A possible solution to this issue is to get rid of μ-types and work with fully unfolded types (infinite trees). Originally, Honda et al. [17] suggested this approach informally. Later, Ghilezan et al. [13] turned this intuition into a version of global types which, instead of using an inductive definition, uses coinductive types. This had the drawback of projection not being computable. The goal of this section is to define coinductive types, a way to relate them to inductive types, and then a definition of projection without μ-types. Although we do not compute projections of coinductive types, we use them as a specification of how a correct projection should behave.

Syntax. We start by giving the coinductive definition of both global and local types.

▶ **Definition 2** (Coinductive Types). *The syntax of coinductive global and local types, denoted as G^{ν} and T^{ν} respectively, is coinductively defined as:*

$$\begin{aligned} G^{\nu} &::= \mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\langle U\rangle.G^{\nu} \mid \mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\{l_j : G^{\nu}_j\}_{j\in J} \mid \mathsf{end}^{\nu} \\ T^{\nu} &::= !^{\nu}k\langle U\rangle.T^{\nu} \mid ?^{\nu}k\langle U\rangle.T^{\nu} \mid k \oplus^{\nu} \{l_j : T^{\nu}_j\}_{j\in J} \mid k \,\&^{\nu}\, \{l_j : T^{\nu}_j\}_{j\in J} \mid \mathsf{end}^{\nu} \end{aligned}$$

Coinductive types can be infinite but regular coinductive types can be finitely represented. A regular coinductive type has a finite set of distinct subterms [20] meaning that it must be circularly defined and have repeating structure if it is infinitely large. This makes it possible to store a regular coinductive type in, e.g., computer memory, or represent it as a μ-type.

In order to reason effectively about μ-types and their coinductive counterparts we need a means to relate the two. We follow the style of Castro-Perez et al. [7], using an unravelling relation $\mathcal{R}$, formally defined as:

▶ **Definition 3** (Unravelling). Unravelling, *for both global and local types, and denoted by* $G^\mu \mathrel{\mathcal{R}} G^\nu$ *and* $T^\mu \mathrel{\mathcal{R}} T^\nu$ *respectively, is defined by the following rules:*

$$\frac{}{\mathsf{end}^\mu \mathrel{\mathcal{R}} \mathsf{end}^\nu} \qquad \frac{G^\mu[\mu \mathbf{t}.G^\mu/\mathbf{t}] \mathrel{\mathcal{R}} G^\nu}{\mu \mathbf{t}.G^\mu \mathrel{\mathcal{R}} G^\nu} \qquad \frac{G^\mu \mathrel{\mathcal{R}} G^\nu}{\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu \mathrel{\mathcal{R}} \mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\langle U\rangle.G^\nu}$$

$$\frac{\forall j \in J.\ G_j^\mu \mathrel{\mathcal{R}} G_j^\nu}{\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G_j^\mu\}_{j\in J} \mathrel{\mathcal{R}} \mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\{l_j : G_j^\nu\}_{j\in J}} \qquad \frac{\forall j \in J.\ T_j^\mu \mathrel{\mathcal{R}} T_j^\nu}{k \oplus^\mu \{l_j : T_j^\mu\}_{j\in J} \mathrel{\mathcal{R}} k \oplus^\nu \{l_j : T_j^\nu\}_{j\in J}}$$

$$\frac{T^\mu \mathrel{\mathcal{R}} T^\nu}{!^\mu k\langle U\rangle.T^\mu \mathrel{\mathcal{R}} !^\nu k\langle U\rangle.T^\nu} \qquad \frac{T^\mu \mathrel{\mathcal{R}} T^\nu}{?^\mu k\langle U\rangle.T^\mu \mathrel{\mathcal{R}} ?^\nu k\langle U\rangle.T^\nu} \qquad \frac{\forall j \in J.\ (T_j^\mu \mathrel{\mathcal{R}} T_j^\nu)}{k \,\&^\mu \{l_j : T_j^\mu\}_{j\in J} \mathrel{\mathcal{R}} k \,\&^\nu \{l_j : T_j^\nu\}_{j\in J}}$$

The unravelling relation is defined using both inductive and coinductive inference rules, where we use single lines for inductive rules and double lines for coinductive ones. A coinductive derivation may be circular and discharged by referring to a previous identical part of the inference tree whereas inductive leaves are discharged using an inductive base-case rule in the standard manner, which in our case are the rules relating end^μ and end^ν. The reason for this split is that if the μ-operator could be unravelled using a coinductive rule [unfold^ν] then we could relate any non-contractive μ-type to any coinductive type G^ν.

$$\text{Incorrect rule: } \frac{G[\mu \mathbf{t}.G] \mathrel{\mathcal{R}} G^\nu}{\mu \mathbf{t}.G \mathrel{\mathcal{R}} G^\nu}\ [\mathsf{unfold}^\nu] \qquad \text{Unwanted derivation: } \frac{\dfrac{\dfrac{}{\mu \mathbf{t}.t \mathrel{\mathcal{R}} G^\nu}}{\mu \mathbf{t}.t \mathrel{\mathcal{R}} G^\nu}\ [\mathsf{unfold}^\nu]}{} \tag{8}$$

Castro-Perez et al. have a rule like [unfold^ν] and they solve this problem by requiring that all μ-types are contractive. We make this side condition redundant by making [unfold^ν] inductive and we have found that this simplifies our proofs. This is because the usual two conditions on μ-types, namely closedness and contractiveness, are captured by unravelling.

▶ **Proposition 4.** *G^μ is closed and contractive iff there exists G^ν such that $G^\mu \mathrel{\mathcal{R}} G^\nu$*

Proof. The direction ($\Longleftarrow$) is harder than the other. We prove it by contradiction, assuming both an inductive definition of non-contractiveness and $G^\mu \mathrel{\mathcal{R}} G^\nu$. ◀

The mixing of inductive and coinductive inference rules is non-standard and in Section 6 we show concretely how to formally define such inference systems. For now, we show an example of how to relate an inductive and coinductive global type by $\mathcal{R}$.

▶ **Example 5.** Consider the unravelling of

$$\mu \mathbf{t}.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k\{l_1 : \mathbf{t},\ l_2 : \mathsf{end}^\mu\}$$

One branch is a recursion variable and the other is end indicating we will need both inductive and coinductive rules to close the derivation. We show this inductive global type unravels to

$$G^\nu := \mathsf{r} \xrightarrow{\nu} \mathsf{s} : k\{l_1 : G^\nu,\ l_2 : \mathsf{end}^\nu\}$$

This is shown by the derivation below

$$\cfrac{\cfrac{\cfrac{}{\mu \mathbf{t}.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k\{l_1 : \mathbf{t},\ l_2 : \mathsf{end}^\mu\} \mathrel{\mathcal{R}} G^\nu} \qquad \cfrac{}{\mathsf{end}^\mu \mathrel{\mathcal{R}} \mathsf{end}^\nu}}{\mathsf{r} \xrightarrow{\mu} \mathsf{s} : k\{l_1 : \mu \mathbf{t}.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k\{l_1 : \mathbf{t},\ l_2 : \mathsf{end}^\mu\},\ l_2 : \mathsf{end}^\mu\} \mathrel{\mathcal{R}} G^\nu}}{\longrightarrow \mu \mathbf{t}.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k\{l_1 : \mathbf{t},\ l_2 : \mathsf{end}^\mu\} \mathrel{\mathcal{R}} G^\nu} \tag{9}$$

where the arrow marks the cycle that solves the coinductive part of the proof. Visually, the arrow must pass a double line for the proof to be valid.

$$\frac{\mathsf{p} \in \{\mathsf{p}_1, \mathsf{p}_2\} \vee \mathsf{guarded}^{\nu}_{\mathsf{p}}(G^{\nu})}{\mathsf{guarded}^{\nu}_{\mathsf{p}}(\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\langle U\rangle.G^{\nu})} \qquad \frac{\mathsf{p} \in \{\mathsf{p}_1, \mathsf{p}_2\} \vee \forall j.\ \mathsf{guarded}^{\nu}_{\mathsf{p}}(G^{\nu}_j)}{\mathsf{guarded}^{\nu}_{\mathsf{p}}(\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\{l_j : G^{\nu}_j\}_{j\in J})} \qquad \frac{\mathsf{guarded}^{\mu}_{\mathsf{p}}(G[\mu\mathbf{t}.G/t])}{\mathsf{guarded}^{\mu}_{\mathsf{p}}(\mu\mathbf{t}.G)}$$

$$\frac{\mathsf{p} \in \{\mathsf{p}_1, \mathsf{p}_2\} \vee \mathsf{partOf}^{\nu}_{\mathsf{p}}(G^{\nu})}{\mathsf{partOf}^{\nu}_{\mathsf{p}}(\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\langle U\rangle.G^{\nu})} \qquad \frac{\mathsf{p} \in \{\mathsf{p}_1, \mathsf{p}_2\} \vee \exists j \in J.\ \mathsf{partOf}^{\nu}_{\mathsf{p}}(G^{\nu}_j)}{\mathsf{partOf}^{\nu}_{\mathsf{p}}(\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\{l_j : G^{\nu}_j\}_{j\in J})} \qquad \frac{\mathsf{partOf}^{\mu}_{\mathsf{p}}(G[\mu\mathbf{t}.G/t])}{\mathsf{partOf}^{\mu}_{\mathsf{p}}(\mu\mathbf{t}.G)}$$

Figure 2 Definitions of predicates $\mathsf{guarded}^{\nu}$, $\mathsf{guarded}^{\mu}$, partOf^{ν}, and partOf^{μ}. The $\mathsf{guarded}^{\mu}$ and partOf^{μ} predicates additionally have identical rules to their $\mathsf{guarded}^{\nu}$ and partOf^{ν} counterparts, except for being defined for G^{μ} and not G^{ν} – these rules have been elided.

$$\frac{G^{\nu} \upharpoonright^{\nu}_{\mathsf{p}} T^{\nu}}{\mathsf{p} \xrightarrow{\nu} \mathsf{p}_2 : k\langle U\rangle.G^{\nu} \upharpoonright^{\nu}_{\mathsf{p}} !^{\nu}k\langle U\rangle.T^{\nu}}\ [\mathsf{M1}\upharpoonright^{\nu}] \qquad \frac{\mathsf{p} \neq \mathsf{p}_1 \quad G^{\nu} \upharpoonright^{\nu}_{\mathsf{p}} T^{\nu}}{\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p} : k\langle u\rangle.G^{\nu} \upharpoonright^{\nu}_{\mathsf{p}} ?^{\nu}k\langle U\rangle.T^{\nu}}\ [\mathsf{M2}\upharpoonright^{\nu}] \qquad \frac{\neg\mathsf{partOf}^{\nu}_{\mathsf{p}}(G^{\nu})}{G^{\nu} \upharpoonright^{\nu}_{\mathsf{p}} \mathsf{end}^{\nu}}\ [\mathsf{End}\upharpoonright^{\nu}]$$

$$\frac{\mathsf{p} \notin \{\mathsf{p}_1, \mathsf{p}_2\} \qquad \mathsf{guarded}^{\nu}_{\mathsf{p}}(G^{\nu}) \qquad G^{\nu} \upharpoonright^{\nu}_{\mathsf{p}} T^{\nu}}{\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\langle U\rangle.G^{\nu} \upharpoonright^{\nu}_{\mathsf{p}} T^{\nu}}\ [\mathsf{M}\upharpoonright^{\nu}] \qquad \frac{\forall j.\ G^{\nu}_j \upharpoonright^{\nu}_{\mathsf{p}} T^{\nu}_j}{\mathsf{p} \xrightarrow{\nu} \mathsf{p}_2 : k\{l_j : G^{\nu}_j\}_{j\in J} \upharpoonright^{\nu}_{\mathsf{p}} k \oplus^{\nu} \{l_j : T^{\nu}_j\}_{j\in J}}\ [\mathsf{B1}\upharpoonright^{\nu}]$$

$$\frac{J \neq \{\} \qquad \mathsf{p} \notin \{\mathsf{p}_1, \mathsf{p}_2\} \qquad \forall j.\ G^{\nu}_j \upharpoonright^{\nu}_{\mathsf{p}} T^{\nu} \wedge \mathsf{guarded}^{\nu}_{\mathsf{p}}(G^{\nu}_j)}{\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\{l_j : G^{\nu}_j\}_{j\in J} \upharpoonright^{\nu}_{\mathsf{p}} T^{\nu}}\ [\mathsf{B}\upharpoonright^{\nu}] \qquad \frac{\mathsf{p} \neq \mathsf{p}_1 \qquad \forall j,\ G^{\nu}_j \upharpoonright^{\nu}_{\mathsf{p}} T^{\nu}_j}{\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p} : k\{l_j : G^{\nu}_j\}_{j\in J} \upharpoonright^{\nu}_{\mathsf{p}} k \&^{\nu} \{l_j : T^{\nu}_j\}_{j\in J}}\ [\mathsf{B2}\upharpoonright^{\nu}]$$

Figure 3 The projection on coinductive types, denoted $G^{\nu} \upharpoonright^{\nu}_{\mathsf{p}} T^{\nu}$, is defined by coinductive rules.

In order to define projection from coinductive global types to coinductive local types, we require the two auxiliary predicates $\mathsf{guarded}^{\nu}_{\mathsf{p}}(G^{\nu})$ and $\mathsf{partOf}^{\nu}_{\mathsf{p}}(G^{\nu})$. The former asserts that p appears in all branches of G^{ν} at finite depth, and the latter asserts that p occurs somewhere in G^{ν} at finite depth. To reason about finite depth these predicates are inductively defined. We also define similar predicates $\mathsf{guarded}^{\mu}_{\mathsf{p}}(G^{\mu})$ and $\mathsf{partOf}^{\mu}_{\mathsf{p}}(G^{\mu})$ for inductive global types G^{μ}. All four predicates are defined in Fig. 2.

The rules for projection are presented in Figure 3. Rules $[\mathsf{M1}\upharpoonright^{\nu}]$, $[\mathsf{M2}\upharpoonright^{\nu}]$, $[\mathsf{B1}\upharpoonright^{\nu}]$, and $[\mathsf{B2}\upharpoonright^{\nu}]$ handle the cases where a projected role p takes part in communication or branching. Note that our projection allows sender and receiver in a communication to be equal. This case is a special case of rule $[\mathsf{M1}\upharpoonright^{\nu}]$. The rules $[\mathsf{M}\upharpoonright^{\nu}]$, $[\mathsf{B}\upharpoonright^{\nu}]$, and $[\mathsf{End}\upharpoonright^{\nu}]$ handle the cases where p does not take part. In these cases, in order for projection to continue, p must occur in all possible future branches, otherwise the projection maps to end. These rules are similar to those given by Castro-Perez et al. [7] as well as Jacobs et al. [19].

Guardedness, enforced by the predicate $\mathsf{guarded}^{\nu}_{\mathsf{p}}$ in the $[\mathsf{M}\upharpoonright^{\nu}]$ and $[\mathsf{B}\upharpoonright^{\nu}]$ rules, is necessary in order to avoid unwanted derivations similar to that for unravelling in (8).

▶ **Example 6.** We can now use unravelling and coinductive projection to relate the global type $\mu\mathbf{t}.\ \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mu\mathbf{t}'.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\{l_1 : \mathbf{t}, \quad l_2 : \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mathbf{t}'\}$ seen in (7) with $\mu\mathbf{t}.!^{\mu}k\langle U\rangle.\mathbf{t}$. They respectively unravel to

$$G^{\nu} := \mathsf{p} \xrightarrow{\nu} \mathsf{q} : k\langle U\rangle.\mathsf{r} \xrightarrow{\nu} \mathsf{s} : k'\{l_1 : G^{\nu}, \quad l_2 : G^{\nu}\}$$
$$E^{\nu} :=!^{\nu}k\langle U\rangle.\ E^{\nu}$$

We can now derive $G^\nu \downarrow^\nu_{\mathsf{p}} E^\nu$ by [M1$\downarrow^\nu$], [B$\downarrow^\nu$] followed by [M1$\downarrow^\nu$] and mark a cycle to the conclusion $G^\nu \downarrow^\nu_{\mathsf{p}} E^\nu$. This precisely justifies why we wish to project the inductive global type in (7) over role p to the local type $\mu \mathbf{t}.!^\mu k\langle U\rangle.\mathbf{t}$.

4 Projection on Inductive Types: Soundness and Completeness

The coinductive projection predicate $\downarrow^\nu$ represents the specification of an ideal projection from coinductive global types to coinductive local types. In this section, we present a projection function proj on μ-types that is sound and complete with respect to the $\downarrow^\nu$ projection predicate. We extend on previous work by Castro-Perez et al. [7] whose projection function is shown to be sound but not complete.

▶ **Definition 7** (proj). *The function* $\mathsf{proj} : \mathcal{P} \to G^\mu \rightharpoonup T^\mu$, *written* $\mathsf{proj}_{\mathsf{p}}(G^\mu)$, *is the projection of the global* μ*-type* G^μ *with respect to the role* p *and is defined as:*

$$\mathsf{proj}_{\mathsf{p}}(G^\mu) = \begin{cases} \mathsf{trans}_{\mathsf{p}}(G^\mu) & \text{if } \mathsf{projectable}_{\mathsf{p}}(G^\mu) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Our projection function features two auxiliary entities, namely the translation function trans and the predicate $\mathsf{projectable}$ which precisely separate the generation of the local type and the check for projectability respectively.

▶ **Definition 8** (trans). *The function* $\mathsf{trans} : \mathcal{P} \to G^\mu \to T^\mu$ *is identical to the function* $\lfloor^\mu$ *(see Figure 1) except for the branching case, defined as:*

$$\mathsf{trans}_{\mathsf{p}}(\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G^\mu_j\}_{j\in J}) = \begin{cases} k \oplus^\mu \{l_j : \mathsf{trans}_{\mathsf{p}}(G^\mu_j)\}_{j\in J} & \text{if } \mathsf{p} = \mathsf{p}_1 \text{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ k \,\&^\mu\, \{l_j : \mathsf{trans}_{\mathsf{p}}(G^\mu_j)\}_{j\in J} & \text{if } \mathsf{p} = \mathsf{p}_2 \text{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ \mathsf{trans}_{\mathsf{p}}(G^\mu_1) & \text{if } \mathsf{p} \notin \{\mathsf{p}_1, \mathsf{p}_2\} \end{cases}$$

The only difference from Definition 2 is that the removal of the branching condition has made trans total. These conditions are checked by the $\mathsf{projectable}$ predicate and the challenging part of implementing proj is proving decidability of this predicate.

▶ **Definition 9** ($\mathsf{projectable}$). *The predicate* $\mathsf{projectable}_{\mathsf{p}}(G^\mu)$ *states that the global* μ*-type* G^μ *is projectable with respect to the role* p *and is defined as:*

$$\mathsf{projectable}_{\mathsf{p}}(G^\mu) = \exists G^\nu\, T^\nu.\ G^\mu \mathrel{\mathcal{R}} G^\nu \ \wedge\ \mathsf{trans}_{\mathsf{p}}(G^\mu) \mathrel{\mathcal{R}} T^\nu \ \wedge\ G^\nu \downarrow^\nu_{\mathsf{p}} T^\nu$$

The predicate states that the μ-types G^μ and $\mathsf{trans}_{\mathsf{p}}(G^\mu)$ are related by unravelling to some coinductive types G^ν and T^ν respectively, and that T^ν is the coinductive projection of G^ν with respect to p. This predicate is decidable and we detail why in Section 5.

Soundness. Proving that proj is sound with respect to $\downarrow^\nu$ is relatively straightforward.

▶ **Theorem 10.** *If* $\mathsf{proj}_{\mathsf{p}}(G^\mu)$ *is defined then there exist coinductive types* G^ν *and* T^ν *such that* $G^\mu \mathrel{\mathcal{R}} G^\nu$, $\mathsf{proj}_{\mathsf{p}}(G^\mu) \mathrel{\mathcal{R}} T^\nu$ *and* $G^\nu \downarrow_p T^\nu$.

Proof. Follows directly from the definition of proj and $\mathsf{projectable}$ by setting G^ν and T^ν to their corresponding types obtained from $\mathsf{projectable}$. ◀

$$\begin{array}{ccc} G^\nu & \xrightarrow{\downarrow^\nu} & T^\nu \\ \mathcal{R}\uparrow & & \mathcal{R}\uparrow \\ G^\mu & \xrightarrow{\mathsf{proj}} & T^\mu \end{array}$$

Completeness. For completeness, we require an auxiliary operation $\mathsf{unfold}(\cdot)$ on global and local μ-types that unfolds all binders until an interaction prefix or end are exposed.

$$\mathsf{unfold}(G^\mu) = \mathsf{unfold\_once}^{|G^\mu|}(G^\mu) \qquad \mathsf{unfold\_once}(G^\mu) = \begin{cases} G_1^\mu[\mu\mathbf{t}.G_1^\mu/t], & \text{if } G^\mu = \mu\mathbf{t}.G_1^\mu \\ G^\mu & \text{otherwise} \end{cases}$$

Above, $|G^\mu|$ is the μ-height of G^μ, i.e., the number of initial consecutive binders found in G^μ. Here, f^n denotes repeated function composition. For example, $|\mu\mathbf{t}.\mathsf{end}| = 1$ and $|\mathsf{p} \xrightarrow{\mu} \mathsf{p}' : k\langle U\rangle.\mu\mathbf{t}.\ \mathsf{end}| = 0$. We overload unfolding with $\mathsf{unfold}(T^\mu)$ and $|T^\mu|$, for having the corresponding meaning on local types.

$$\begin{array}{ccc} G^\nu & \xrightarrow{\downarrow^\nu} & T^\nu \\ {\scriptstyle\mathcal{R}}\downarrow & & {\scriptstyle\mathcal{R}}\downarrow \\ G^\mu & \xrightarrow{\mathsf{proj}} & T^\mu \end{array}$$

In order to show completeness of proj with respect to $\downarrow^\nu$, we need to show that if $G^\nu \downarrow^\nu_\mathsf{p} T^\nu$ and $G^\mu \ \mathcal{R}\ G^\nu$ then $\mathsf{proj}_\mathsf{p}(G^\mu)$ is defined and $\mathsf{proj}_\mathsf{p}(G^\mu)\ \mathcal{R}\ T^\nu$. We prove this by showing that $\mathsf{trans}_\mathsf{p}(G^\mu)$ unravels to $\mathsf{tocoind}(\mathsf{trans}_\mathsf{p}(G^\mu))$; then, we show $\mathsf{trans}_\mathsf{p}(G^\mu) = \mathsf{proj}_\mathsf{p}(G^\mu)$ and $\mathsf{tocoind}(\mathsf{trans}_\mathsf{p}(G^\mu)) = T^\nu$. The function $\mathsf{tocoind}$ is defined as

▶ **Definition 11** ($\mathsf{tocoind}$). *The corecursive function* $\mathsf{tocoind} : T^\mu \to T^\nu$ *is defined as*

$$\mathsf{tocoind}(T^\mu) = \begin{cases} !^\nu k\langle U\rangle.\mathsf{tocoind}(T^\mu) & \textit{if } \mathsf{unfold}(T^\mu) = !^\mu k\langle U\rangle.T^\mu \\ ?^\nu k\langle U\rangle.\mathsf{tocoind}(T^\mu) & \textit{if } \mathsf{unfold}(T^\mu) = ?^\mu k\langle U\rangle.T^\mu \\ k \oplus^\nu \{l_j : \mathsf{tocoind}(T_j^\mu)\}_{j\in J} & \textit{if } \mathsf{unfold}(T^\mu) = k \oplus^\mu \{l_j : T_j^\mu\}_{j\in J} \\ k \ \&^\nu \{l_j : \mathsf{tocoind}(T_j^\mu)\}_{j\in J} & \textit{if } \mathsf{unfold}(T^\mu) = k \ \&^\mu \{l_j : T_j^\mu\}_{j\in J} \\ \mathsf{end}^\nu & \textit{otherwise} \end{cases}$$

Note that, $T^\mu\ \mathcal{R}\ \mathsf{tocoind}(T^\mu)$ does not always hold, as $\mathcal{R}$ is only defined for closed and contractive T^μ. However, for closed global types, trans does unravel to a coinductive type.

▶ **Lemma 12** (trans Unraveling). *If* G^μ *is closed then* $\mathsf{trans}_\mathsf{p}(G^\mu)\ \mathcal{R}\ \mathsf{tocoind}(\mathsf{trans}_\mathsf{p}(G^\mu))$.

Proof. Since G^μ is closed, we know that $\mathsf{trans}_\mathsf{p}(G^\mu)$ is closed. Moreover, the image of $\mathsf{trans}_\mathsf{p}$ is always contractive. For any closed and contractive local type T^μ, we know that $T^\mu\ \mathcal{R}\ \mathsf{tocoind}(T^\mu)$, by coinduction on $\mathcal{R}$. In particular this holds for $\mathsf{trans}_\mathsf{p}(G^\mu)$. ◀

▶ **Lemma 13** (trans as projection). *If* $G^\mu\ \mathcal{R}\ G^\nu$ *and* $G^\nu \downarrow^\nu_\mathsf{p} T^\nu$ *then* $\mathsf{tocoind}(\mathsf{trans}_\mathsf{p}(G^\mu)) = T^\nu$.

Proof. By coinduction using the candidate relation $\{(\mathsf{tocoind}(\mathsf{trans}_\mathsf{p}(G^\mu)), T^\nu) \mid G^\nu \downarrow^\nu_\mathsf{p} T^\nu \wedge G^\mu\ \mathcal{R}\ G^\nu\}$. From $G^\nu \downarrow^\nu_\mathsf{p} T^\nu$, derive $\mathsf{guarded}^\nu_\mathsf{p}(G^\nu) \vee T^\nu = \mathsf{end}^\nu$. The first case is proven by induction on $\mathsf{guarded}^\nu_\mathsf{p}(G^\nu)$; for the second we know from $G^\nu \downarrow^\nu_\mathsf{p} \mathsf{end}^\nu$ and $G^\mu\ \mathcal{R}\ G^\nu$ that $\neg\mathsf{partOf}^\nu_\mathsf{p}(G^\mu)$ and hence $\mathsf{tocoind}(\mathsf{trans}_\mathsf{p}(G^\mu)) = \mathsf{end}^\nu$. ◀

From these Lemmas, completeness follows immediately.

▶ **Theorem 14.** *If* $G^\nu \downarrow^{co}_p T^\nu$ *and* $G^\mu\ \mathcal{R}\ G^\nu$ *then* $\mathsf{proj}_\mathsf{p}(G^\mu)$ *is defined and* $\mathsf{proj}_\mathsf{p}(G^\mu)\ \mathcal{R}\ T^\nu$.

Proof. From $G^\mu\ \mathcal{R}\ G^\nu$, we know using Proposition 4 that G^μ is closed. Applying Lemma 12, we have that $\mathsf{trans}_\mathsf{p}(G^\mu)\ \mathcal{R}\ \mathsf{tocoind}(\mathsf{trans}_\mathsf{p}(G^\mu))$. Finally, from Lemma 13, we have that $\mathsf{trans}_\mathsf{p}(G^\mu)\ \mathcal{R}\ T^\nu$. It thus holds that $\mathsf{projectable}_\mathsf{p}(G^\mu)$, so $\mathsf{proj}_\mathsf{p}(G^\mu)$ is defined and $\mathsf{trans}_\mathsf{p}(G^\mu) = \mathsf{proj}_\mathsf{p}(G^\mu)$ letting us conclude $\mathsf{proj}_\mathsf{p}(G^\mu)\ \mathcal{R}\ T^\nu$. ◀

$$\frac{\mathsf{unfold}(G^\mu) \Downarrow^\mu_\mathsf{p} \ \mathsf{unfold}(T^\mu)}{G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu}\ [\mathsf{Unf}\downarrow^\mu] \qquad \frac{\mathsf{p} \notin \{\mathsf{p}_1, \mathsf{p}_2\} \quad \mathsf{guarded}^\mu_\mathsf{p}(G^\mu) \quad G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu}{\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu \Downarrow^\mu_\mathsf{p} \ T^\mu}\ [\mathsf{M}\downarrow^\mu]$$

$$\frac{G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu}{\mathsf{p} \xrightarrow{\mu} \mathsf{p}_1 : k\langle U\rangle.G^\mu \Downarrow^\mu_\mathsf{p} \ !^\mu k\langle U\rangle.T^\mu}\ [\mathsf{M1}\downarrow^\mu] \qquad \frac{\forall j.\ G^\mu_j \downarrow^\mu_\mathsf{p} \ T^\mu_j}{\mathsf{p} \xrightarrow{\mu} \mathsf{p}_1 : k\{l_j : G^\mu_j\}_{j\in J} \Downarrow^\mu_\mathsf{p} \ k \oplus^\mu \{l_j : T^\mu_j\}_{j\in J}}\ [\mathsf{B1}\downarrow^\mu]$$

$$\frac{\mathsf{p} \neq \mathsf{p}_1 \quad G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu}{\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p} : k\langle U\rangle.G^\mu \Downarrow^\mu_\mathsf{p} \ ?^\mu k\langle U\rangle.T^\mu}\ [\mathsf{M2}\downarrow^\mu] \qquad \frac{\mathsf{p} \neq \mathsf{p}_1 \quad \forall j.\ G^\mu_j \downarrow^\mu_\mathsf{p} \ T^\mu_j}{\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p} : k\{l_j : G^\mu_j\}_{j\in J} \Downarrow^\mu_\mathsf{p} \ k \ \&^\mu \{l_j : T^\mu_j\}_{j\in J}}\ [\mathsf{B2}\downarrow^\mu]$$

$$\frac{\neg\mathsf{partOf}^\mu_\mathsf{p}(G^\mu) \qquad \mathsf{Unravels}(G^\mu)}{G^\mu \Downarrow^\mu_\mathsf{p} \ \mathsf{end}_c}\ [\mathsf{End}\downarrow^\mu] \qquad \frac{J \neq \{\} \quad \mathsf{p} \notin \{\mathsf{p}_1, \mathsf{p}_2\} \quad \forall j.\ G^\mu_j \downarrow^\mu_\mathsf{p} \ T^\mu \wedge \mathsf{guarded}^\mu_\mathsf{p}(G^\mu_j)}{\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G^\mu_j\}_{j\in J} \Downarrow^\mu_\mathsf{p} \ T^\mu}\ [\mathsf{B}\downarrow^\mu]$$

Figure 4 *Intermediate projection* on inductive types, written as $G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu$.

5 Deciding Projectability

In this section, we show that projectable is decidable. We do this in two steps: first, we define the *intermediate projection* $G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu$ and show that it is sound and complete with respect to our coinductive projection; second, given a pair (G^μ, T^μ), we construct a graph and show that deciding $G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu$ can be reduced to checking properties of that graph.

An Intermediate Projection. The rules defining $G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu$, presented in Figure 4, are similar to those for coinductive projection, but also enforce the unfolding operation unfold on both μ-types. Initially, the only applicable rule is $[\mathsf{Unf}\downarrow^\mu]$, which unfolds μ-types. Then, the rules inspired by coinductive projection are used. In order to enforce unfolding every time we apply any other rule, we use the auxiliary relation $\Downarrow^\mu_\mathsf{p}$.

We now show that there is a correspondence between intermediate projection $\downarrow^\mu_\mathsf{p}$ and coinductive projection $\downarrow^\nu_\mathsf{p}$. In order to do so, we need to define how to construct a coinductive type from an inductive one. We have shown how to do this for inductive local types with $\mathsf{tocoind}(T^\mu)$ and we overload this tocoind function to similarly work with inductive global types G^μ. We use the abbreviations $\mathsf{Unravels}(G^\mu)$ and $\mathsf{Unravels}(T^\mu)$ for $G^\mu \ \mathcal{R} \ \mathsf{tocoind}(G^\mu)$ and $T^\mu \ \mathcal{R} \ \mathsf{tocoind}(T^\mu)$ respectively.

▶ **Lemma 15** (Unraveling of Projection)**.**
$G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu$ *iff* $\mathsf{Unravels}(G^\mu)$ *and* $\mathsf{Unravels}(T^\mu)$ *and* $\mathsf{tocoind}(G^\mu) \downarrow^\nu_\mathsf{p} \ \mathsf{tocoind}(T^\mu)$.

Proof. ($\Longrightarrow$) Derive both $\mathsf{Unravels}(G^\mu)$ and $\mathsf{Unravels}(T^\mu)$ by coinduction on $\mathcal{R}$ and inversion on $G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu$. Prove $\mathsf{tocoind}(G^\mu) \downarrow^\nu_\mathsf{p} \ \mathsf{tocoind}(T^\mu)$ by coinduction on $\downarrow^\nu_\mathsf{p}$ and derive from $G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu$ that $\mathsf{guarded}^\mu_\mathsf{p}(G^\mu) \vee \mathsf{unfold}(T^\mu) = \mathsf{end}^\mu$ and proceed as in Lemma 13.
($\Longleftarrow$) Proof by coinduction on $\downarrow^\mu_\mathsf{p}$ and derive from $\mathsf{tocoind}(G^\mu) \downarrow^\nu_\mathsf{p} \ \mathsf{tocoind}(T^\mu)$ that $\mathsf{guarded}^\nu_\mathsf{p}(\mathsf{tocoind}(G^\mu)) \vee \mathsf{tocoind}(T^\mu) = \mathsf{end}^\nu$, case analysis on the disjunction as in Lemma 13, inverting $\mathsf{Unravels}(G^\mu)$ and $\mathsf{Unravels}(T^\mu)$ to derive the shape of $G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu$. ◀

▶ **Corollary 16.** $\mathsf{projectable}_\mathsf{p}(G^\mu)$ *iff* $G^\mu \downarrow^\mu_\mathsf{p} \ \mathsf{trans}_\mathsf{p}(G^\mu)$.

Proof. For ($\Longrightarrow$) we first show for any G^μ and G^ν, if $G^\mu \ \mathcal{R} \ G^\nu$ then $G^\nu = \mathsf{tocoind}(G^\mu)$ (and similarly for local types). Then both directions follow from Lemma 15. ◀

Deciding $G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu$ is similar to deciding recursive type equivalence. Treatment of recursive types as graphs for equivalence testing is a well known approach [26] and solves the

problem by testing properties of reachable nodes in a directed graph. In this section, we do the same for deciding $G^\mu \downarrow^\mu_{\mathsf{p}} T^\mu$. First, we show how to transform global and local types into graphs. Then, we obtain a graph of the pair (G^μ, T^μ) by joining the graphs of G^μ and T^μ. Deciding $G^\mu \downarrow^\mu_{\mathsf{p}} T^\mu$ corresponds to testing a property on all reachable nodes of that graph.

Graphs. We first give the formal definition of graph, following that of Eikelder [26].

▶ **Definition 17** (Graph). *A directed graph is a triple (Q, d, δ) where:*

- *Q is a finite set of nodes*
- *d: $Q \to \mathbb{N}$ is a function returning the number of outgoing edges from a node*
- *$\delta : (Q \times \mathbb{N}) \rightharpoonup Q$ is the partial successor function such that $\delta(q, i)$ is the i^{th} successor of q, for $0 < i \leq d(q)$ nodes, and is undefined for all other i.*

Given a graph (Q, d, δ), we define the procedure sat_P which computes if all reachable nodes from an initial node q satisfy a given property P.

▶ **Definition 18** (sat_P). *The function $\mathit{sat}_P : 2^Q \to Q \to \{0, 1\}$, parameterised by a boolean predicate $P : Q \to \{0, 1\}$, is defined as:*

$$\mathit{sat}_P(V, q) = \begin{cases} 1 & \text{if } q \in V \\ P(q) \ \wedge \ \bigwedge_{i<d(q)} \mathit{sat}_P\big(\{q\} \cup V,\ \delta(q, i)\big) & \text{otherwise} \end{cases}$$

Given a set of visited nodes V, a current state q, and the predicate P, the function returns 1 if the node has already been visited; otherwise, it will recursively check the successors.

Global and Local Types as Graphs. We now give a procedure for constructing a graph from a global type. The graph construction for local types is similar and therefore omitted.

▶ **Definition 19** (Global type graph). *The graph of a global type G^μ is $(\mathit{enum}_g(G^\mu), d_g, \delta_g)$ where enum_g, d_g and δ_g are defined as:*

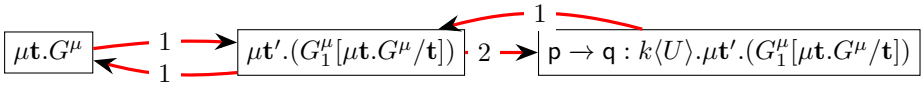
$$\mathit{enum}_g(\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu) = \{\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu\} \cup \mathit{enum}_g(G^\mu) \qquad \mathit{enum}_g(\mathsf{end}) = \{\mathsf{end}\}$$
$$\mathit{enum}_g(\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G^\mu_j\}_{j\in J}) = \{\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G^\mu_j\}_{j\in J}\} \ \cup \ \bigcup_{j\in J} \mathit{enum}_g(G^\mu_j)$$
$$\mathit{enum}_g(\mathbf{t}) = \{\mathbf{t}\} \qquad \mathit{enum}_g(\mu\mathbf{t}.G^\mu) = \{\mu\mathbf{t}.G^\mu\} \cup \{G^\mu_1[\mu\mathbf{t}.G^\mu/\ \mathbf{t}] \mid G^\mu_1 \in \mathit{enum}_g(G^\mu)\}$$

$$d_g(G^\mu) = \begin{cases} 1 & \text{if } \mathit{unfold}(G^\mu) = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle u\rangle.G^\mu \\ |J| & \text{if } \mathit{unfold}(G^\mu) = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G^\mu_j\}_{j\in J} \\ 0 & \text{otherwise} \end{cases}$$

$$\delta_g(G^\mu, i) = \begin{cases} G^\mu_1 & \text{if } \mathit{unfold}(G^\mu) = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle u\rangle.G^\mu_1 \ \wedge \ i = 1 \\ G^\mu_i & \text{if } \mathit{unfold}(G^\mu) = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G^\mu_j\}_{j\in J} \ \wedge \ 0 < i \leq |J| \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

The enumeration function enum_g collects all subterms of a global type. In the case of $\mu\mathbf{t}.G^\mu$, it enumerates all subterms of the body G^μ that can contain free occurrences of $\mathbf{t}$, and substitute them all for $\mu\mathbf{t}.G^\mu$. These subterms are all nodes of the global type graph G^μ.

▶ **Example 20.** We show the global type graph of our main example from (7).



where $G^\mu := \mathsf{p} \to \mathsf{q} : k\langle U\rangle.\ \mu\mathbf{t}'.G_1^\mu$ and $G_1^\mu := \mathsf{r} \to \mathsf{s} : k'\{l_1 : \mathbf{t},\ l_2 : \mathsf{p} \to \mathsf{q} : k\langle U\rangle.\ \mathbf{t}'\}$.

Given a global type G^μ, we wish to use $\mathsf{sat}_P(\{\}, G^\mu)$ to assert whether P holds for all nodes reachable by δ_g. To ensure termination of this procedure, we show that the set of reachable nodes is finite, a consequence of Q being closed under δ_g.

▶ **Lemma 21.** *If $G_1^\mu \in \mathit{enum}_g(G^\mu)$ and $0 \leq i < d_g(G_1^\mu)$, then $\delta_g(G_1^\mu, i) \in \mathit{enum}_g(G^\mu)$.*

Proof. Let δ_{aux} be δ_g without the use of unfold in the first two case distinctions, i.e., $\delta_g = \delta_{aux} \circ \mathsf{unfold}$. Showing $\mathsf{enum}_g(G^\mu)$ is closed under δ reduces to showing $\mathsf{enum}_g(G^\mu)$ is closed under δ_{aux} and unfold. By definition, $\mathsf{enum}_g(G^\mu)$ is closed under δ_{aux}. For $\mathsf{enum}_g(G^\mu)$ to be closed under unfold, it suffices to show that it is closed under unfold_once, which follows by induction on the μ-height. ◀

We are now ready to show how to use our graph construction for proving a property of a global type using sat_P. We do that by proving that $\mathsf{Unravels}(G^\mu)$ is decidable. In this case, we instantiate P in sat_P with a predicate that disallows global types to unfold to a top level μ-operator or a recursion variable.

▶ **Definition 22** (UnravelPred). *The predicate $\mathit{UnravelPred} : G^\mu \to \{0, 1\}$ is defined as:*

$$\mathit{UnravelPred}(G^\mu) = \begin{cases} 0 & \text{if } \mathit{unfold}(G^\mu) = \mu\mathbf{t}.G_1^\mu \vee \mathit{unfold}(G^\mu) = \mathbf{t} \\ 1 & \text{otherwise} \end{cases}$$

▶ **Lemma 23.** *$\mathit{Unravels}(G^\mu)$ iff $\mathit{sat}_{\mathit{UnravelPred}}(\{\}, G^\mu) = 1$*

The instantiation $\mathsf{sat}_{\mathsf{UnravelPred}}$ tests that G^μ and all successors of G^μ unfold to a message communication, a branching or end^μ. The procedure will for example fail for $\mu\mathbf{t}.\mathbf{t}$. More details on this procedure are given in Section 6.

We conclude this part by defining the partial functions LG, LT and PL_p. Given an inductive global type, function LG returns its *unfolded prefix*, i.e., information about its first occurring interaction.

▶ **Definition 24** (LG). *The function $LG \in G \rightharpoonup (\mathcal{P} \times \mathcal{P} \times \mathcal{C} \times (\{\bot\} \cup U))$ is defined as:*

$$LG(G^\mu) = \begin{cases} (\mathsf{p}_1, \mathsf{p}_2, k, U) & \text{if } \mathit{unfold}(G^\mu) = \mathsf{p}_1 \to \mathsf{p}_2 : k\langle U\rangle.G_1^\mu \\ (\mathsf{p}_1, \mathsf{p}_2, k, \bot) & \text{if } \mathit{unfold}(G^\mu) = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G_j^\mu\}_{j\in J} \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

Similar to how LG returns the unfolded prefix in a global type, we define the corresponding operation on local types as LT. We use the set $\{!, ?\}$ to indicate whether the communication is a send (!) or a receive (?).

▶ **Definition 25** (LT). *The function $LT : T \rightharpoonup (\{!, ?\} \times \mathcal{C} \times (\{\bot\} \cup U))$ is defined as:*

$$LT(T^\mu) = \begin{cases} (!, k, U) & \text{if } \mathit{unfold}(T^\mu) = !^\mu k\langle U\rangle.T_1^\mu \\ (?, k, U) & \text{if } \mathit{unfold}(T^\mu) = ?^\mu k\langle U\rangle.T_1^\mu \\ (!, k, \bot) & \text{if } \mathit{unfold}(T^\mu) = k \oplus^\mu \{l_j : T_j^\mu\}_{j\in J} \\ (?, k, \bot) & \text{if } \mathit{unfold}(T^\mu) = k \,\&^\mu\, \{l_j : T_j^\mu\}_{j\in J} \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

Finally, we can define a projection function on prefixes, i.e., a function that given a role and an unfolded prefix of a global type, returns an unfolded prefix of a local type.

▶ **Definition 26.** *The function $PL_\mathsf{p} \in (\mathcal{P} \times \mathcal{P} \times \mathcal{C} \times (\{\bot\} \cup U)) \rightharpoonup (\{!, ?\} \times \mathcal{C} \times (\{\bot\} \cup U))$ is defined as:*

$$PL_\mathsf{p}(\mathsf{p}_1, \mathsf{p}_2, k, U) = \begin{cases} (!, k, U), & \text{if } \mathsf{p}_1 = \mathsf{p} \\ (?, k, U), & \text{if } \mathsf{p}_2 = \mathsf{p} \text{ and } \mathsf{p} \neq \mathsf{p}_1 \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

Combining global and local type graphs. The next step towards deciding membership in $\lfloor\!\lfloor^{\mu}_{\mathsf{p}}$ is to combine the graphs of G^μ and T^μ into a single graph with respect to a role p.

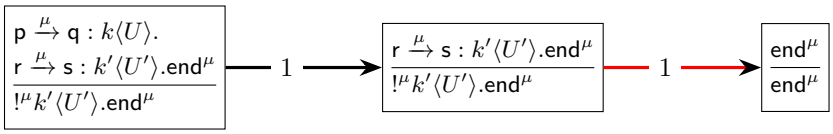
▶ **Definition 27** (Joint Global and Local Type Graph). *The graph of* (G^μ, T^μ) *with respect to* p *is the graph* $(\mathit{enum}(G^\mu, T^\mu), d, \delta_{\mathsf{p}})$*, where* enum, d *and* δ_{p} *defined as:*

$$\mathit{enum}(G^\mu, T^\mu) = \mathit{enum}(G^\mu) \times \mathit{enum}(T^\mu) \qquad d(G^\mu, T^\mu) = min(d_g(G^\mu), d_l(T^\mu))$$

$$\delta_{\mathsf{p}}((G^\mu, T^\mu), i) = \begin{cases} (\delta_g(G^\mu, i), \delta_l(T^\mu, i)) & \text{if } \mathsf{p} \in LG(G^\mu) \wedge 0 < i \leq d(G^\mu, T^\mu) \\ (\delta_g(G^\mu, i), T^\mu) & \text{if } \mathsf{p} \notin LG(G^\mu) \wedge 0 < i \leq d_g(G^\mu) \\ \mathit{undefined} & \mathit{otherwise} \end{cases}$$

The set of nodes is the Cartesian product of the nodes in the global type graph and the local type graph. The successor function δ_{p} takes a step with respect to a role p and the case distinction depends on this role. If p is in $LG(G^\mu)$, the i^{th} successor of the graph is the i^{th} successor of the global and local type graph respectively. If p is not in the unfolded prefix, the global type moves to its successor while the local type stays fixed.

▶ **Example 28.** We show the joint graph of $\mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\langle U'\rangle.\mathsf{end}^\mu$ and $!^\mu k\langle U'\rangle.\mathsf{end}^\mu$ with respect to role c marking the edges black when the local type stays fixed.



Deciding membership in $\lfloor\!\lfloor^{\mu}_{\mathsf{p}}$. We define the predicate $\mathsf{ProjPred}_{\mathsf{p}}$ to decide membership in $\lfloor\!\lfloor^{\mu}_{\mathsf{p}}$. Intuitively, this predicate partitions the rules of $\lfloor\!\lfloor^{\mu}_{\mathsf{p}}$ into three sets such that the projected role p

1. is in the unfolded prefix ($[\mathsf{M1}\lfloor^\mu], [\mathsf{M2}\lfloor^\mu], [\mathsf{B1}\lfloor^\mu], [\mathsf{B2}\lfloor^\mu]$)
2. is not in the unfolded prefix, but the role is guarded in the global type ($[\mathsf{B}\lfloor^\mu]$,$[\mathsf{M}\lfloor^\mu]$)
3. is not part of the global type ($[\mathsf{End}\lfloor^\mu]$).

We call rules in the first set *prefix rules* and rules in the second set *guarded rules*. The only rule that is not yet mentioned is unfolding, $[\mathsf{Unf}\lfloor^\mu]$, which is implicitly applied by the definition of δ.

▶ **Definition 29** ($\mathsf{ProjPred}_{\mathsf{p}}$). *The boolean predicate* $\mathit{ProjPred}_{\mathsf{p}} \in G^\mu \times T^\mu \to \{0, 1\}$ *is defined as:*

$$\mathit{ProjPred}_{\mathsf{p}}(G^\mu, T^\mu) = \begin{cases} (\mathbf{1})\ PL_{\mathsf{p}}(LG(G^\mu)) = LT(T^\mu) \wedge & \\ \quad d_g(G^\mu) = d_l(T^\mu) & \text{if } PL_{\mathsf{p}}(LG(G^\mu)) \text{ is defined} \\ (\mathbf{2})\ 0 < d_g(G^\mu) & \text{if } \mathsf{partOf}^\mu_{\mathsf{p}}(G^\mu) \text{ and } \mathsf{guarded}^\mu_{\mathsf{p}}(G^\mu) \\ (\mathbf{3})\ \mathit{sat}_{\mathit{UnravelPred}}(\{\}, G^\mu) \wedge & \\ \quad \neg\mathsf{partOf}^\mu_{\mathsf{p}}(G^\mu) \wedge & \\ \quad \mathit{unfold}(T^\mu) = \mathit{end} & \mathit{otherwise} \end{cases}$$

We explain the three cases of the predicate.

1. Attempt to apply a prefix rule: This requires p to be in the unfolded prefix of the global type. This is checked by requiring that PL_{p} is defined. We then apply PL_{p} to the unfolded prefix, and assert it equal to the unfolded prefix of the local type. All prefix rules require the global and local type to have equally many outgoing edges, which we check by $d_g(G^\mu) = d_l(T^\mu)$.

2. Attempt to apply a guarded rule: We rely on decidability of partOf^{μ} and $\mathsf{guarded}^{\mu}$ which is straightforward so we do not detail how[2]. All guarded-rules require the set of outgoing edges of the global type to be greater than zero, which we assert. Concretely this test corresponds to the first premise of rule $[\mathsf{B}\downarrow^{\mu}]$, asserting its label set is non-empty.
3. Attempt to apply $[\mathsf{End}\downarrow^{\mu}]$.

▶ **Theorem 30.** $G^{\mu} \downarrow^{\mu}_{\mathsf{p}} T^{\mu}$ *iff* $\mathit{sat}_{\mathit{ProjPred}_{\mathsf{p}}}(\{\}, (G^{\mu}, T^{\mu})) = 1$

Proof. For $(\Longrightarrow)$, we show the property for any visited list v, that is, $G^{\mu} \downarrow^{\mu}_{\mathsf{p}} T^{\mu}$ implies $\mathsf{sat}_{\mathsf{ProjPred}_{\mathsf{p}}}(v, (G^{\mu}, T^{\mu})) = 1$. Proceed by functional induction on $\mathsf{sat}_{\mathsf{ProjPred}_{\mathsf{p}}}(v, (G^{\mu}, T^{\mu}))$. For $(\Longleftarrow)$, for any v, it suffices to show $\mathsf{sat}_{\mathsf{ProjPred}_{\mathsf{p}}}(v, (G^{\mu}, T^{\mu})) = 1$ implies $(G^{\mu}, T^{\mu}) \in v \vee G^{\mu} \downarrow^{\mu}_{\mathsf{p}} T^{\mu}$. Proceed by functional induction on $\mathsf{sat}_{\mathsf{ProjPred}_{\mathsf{p}}}(v, (G^{\mu}, T^{\mu}))$. In the second case where v is non-empty, pick the right disjunct $G^{\mu} \downarrow^{\mu}_{\mathsf{p}} T^{\mu}$ and proceed by coinduction. ◀

▶ **Corollary 31.** $\mathsf{projectable}_{\mathsf{p}}(G^{\mu})$ *is decidable.*

Proof. Follows from Theorem 30 and Corollary 16. ◀

6 Mechanisation

All of our results are mechanised in Coq [6] using SSReflect [14] for writing proofs, the Paco library [18] for defining coinductive predicates, the Equations package [24] for defining functions by well-founded recursion (such as sat_P), and Autosubst2 [25] to generate syntax of inductive global and local types with binders represented by De Bruijn indices [10].

The mechanisation uses coinductive extensional equivalence relations to equate coinductive terms. For presentation purposes, e.g. in the conclusion of Lemma 12, we use propositional equality to equate coinductive types. These two types of equality are consistent [1].

In this section, we cover how to create predicates and relations that are defined using both inductive and coinductive inference rules, like our unravelling relation from Definition 3. We discuss how to create an inversion principle that allows us to do case analysis on predicates of the form $\mathsf{Unravels}(G^{\mu})$ which, as discussed in Section 5, is defined as $G^{\mu}\ \mathcal{R}\ \mathsf{tocoind}(G^{\mu})$. Finally, we show how we prove decidability of $\mathsf{Unravels}$ using sat_P.

Mixed inductive and coinductive definitions. The unravelling relation presented in Definition 3 uses a combination of inductive and coinductive rules, which is non-standard. We do this because it greatly simplifies our proofs and disallows unwanted derivations like the one presented in Section 3 (8) by construction. We mix inductive and coinductive rules by taking the greatest fixed point of a generating function defined as a least fixed point, a technique that Zakowski et al. [29] also have used to define weak bisimilarity of streams.

```
Definition grel := gType -> gcType -> Prop
Inductive UnravelF (R : grel) : grel := (* Generating function UnravelF *)
 | UnrF1 g gc a u : R g gc -> UnravelF R (GMsg a u g) (GCMsg a u gc)
 (* The branching rule is elided *)
 | UnrF_unf1 g gc : UnravelF R (unf1 (GRec g)) gc -> UnravelF R (GRec g) gc
 | UnrF_end       : UnravelF R GEnd GCEnd.
Definition Unravelling : grel := paco2 UnravelF bot2 (* gfp UnravelF *)
```

[2] We need to assert both $\mathsf{partOf}^{\mu}_{\mathsf{p}}(G^{\mu})$ and $\mathsf{guarded}^{\mu}_{\mathsf{p}}(G^{\mu})$ for completeness as we from $G^{\nu} \downarrow^{\nu}_{\mathsf{p}} \mathsf{end}^{\nu}$ and $G^{\mu}\ \mathcal{R}\ G^{\nu}$ then can conclude the third case of $\mathsf{ProjPred}_{\mathsf{p}}$.

We represent $\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu$ and $\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\langle U\rangle.G^\nu$ as GMsg a u g of type gType and GCMsg a u gc of type gcType respectively, where a contains roles p_1, p_2 and channel k, u is U, g is G^μ, and gc is G^ν. The terms GEnd and GCEnd represent end^μ and end^ν respectively and the function unf1 is the unfold_once function from Section 4.

UnravelF is an inductively defined relation relating global inductive types to global coinductive types. It is parameterised by a relation R of the same type where (g, gc) $\in$ UnravelF R if, after unfolding a finite number of binders from g resulting in type g', either g' = GEnd and gc = GCEnd, or g' = GMsg a u g'', gc = GCMsg a u gc'', and (g'', gc'') $\in$ R, or similarly for the elided branch case.

Intuitively, UnravelF is a generating function defined as a least fixed point and by taking the greatest fixed point of this function we obtain a hybrid inductive/coinductive relation where any occurrence of R in a premise of UnfoldF require us to take coinductive steps in our proofs and any recursive occurrence of UnfoldF requires us to take inductive steps. This allows us to do proofs like (9) where proofs are finished by circling back to previous equivalent nodes in the tree in the coinductive cases or by reaching a base case in the inductive cases. Moreover this approach forbids us from unfolding binders indefinitely since UnrF_unf1 is inductive and not coinductive.

We use paco2 from the Paco library to define Unravelling as the greatest fixed point of UnravelF. Paco stands for parameterised coinduction and paco2 F R defines the greatest fixed point of F parameterised by a binary relation R, which is equivalent to gfp(λX. F($X \cup$ R)). When R is the empty set this coincides with the standard greatest fixed point.

Custom inversion principles. Many proofs on inductive global types work up to unfolding. Unravelling, for instance, unravels a finite number of μ-binders at every step and our intermediate projection function $\lfloor \cdot \rfloor^\mu_{\mathsf{p}}$ and sat procedure both work in a similar way. To abstract away from finite unfoldings we use the following InvPred predicate.

```
Variant InvPredF (P : gType -> Prop) : gType -> Prop :=
| HTM g a u : P g          -> InvPredF P (GMsg a u g)
| HTB gs d  : Forall P es -> InvPredF P (GBranch d gs)
| HTE       :                 InvPredF P GEnd
Definition unf g := (iter (mu_height g) unf1 g).
Variant UnfoldF (P : gType -> Prop) : gType -> Prop :=
 | UnfF1 g :  P (unf g) -> UnfoldF g.
Definition InvPred : (gType -> Prop) := paco1 (UnfoldF \o InvPredF) bot.
                                              (*function composition*)
```

We define two generating functions InvPredF and UnfoldF and generate InvPred as the greatest fixed point of their composition. The function unf corresponds to unfold from Section 4. InvPredF contains cases for all constructors of inductive global types except for $\mu\mathbf{t}$ and $\mathbf{t}$. UnfoldF unfolds the top-level μ-binders from a global type. The key insight is that InvPred(G^μ) is equivalent to asserting closedness and contractiveness of G^μ.

The inversion principle of InvPred is convenient for proving predicates P that are closed under unfolding of inductive global types, i.e. $\forall G.\ P\ \mu\mathbf{t}.G \iff P\ G[\mu\mathbf{t}.G]$, as any unfolding applied by inverting UnfoldF can similarly be applied in the goal. In particular the predicate Unravels(G^μ) is closed under unfolding and provable by inversion of UnfoldF.

Well-foundedness of sat$_P$. Lemma 23 proves decidability of Unravels. This proof is mechanised by proving decidability of InvPred(G^μ), which as we show above implies Unravels(G^μ). The invP predicate corresponds to Definition 22.

```
Definition invP g :=
match unf g with | GRec _ | GVar _ => false | _ => true end.

Definition invpred g := sat nil invP g.
Theorem InvPred_dec : forall g, InvPred g <-> invpred g = true
```

We use the Equations package to define sat_P by well-founded recursion on the decreasing measure gmeasure g V which is defined as the number of unique nodes in the graph created from g minus the cardinality of the visited set V. The successor function δ_g is implemented by nextg : gType -> seq gType.

```
1  Definition gmeasure (g : gType) (V : seq gType) :=
2    size (rep_rem V (undup (enumg g))).
3  Lemma closed_enum  : forall g0 g1 g2, g1 \in nextg (unf g) ->
4                                        g2 \in enumg g1  -> g2 \in enumg g.
5  Equations sat (V : seq  gType)  (P : gType -> bool)
6                     (g : gType) : bool by wf (gmeasure g V) :=
7   sat V P g with (dec (g \in V)) => {
8    sat  _ _ _ in_left  := true;
9    sat V P g  in_right :=  (P g) &&
10                           (foldInMap (nextg (unf g))
11                                      (fun g' _ => sat (g::V) P g')) }.
```

Defining sat generates one obligation that must be proved to show termination. If we write gmeasure g V as $M(g, V)$, then we must show it is decreasing for arguments to the recursive call, i.e. that $M(g', \{g\} \cup V) < M(g, V)$

Using a variant of the familiar map on inductive lists called foldInMap our obligation is enriched with the assumption that $g' = \delta(g, i)$ for some $0 < i \leq d_g(g)$. The boolean wrapper dec further enriches the obligation with the case of the if-statement, $g \notin V$.

What must be proven in this obligation is slightly different from the termination argument in Section 5 which relied on the finiteness of a graph's nodes. The obligation instead relies on a lemma closed_enum (l. 3). The lemma states that the enumerations of a global types continuations, will all be part of the initial global types enumeration. The proof of this lemma is short, less than 100 lines.

The full termination proof for sat is short (about 250 lines) and the approach is general. The mechanisation also proves termination of the decision procedure for membership in $\downarrow^{ind}$. This task only requires adapting the algorithm to pairs of terms. This termination proof is also short. The conciseness is due to the space of continuations being computed by structural recursion by enum. This makes it straightforward to prove substitution properties about it by induction on syntax.

7 Related Work and Discussion

Related Work. Ghilezan et al. [13] are the first to introduce coinductive projection on coinductive global and local types. They use it to show soundness and completeness of synchronous multiparty session subtyping. A key difference is that whereas we represent the infinite unfolding of a μ-type as a coinductive type, they represent it as a partial function. Projection on μ-types is then defined indirectly in terms of the coinductive projection of their corresponding partial functions. Because of this indirect definition, their projection is not computable. Our intermediate projection $\downarrow^{\mu}$ is similar to their projection on μ-types.

However, ours is defined with inference rules stated directly on the μ-types which is why we can decide membership and thus compute projection. Castro-Perez et al. [7] use coinductive projection to express their meta theory about multiparty session types. Their main result is trace equivalence between processes, coinductive local types and coinductive global types, which they mechanise in Coq. Like us, they show soundness of their projection on μ-types. Their projection is however not complete, which is what inspired us to investigate approaches to sound and complete projection. A consequence of their projection on μ-types not being complete, is that there are many inductive global types that have the trace equivalence property, but must be excluded since their projection is undefined. Jacob et al. [19] show deadlock and leak freedom of multiparty GV, an extension of the functional language GV [12, 28]. They use coinductive projection to define when local types are compatible and do not define a projection on μ-types. Other work has formalised the notion of projection in Coq. Cruz-Filipe et al. [9, 8] formalise syntax and semantics of tail-recursive choreographies and a projection that includes full merge. However, this work does not approach coinductive syntax and therefore does not show any soundness and completeness results.

Our graph algorithm from Section 5 implements a procedure proposed by Eikelder [26]. This work provides several algorithms for deciding recursive type equivalence that, like ours, use predicates on reachable nodes of a graph. Also, our proof of termination is quite similar to theirs. However, they define the set of reachable states as set comprehension, whereas we constructively produce a list of nodes. Similarly, showing their set comprehension is finite, boils down to substitution lemmas. Unlike ours, their work has not been mechanised in a proof assistant/theorem prover. The idea of defining the space of continuations for global and local type as an explicit enumeration is inspired by Asperti [3] who mechanise a concise proof of regular expression equivalence in the Matita theorem prover [4]. They do this by a new construction called pointed regular expressions. Essentially, this adds marks to a regular expression, such that one can encode state transitions by moving marks. This makes computing reachable configurations as trivial as computing all markings.

We define unravelling using a mix of inductive and coinductive rules. In Section 6, we make this precise by defining unravelling as the greatest fixed point of a generating function itself defined as a least fixed point. Zakowski et al. [29] use the same technique to define a weak bisimilarity on streams.

The primary focus of this work is on global types. Scalas and Yoshida [23] propose a more general approach that shows that properties such as deadlock freedom can be derived directly on local types without the need for global types and the corresponding projection. However, their approach misses the main advantage provided by global types which is providing a specification (blueprint) of the used protocols.

Discussion and Future work. This work is part of the MECHANIST project that aims at mechanising the full theory of multiparty asynchronous session types [17]. Our next step is to mechanise a proof of semantic equivalence between global types and their projections to local types through $\mathsf{proj_p}$. Semantic equivalence is a property similar to trace equivalence which Castro-Perez et al. [7] mechanised. However, there are some key differences in our objectives. Their main result is Zooid, a tool that extracts certified message-passing programs, which is why their process syntax differs significantly from the original syntax by Honda et al (e.g., no parallel composition). Instead, we aim at mechanising the exact process calculus presented by Honda et al.. As the meta theory in Castro-Perez et al. [7] is independent of their projection function, it would also be interesting future work to adapt $\mathsf{proj_p}$ to their setting. Finally, $\mathsf{proj_p}$ implements the restrictive plain merge but related work also uses full merge [13, 8]. It would be interesting to define a binder-agnostic projection using full merge.

8 Conclusions

Projection is a function that maps global types to local types. The projections found in the literature impose syntactic restrictions that make them incomplete with respect to coinductive projection. This work shows the existence of a decidable projection that is sound and complete. Our procedure works in two phases: first a decision procedure tests a soundness property and, if successful, a second procedure translates the global type to a local type. The latter is very similar to the existing projections in the literature. The novelty of our work is in the decision procedure. All results have been mechanised in Coq.

References

1 Coinductive types and corecursive functions. https://coq.inria.fr/refman/language/core/coinductive.html. Accessed: May 2023.

2 Session types in programming languages: A collection of implementations. http://www.simonjf.com/2016/05/28/session-type-implementations.html. Accessed: May 2023.

3 Andrea Asperti. A compact proof of decidability for regular expression equivalence. In *proceedings of ITP*, volume 7406 of *LNCS*, pages 283–298. Springer, 2012. doi:10.1007/978-3-642-32347-8_19.

4 Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The Matita interactive theorem prover. In *Proceedings of CADE*, volume 6803 of *LNCS*, pages 64–69. Springer, 2011. doi:10.1007/978-3-642-22438-6_7.

5 Andi Bejleri and Nobuko Yoshida. Synchronous multiparty session types. In *Proceedings of PLACES*, volume 241 of *ENTCS*, pages 3–33. Elsevier, 2008. doi:10.1016/j.entcs.2009.06.002.

6 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.

7 David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. Zooid: a DSL for certified multiparty computation: from mechanised metatheory to certified multiparty processes. In *Proceedings of PLDI*, pages 237–251. ACM, 2021. doi:10.1145/3453483.3454041.

8 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Certifying choreography compilation. In Antonio Cerone and Peter Csaba Ölveczky, editors, *Proceedings of ICTAC 2021*, volume 12819 of *Lecture Notes in Computer Science*, pages 115–133. Springer, 2021. doi:10.1007/978-3-030-85315-0_8.

9 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Formalising a turing-complete choreographic language in coq. In Liron Cohen and Cezary Kaliszyk, editors, *Proceedings of ITP 2021*, volume 193 of *LIPIcs*, pages 15:1–15:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ITP.2021.15.

10 Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. doi:10.1016/1385-7258(72)90034-0.

11 Romain Demangeon and Nobuko Yoshida. On the expressiveness of multiparty sessions. In *Proceedings of FSTTCS*, volume 45 of *LIPIcs*, pages 560–574, 2015. doi:10.4230/LIPIcs.FSTTCS.2015.560.

12 Simon Gay and Vasco Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, 2010. doi:10.1017/S0956796809990268.

13 Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for synchronous multiparty sessions. *Journal of Logical and Albegraic Methods in Programming*, 104:127–173, 2019. doi:10.1016/j.jlamp.2018.12.002.

14 Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A small scale reflection extension for the coq system, 2016. URL: https://inria.hal.science/inria-00258384.

15 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Proceedings of ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.

16 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of POPL*, pages 273–284. ACM, 2008. doi:10.1145/1328438.1328472.

17 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *Journal of the ACM*, 63(1):9:1–9:67, 2016. doi:10.1145/2827695.

18 Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The power of parameterization in coinductive proof. In *Proceedings of POPL*, pages 193–206. ACM, 2013. doi:10.1145/2429069.2429093.

19 Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Multiparty GV: functional multiparty session types with certified deadlock freedom. *Proceedings of the ACM on Programming Languages*, 6(ICFP):466–495, 2022. doi:10.1145/3547638.

20 Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. Cocaml: Functional programming with regular coinductive types. *Fundamenta Informaticae*, 150:347–377, 2017. doi:10.3233/FI-2017-1473.

21 The Coq development team. The Coq Proof Assistant. https://coq.inria.fr. Accessed: May 2023.

22 Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

23 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, 2019. doi:10.1145/3290343.

24 Matthieu Sozeau and Cyprien Mangin. Equations reloaded: High-level dependently-typed functional programming and proving in coq. *Proceedings of the ACM on Programming Languages*, 3(ICFP):86:1–86:29, 2019. doi:10.1145/3341690.

25 Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: reasoning with multi-sorted de bruijn terms and vector substitutions. In *Proceedings of CPP*, pages 166–180. ACM, 2019. doi:10.1145/3293880.3294101.

26 Huub ten Eikelder. Some algorithms to decide the equivalence of recursive types. https://pure.tue.nl/ws/files/2150345/9211264.pdf, 1991. Accessed: May 2023.

27 Rob van Glabbeek, Peter Höfner, and Ross Horne. Assuming just enough fairness to make session types complete for lock-freedom. In *Proceedings of LICS*, pages 1–13. IEEE, 2021. doi:10.1109/LICS52264.2021.9470531.

28 Philip Wadler. Propositions as sessions. In *Proceedings of ICFP*, pages 273–286. ACM, 2012. doi:10.1145/2364527.2364568.

29 Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. An equational theory for weak bisimulation via generalized parameterized coinduction. In *Proceedings of CPP*, pages 71–84. ACM, 2020. doi:10.1145/3372885.3373813.

Real-Time Double-Ended Queue Verified (Proof Pearl)

Balazs Toth
Department of Computer Science, Technische Univerität München, Germany

Tobias Nipkow
Department of Computer Science, Technische Univerität München, Germany

Abstract

We present the first verification of the real-time doubled-ended queue by Chuang and Goldberg where all operations take constant time. The main contributions are the full system invariant, the precise definition of all abstraction functions, the structure of the proof and the main lemmas.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Theory of computation → Data structures design and analysis; Software and its engineering → Functional languages

Keywords and phrases Double-ended queue, data structures, verification, Isabelle

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.29

Funding *Tobias Nipkow*: Research partially supported by DFG Koselleck grant NI 491/16-1.

1 Introduction

Based on the work of Chuang and Goldberg [2] we implement and formally verify a double-ended queue (*deque*) in a purely functional language such that each *enqueue* and *dequeue* operation on either end takes $\mathcal{O}(1)$ time in the worst case. This is what *real-time* means. Operations on previous versions of a deque are in constant time since purely functional data structures are persistent by default.

The deque implementation by Chuang and Goldberg consists of two stacks, with each stack corresponding to one of the two ends of the deque. These two stacks are balanced at all time, meaning that the bigger stack is never more than three times bigger than the smaller stack. The *enqueue* and *dequeue* operations use the respective stacks (by pushing and popping). The deque maintains its size invariant by rebalancing the two ends. Since such a rebalancing takes time $\mathcal{O}(n)$, it distributes a constant fraction of the rebalancing steps on the *enqueue* and *dequeue* operations before the invariant can be violated again. This achieves worst-case and not just amortized constant time for each operation. We show the detailed implementation in Section 5.

Chuang and Goldberg [2, p.292] describe the main size invariant of a real-time deque and explain how this invariant is re-established via rebalancing of the two ends. But to formally verify the implementation, we need much more detailed invariants, which also capture the state during rebalancing. For example, an explicit measure of the remaining rebalancing steps is needed. We verify the implementation w.r.t. a formal specification of deques. The verification uses the interactive theorem prover Isabelle/HOL [12, 11]. The Isabelle theories are available online [14] and comprise 4400 lines of definitions and proofs. Some of the names in this paper have been modified (mostly shortened) for presentation reasons.


© Balazs Toth and Tobias Nipkow;
licensed under Creative Commons License CC-BY 4.0

1.1 Related Work

The quest for efficient functional queues started with the two-stack implementation by Burton [1] where all operations take amortized constant time. Hood and Melville [7] show how to obtain a real-time implementation by distributing the work of moving elements from one of the two stacks to the other one over a whole sequence of *enqueue* and *dequeue* operations. A verification of Hood and Melville's queue can be found elsewhere [4, 10]. The principles of the real-time deque were already described by Hood [6], apparently unbeknownst to Chuang and Goldberg. Madhavan and Kuncak [9] automatically verified the amortized constant-time complexity of a simpler deque (see start of Section 4).

Okasaki [13] shows how to obtain simpler implementations of real-time queues and deques by relying on lazy evaluation. The resource requirements of his code were analyzed semi-automatically by Madhavan *et al.* [8].

2 Preliminaries

Isabelle types are built from type variables, e.g. $'a$, and (postfix) type constructors, e.g. $'a\ list$; the function type arrow is $\Rightarrow$. The notation $t :: \tau$ means that term t has type τ. The notation f^n, where $f :: \tau \Rightarrow \tau$, is the n-fold composition of f with itself.

Type $'a\ list$ are lists of elements of type $'a$. They come with the following vocabulary: (#) (list constructor), (@) (append), $|xs|$ (length of list xs), $rev\ xs$ (reverse of xs), hd (head), tl (tail, where $tl\ [] = []$), $take\ n\ xs$ (take the first n elements of list xs), $drop\ n\ xs$ (drop the first n elements of list xs), $take\_last\ n\ xs$ (take the last n elements of list xs), and other self-explanatory notation. Type nat are the natural numbers. Pairs come with the projection functions fst and snd. Logical equivalence is written $=$ instead of $\longleftrightarrow$. A yellow background marks the code of the actual implementation of the data structure, while code without a background is just used for the verification.

3 Specification

The interface is comprised of the functions

$empty :: 'q$	$is\_empty :: 'q \Rightarrow bool$
$enqL :: 'a \Rightarrow 'q \Rightarrow 'q$	$enqR :: 'a \Rightarrow 'q \Rightarrow 'q$
$deqL :: 'q \Rightarrow 'q$	$deqR :: 'q \Rightarrow 'q$
$firstL :: 'q \Rightarrow 'a$	$firstR :: 'q \Rightarrow 'a$

where $'q$ is the type of deques and $'a$ the type of elements. They allow enqueuing and dequeuing elements on both ends (as indicated by the L/R suffixes). We express the specification using an abstraction function $listL :: 'q \Rightarrow 'a\ list$

$listL\ empty = []$	$is\_empty\ q = (listL\ q = [])$
$listL\ (enqL\ x\ q) = x \# listL\ q$	$listR\ (enqR\ x\ q) = x \# listR\ q$
$listL\ q \neq [] \longrightarrow listL\ (deqL\ q) = tl\ (listL\ q)$	$listR\ q \neq [] \longrightarrow listR\ (deqR\ q) = tl\ (listR\ q)$
$listL\ q \neq [] \longrightarrow firstL\ q = hd\ (listL\ q)$	$listR\ q \neq [] \longrightarrow firstR\ q = hd\ (listR\ q)$

where $listR\ q = rev\ (listL\ q)$. The above properties express that $listL$ and $listR$ are homomorphisms from deques to lists. There is also an invariant $invar :: 'q \Rightarrow bool$ and $invar\ q$ is an additional precondition of the above equations, except for $listL\ empty = []$. All operations are required to preserve $invar$ – we do not show the corresponding propositions.

4 Abstract Description of Implementation

A deque is represented by two stacks, one for each end of the deque. Things work well as long as both stacks remain non-empty. As soon as one becomes empty and a *deq* (= *pop*) operation is to be performed, we need to move part of the other stack over to the empty side first. It can be shown that if the (bottom) half of the non-empty stack is moved (and reversed), this leads to an implementation with amortized constant-time operations.

To achieve worst-case constant-time complexity the invariant $n \geq m \geq 1 \wedge 3 * m \geq n$ is maintained where m and n are the sizes of the smaller and the bigger stacks S and B. If the length of the deque is ≤ 3, it is represented by a single list, all operations are trivially constant-time and the above invariant does not apply. We focus on the two-stack situation. The invariant can be violated by dequeuing on the smaller stack or enqueuing on the larger stack. Let S and B be the stacks after the violating operation and let $m = |S|$ and $n = |B|$. Then $3 * m < n$ and either $3 * (m + 1) \geq n$ (pop) or $3 * m \geq n - 1$ (push), i.e. $n = 3 * m + k$ where $1 \leq k \leq 3$. Thus there are P and Q such that $B = P$ @ Q and $|Q| = m + 1$. Now we transform S into S @ *rev* Q and B into P in 5 phases. In each phase, we pop the elements off one stack and push them onto another stack, thus reversing the order.

*Big*1	Pop the top $2 * m + k - 1$ elements off B onto a new stack rP: $B = Q$ and $rP =$ *rev* P
*Small*1	Reverse S onto a new stack rS: $rS =$ *rev* S
*Big*2	Reverse rP onto a new stack B': $B' = P$
*Small*2	Reverse B onto a new stack S': $S' =$ *rev* Q
*Small*3	Reverse rS onto S': $S' = S$ @ *rev* Q

Now S' and B' are the new stacks. Phases *Big*1 and *Small*1 can be performed in parallel thus taking at most $2 * m + 2 + 1$ steps – the 1 is an administrative step between phases. Similarly, phase *Big*2 can be performed in parallel with phases *Small*2 followed by *Small*3, thus taking at most $2 * m + 2 + 1$ steps again. The $4 * m + 6$ steps are spread out as follows: 6 steps are performed in the violating operation and 4 steps in each subsequent *enq* and *deq*. The invariant cannot be violated again during those m operations: we start with stacks S' and B' of size $2 * m + 1$ and $2 * m + k - 1$; in the worst case $k = 1$ and all m operations are *deq*s on B'; in the end we still have $3 * (2 * m + k - 1 - m) \geq 2 * m + 1$. In fact, it takes about $4/3 * m$ *deq*s or $4 * m$ pops before the invariant can be violated again. Because rebalancing happens in parallel with enqueuing and dequeuing, the stacks are augmented with further data structures. A counter keeps track of how many elements of the original stacks are still valid – every *deq* decrements the counter. An additional list *ext* is maintained that *enq*s push to. At the end of the 5 phases, we cannot just append S' to *ext* – this would not be constant-time. Thus stacks are actually implemented as pairs of lists (which complicates push and pop a little) and phase *Small*3 returns (*ext*, T) where T is S' or B' above, which are real lists, not stacks.

5 Verified Implementation

A deque can be in one of the following states:

```
datatype 'a deque = Empty | One 'a | Two 'a 'a | Three 'a 'a 'a
                  | Idles ('a idle) ('a idle) | Rebal ('a states)
```

- A deque contains less than four elements (first four constructors), or
- it consists of two stacks representing the ends of the deque (*Idles* constructor), or
- it is in the middle of rebalancing (*Rebal* constructor).

The emptyness check is trivial:

```
is_empty Empty = True
is_empty _ = False
```

Note that all code is shown on coloured background to distinguish it easily from all verification-related material.

In the following, we will show the implementation bottom-up, except for the rebalancing process, where we follow the order of the phases. There are a number of overloaded functions that are defined on multiple types:

- Functions *push* and *pop* that implement *enq* and *deq*.
- Function *step* implements the rebalancing steps.
- An invariant *invar*.
- Two abstraction functions to lists: *list* returns the list abstraction after rebalancing, *list_current* returns the list in the current, non-rebalanced state.
- Function *remaining_steps* calculates the remaining steps of a rebalancing process.

The invariant, list abstractions and *remaining_steps* are not code but key components of the verification and important contributions of our paper. Some other functions are also overloaded. For types that only have a function *list* its *size* is defined as:

$$size\ d = |list\ d|$$

If it has *list* and *list_current* then there are *size* and *size_new*:

$$size\ d = min\ |list\_current\ d|\ |list\ d|$$
$$size\_new\ d = |list\ d|$$

We verified the following properties for every type that have the respective functions:

$$list\ (push\ x\ d) = x\ \#\ list\ d$$
$$invar\ d \longrightarrow size\ (push\ x\ d) = size\ d + 1$$
$$invar\ d \longrightarrow invar\ (push\ x\ d)$$
$$invar\ d \longrightarrow remaining\_steps\ (push\ x\ d) = remaining\_steps\ d$$

$$invar\ d \wedge 0 < size\ d \wedge pop\ d = (x,\ d') \longrightarrow x\ \#\ list\ d' = list\ d$$
$$invar\ d \wedge 0 < size\ d \wedge pop\ d = (x,\ d') \longrightarrow size\ d' = size\ d - 1$$
$$invar\ d \wedge pop\ d = (x,\ d') \longrightarrow invar\ d'$$
$$invar\ d \wedge pop\ d = (x,\ d') \longrightarrow remaining\_steps\ d' \leq remaining\_steps\ d$$

$$invar\ d \longrightarrow list\ (step\ d) = list\ d$$
$$invar\ d \longrightarrow size\ d = size\ (step\ d)$$
$$invar\ d \longrightarrow invar\ (step\ d)$$
$$invar\ d \longrightarrow remaining\_steps\ (step\ d) = remaining\_steps\ d - 1$$

For *list_current* and *size_new* the same properties hold as for *list* and *size*.

Our collection of datatypes is considerably more refined than those by Chuang and Goldberg because we express a number of the implicit invariants in their code explicitly on the level of types. For example, Chuang and Goldberg's type *Deque* has a constructor *LIST* :: *'a list* ⇒ *Deque* that is applied only to lists of size ≤ 4. The latter is an important implicit invariant that guarantees that operations *rev* and (@), which are applied to arguments of *LIST*, execute in constant time. Our type *deque* expresses the invariant clearly via the first four constructors. As a result, our implementation is more explicit but requires more small building blocks.

5.1 Stack

The basic building block for our implementation is the type *'a stack* that serves as the ends of the deque. It actually consists of two stacks represented by lists:

```
datatype 'a stack = Stack ('a list) ('a list)
```

The stack operations below use the left of the two stacks first, and resort to the right list if the left one is empty. As explained towards the end of Section 4 the right list contains elements resulting from a rebalancing, and the left list holds elements that were newly enqueued during rebalancing.

```
push x (Stack left right) = Stack (x # left) right
```

```
pop (Stack [] []) = Stack [] []
pop (Stack (x # left) right) = Stack left right
pop (Stack [] (x # right)) = Stack [] right
```

```
first (Stack (x # left) right) = x
first (Stack [] (x # right)) = x
```

```
is_empty (Stack [] []) = True
is_empty (Stack _ _) = False
```

There is no invariant but a list abstraction function:

```
list (Stack left right) = left @ right
```

5.2 Idle

Datatype *idle* represents an end of the deque that is not in a rebalancing process.

```
datatype 'a idle = Idle ('a stack) nat
```

It contains a *stack* to which it delegates its *push* and *pop* operations. Furthermore, we will need to check the size of the end frequently, to know whether rebalancing is required. To achieve this in constant time, we keep track of the size of the *stack* and update it with every operation accordingly.

```
push x (Idle stk n) = Idle (push x stk) (n + 1)
```

```
pop (Idle stk n) = (first stk, Idle (pop stk) (n − 1))
```

The invariant *invar* (*Idle stk n*) = (*size stk* = *n*) is obvious. The *list* function delegates to the corresponding *list* function on the stack; we omit showing such trivial definitions.

5.3 Current

Now we start to look into the rebalancing procedure. Type *'a current* stores information about operations that happen during rebalancing but which have not become part of the old state that is being rebalanced.

```
datatype 'a current = Current ('a list) nat ('a stack) nat
```

Both ends of the deque contain a *current* state which contains a list of newly enqueued elements and their number. The *push* operation on a *current* state adds to the list and increases its size counter:

```
push x (Current ext extn old tar) = Current (x # ext) (extn + 1) old tar
```

Additionally, *current* has a *stack* keeping track of the end's state before rebalancing. The natural number after it is the **target size** (usually denoted by *tar*) of the end after rebalancing, but without taking the *ext* component into account. The *pop* operation on *current* enables dequeuing of elements during the rebalancing: If there are newly enqueued elements, *pop* dequeues an element from the corresponding list and adjusts its size counter. Otherwise, it dequeues an element from the old state of the end and reduces the target size by one.

```
pop (Current (x # ext) extn old tar) = (x, Current ext (extn − 1) old tar)
pop (Current [] extn old tar) = (first old, Current [] extn (pop old) (tar − 1))
```

The operations preserve the obvious invariant for the counter of newly enqueued elements:

```
invar (Current ext extn _ _) = (|ext| = extn)
```

The abstraction *list* yields the list of the state before rebalancing, but modified by the intervening *push*'s and *pop*'s. *current* has next to its *size* function based on *list*, an additional function *size_new* calculating the target size at the end of rebalancing.

```
list (Current ext _ old _) = ext @ list old
size_new (Current _ extn _ tar) = extn + tar
```

5.4 Rebalancing

Rebalancing transfers elements from the bigger end to the smaller one. Datatype *states* stores both ends (types *big_state* and *small_state* are explained below) together with a *direction* indicating if the transfer happens from left to right or right to left. Therefore it also indicates which end is on which side.

```
datatype 'a states = States direction ('a big_state) ('a small_state)
datatype direction = L | R
```

Table 1 Rebalancing phases.

Big	Small
*Big*1 _ (*P* @ *Q*) [] |*P*|	*Small*1 _ *S* []
↓	↓
*Big*1 _ *Q* (*rev P*) 0	*Small*1 _ [] (*rev S*)
Copy _ (*rev P*) [] 0	*Small*2 _ (*rev S*) *Q* [] 0
	↓
	*Small*2 _ (*rev S*) [] (*rev Q*) |*Q*|
↓(*Big*2)	*Copy* _ (*rev S*) (*rev Q*) |*Q*|
	↓ (*Small*3)
Copy _ [] *P* |*P*|	*Copy* _ [] (*S* @ *rev Q*) (|*S*| + |*Q*|)

The phases described in Section 4 are represented by the following constructors for the big and small end of the deque, with their corresponding behaviour w.r.t. rebalancing steps. *Big*2 and *Small*3 perform the same work and are both represented by the constructor *Copy*.

- *Big*1 :: *'a current* ⇒ *'a stack* ⇒ *'a list* ⇒ *nat* ⇒ *'a big_state*
  *Big*1 _ *S xs n* pops the top *n* elements off *S* and puts them on *xs*.
- *Small*1 :: *'a current* ⇒ *'a stack* ⇒ *'a list* ⇒ *'a small_state*
  *Small*1 _ *S xs* pops all elements off *S* and puts them on *xs*.
- *Small*2 :: *'a current* ⇒ *'a list* ⇒ *'a stack* ⇒ *'a list* ⇒ *nat* ⇒ *'a small_state*
  *Small*2 _ *xs S ys n* pops all elements off *S*, puts them on *ys*, counts them in *n* and leaves *xs* unchanged.
- *Copy* :: *'a current* ⇒ *'a list* ⇒ *'a list* ⇒ *nat* ⇒ *'a common_state*
  Copy cur xs ys n pops elements off *xs*, puts them on *ys*, and counts them, until *n* reaches the *tar* component of *cur*. Stopping before all of *xs* has been moved has the effect of performing the *deq* operations that have accumulated in *cur* during rebalancing.

Every phase contains a *current* state that deals with *enq* and *deq* operations (see Section 5.3).

Table 1 shows how each phase leads to the next at both ends of the deque. The variables are named as in Section 4. For readability we have equated stacks with lists. For simplicity the *Copy* phases assume that the copying is not cut short by a reduced *tar*. We will implement overloaded *step* functions that advance and transition the phases step-by-step.

5.4.1 Big

The bigger end of the deque goes through two phases during rebalancing, modeled with datatype *big_state* with two constructors:

```
datatype 'a big_state = Big1 ('a current) ('a stack) ('a list) nat
                      | Big2 ('a common_state)
```

Both constructors were explained at the beginning of Section 5.4. At that point we pretended that both ends of the deque have a common constructor *Copy*. Instead, constructor *Big*2 is a wrapper around a common state *common_state* (see Section 5.4.3) that both ends delegate their *push*/*pop*/*step* operations to in phases *Big*2 and *Small*3. Operations *push* and *pop* on *Big*1 are delegated to *current*. Function *step* uses *norm* for the transition from phase *Big*1 to *Big*2 which is defined in Section 5.4.3.

```
push x (Big1 cur big aux n) = Big1 (push x cur) big aux n
push x (Big2 state) = Big2 (push x state)
```

```
pop (Big1 cur big aux n) = (let (x, cur) = pop cur in (x, Big1 cur big aux n))
pop (Big2 state) = (let (x, state) = pop state in (x, Big2 state))
```

```
step (Big1 cur big aux 0) = Big2 (norm (Copy cur aux [] 0))
step (Big1 cur big aux n) = Big1 cur (pop big) (first big # aux) (n − 1)
step (Big2 state) = Big2 (step state)
```

The remaining functions on *big_state* again delegate to *common_state* in phase *Big*2. We do not show those equations.

The following invariant is preserved by *push*, *pop* and *step*:

```
invar (Big1 cur big aux n) = let Current _ _ old tar = cur in
      invar cur
∧     tar ≤ |aux| + n                                          (1)
∧     n ≤ size big                                             (2)
∧     take_last (size old) (rev aux @ list big) = list old     (3)
∧     take tar (rev (take n (list big)) @ aux) = rev (take tar (list old))   (4)
```

(1) The target size of the end after the rebalancing (*tar*) is ≤ to the total number of elements that the phase reverses (|*aux*| + *n*). This needs to hold because phase *Big*1 moves to *aux* the elements that remain on this end. Only ≤ but not = holds because of potentially dequeued elements that reduce the target size (see Section 5.3: *pop*).

(2) There must be at least as many elements as the phase wants to reverse.

(3) Undoing the progress of the phase by reversing *aux* back and appending it back to *big* reproduces the old state of the end. We account for potentially dequeued elements by dropping those from the front of the restored end.

(4) Finishing the phase by reversing and appending *n* more elements to *aux* gives us the elements that remain on this end in a reversed order.

In phase *Big*1, *list* finishes rebalancing and returns the final list of the end. In contrast, *list_current* returns the original list of the end.

```
list (Big1 (Current ext _ _ tar) big aux n) =
  let a = rev (take n (list big)) @ aux in ext @ rev (take tar a)

list_current (Big1 cur _ _ _) = list cur
```

The verification also requires the number of remaining *step*s of rebalancing:

```
remaining_steps (Big1 (Current _ _ _ tar) _ _ n) = n + tar + 1
```

In phase *Big*1, *n* more elements need to be moved before 1 additional step transitions to phase *Big*2 which requires *tar* steps.

5.4.2 Small

As depicted in Table 1, the smaller end of the deque goes through three different phases during rebalancing. They are represented by datatype *small_state*:

```
datatype 'a small_state = Small1 ('a current) ('a stack) ('a list)
                        | Small2 ('a current) ('a list) ('a stack) ('a list) nat
                        | Small3 ('a common_state)
```

Just as in *big_state*, constructor *Small*3 contains the common data structure to which the phases *Big*2 and *Small*3 delegate their operations (see Section 5.4.3). This time we do not show any of the trivial delegating equations.

Operations *push* and *pop* are defined analogously to their *big_state* relatives:

```
push x (Small1 cur small aux) = Small1 (push x cur) small aux
push x (Small2 cur aux big new n) = Small2 (push x cur) aux big new n
```

```
pop (Small1 cur small aux) = (let (x, cur) = pop cur in (x, Small1 cur small aux))
pop (Small2 cur aux big new n)
= (let (x, cur) = pop cur in (x, Small2 cur aux big new n))
```

In phase *Small*1, *step* idles once it has emptied its stack because it needs to wait for the big end to finish phase *Big*1 before both ends can transition to their next phases simultaneously (see Section 5.4.4). In phase *Small*2 the stack is popped until it is empty and phase *Small*3 starts:

```
step (Small1 cur small aux)
= (if is_empty small then Small1 cur small aux
    else Small1 cur (pop small) (first small # aux))
step (Small2 cur aux big new n)
= (if is_empty big then Small3 (norm (Copy cur aux new n))
    else Small2 cur aux (pop big) (first big # new) (n + 1))
```

The following invariant, presented phase by phase, is preserved by *push*, *pop* and *step*:

```
invar (Small1 cur small aux) = let Current _ _ old tar = cur in
      invar cur
∧     size old ≤ tar                                   (1)
∧     size old ≤ size small + |aux|                    (2)
∧     take_last (size old) (rev aux @ list small) = list old   (3)
```

(1) The target size is not smaller than the original size of the end. Otherwise, rebalancing would not be successful because the smaller end would shrink further.

(2) The stack holding the original elements of the smaller end (*old*) cannot grow but potentially shrink through *pop* operations. Moreover, since phase *Small*1 is reversing a copy of the original elements of the smaller size, the total number of elements it works on is $\geq$ to the size of the stack *old*.

(3) Undoing the progress of the phase by reversing *aux* back and appending it back to *small* reproduces the old state of the end. We account for potentially dequeued elements by dropping those from the front of the restored end.

$$
\begin{aligned}
&\mathit{invar}\ (\mathit{Small2}\ \mathit{cur}\ \mathit{aux}\ \mathit{big}\ \mathit{new}\ n) = \mathsf{let}\ \mathit{Current}\ \_\ \_\ \mathit{old}\ \mathit{tar} = \mathit{cur}\ \mathsf{in}\\
&\quad \mathit{invar}\ \mathit{cur}\\
&\wedge\quad n = |\mathit{new}| && \textbf{(1)}\\
&\wedge\quad \mathit{tar} = n + \mathit{size}\ \mathit{big} + \mathit{size}\ \mathit{old} && \textbf{(2)}\\
&\wedge\quad \mathit{size}\ \mathit{old} \le |\mathit{aux}| && \textbf{(3)}\\
&\wedge\quad \mathit{rev}\ (\mathit{take}\ (\mathit{size}\ \mathit{old})\ \mathit{aux}) = \mathit{list}\ \mathit{old} && \textbf{(4)}
\end{aligned}
$$

(1) The phase counts its reversed elements correctly.

(2) The elements transferred from the bigger end and the original elements from the smaller end will build the new smaller end. Consequently, the sum of their elements is equal to the target size. Hereby, the number of transferred elements is split into already reversed and not yet reversed ones.

(3, 4) Next to the reversal, phase *Small*2 also holds the already reversed original state of the smaller end. Accordingly, it is equal to *old* when reversed back and accounted for the potentially dequeued elements.

Of the abstraction functions *list* and *list_current* we merely show *list* because *list_current* simply delegates to its counterpart on *current*.

$$
\begin{aligned}
&\mathit{list}\ (\mathit{Small2}\ (\mathit{Current}\ \mathit{ext}\ \_\ \_\ \mathit{tar})\ \mathit{aux}\ \mathit{big}\ \mathit{new}\ n)\\
&= \mathit{ext} \mathbin{@} \mathit{rev}\ (\mathit{take}\ (\mathit{tar} - n - \mathit{size}\ \mathit{big})\ \mathit{aux}) \mathbin{@} \mathit{rev}\ (\mathit{list}\ \mathit{big}) \mathbin{@} \mathit{new}
\end{aligned}
$$

Function *list* is partial. It is lacking a case for phase *Small*1 because phase *Small*1 lacks the elements coming from the bigger end, so it is impossible to simulate all further steps of the rebalancing. The lacking case will be added one level higher for *States* where we also have the bigger end available (see Section 5.4.4).

For phase *Small*2, *list* finishes the reversal of the transferred elements, prepends the reversed result of phase *Small*1 while accounting for potentially dequeued elements, and prepends the potentially enqueued elements.

The smaller end does not have its own remaining steps measurement because they depend on the state of the bigger end.

5.4.3 Common

The datatype *'a common_state* is a joint representation of phases *Big*2 and *Small*3:

$$
\begin{aligned}
&\textbf{datatype}\ 'a\ \mathit{common\_state} = \mathit{Copy}\ ('a\ \mathit{current})\ ('a\ \mathit{list})\ ('a\ \mathit{list})\ \mathit{nat}\\
&\qquad\qquad\qquad\qquad\quad \mid \mathit{Idle}\ ('a\ \mathit{current})\ ('a\ \mathit{idle})
\end{aligned}
$$

Copy represents rebalancing; *Idle* signals termination of rebalancing and keeps the rebalanced state of an end in an *idle* state (see Section 5.2).

$$
\begin{aligned}
&\mathit{step}\ (\mathit{Copy}\ \mathit{cur}\ \mathit{aux}\ \mathit{new}\ n)\\
&= (\mathsf{let}\ \mathit{Current}\ \mathit{ext}\ \mathit{extn}\ \mathit{old}\ \mathit{tar} = \mathit{cur}\\
&\quad \mathsf{in}\ \mathit{norm}\\
&\qquad (\mathsf{if}\ n < \mathit{tar}\ \mathsf{then}\ \mathit{Copy}\ \mathit{cur}\ (\mathit{tl}\ \mathit{aux})\ (\mathit{hd}\ \mathit{aux} \mathbin{\#} \mathit{new})\ (n+1)\\
&\qquad\ \mathsf{else}\ \mathit{Copy}\ \mathit{cur}\ \mathit{aux}\ \mathit{new}\ n))\\
&\mathit{step}\ (\mathit{Idle}\ \mathit{cur}\ \mathit{idle}) = \mathit{Idle}\ \mathit{cur}\ \mathit{idle}
\end{aligned}
$$

Function *norm* performs the transition back to an idle end. If *tar* has been reached, *norm* creates a new *stack* and puts the elements that arrived during rebalancing in the front and the result of rebalancing in the back and sets the size accordingly:

```
norm (Copy cur aux new n)
= (let Current ext extn old tar = cur
    in if tar ≤ n then Idle cur (Idle (Stack ext new) (extn + n))
      else Copy cur aux new n)
```

Both constructors also contain a *current* state on which the *push* and *pop* operations work:

```
push x (Copy cur aux new n) = Copy (push x cur) aux new n
push x (Idle cur (Idle stk n)) = Idle (push x cur) (Idle (push x stk) (n + 1))
```

```
pop (Copy cur aux new n)
= (let (x, cur) = pop cur in (x, norm (Copy cur aux new n)))
pop (Idle cur idle) = (let (x, idle) = pop idle in (x, (Idle (fst (pop cur)) idle))
```

Both operations also update the *idle* component when the respective phase terminated. Additionally, the *pop* operation checks if it dequeued the last element of the reversal and transitions, using *norm*, to the idle phase if so.

For the phases *Big*2 and *Small*3 the invariant is the following:

```
invar (Copy cur aux new n) = let Current _ _ old tar = cur in
      invar cur
∧     n < tar                                                    (1)
∧     n = |new|                                                  (2)
∧     tar ≤ |aux| + n                                            (3)
∧     take tar (list old) = take (size old) (rev (take (tar − n) aux) @ new)   (4)
```

(1) The number of elements for which the rebalancing is finished did not yet reach the target number.
(2) n correctly holds the number of finished elements.
(3) There are enough elements left to reach the target number.
(4) When simulating the termination by reversing the missing elements, the front of the new and old end are the same.

The invariant for the idle state requires that the subcomponents satisfy their invariants and that the fronts of the old and the rebalanced ends are the same:

```
invar (Idle cur idle)
= invar cur ∧ invar idle ∧ take (size idle) (list cur) = take (size cur) (list idle)
```

Function *list* finishes the phases *Big*2/*Small*3 and prepends the elements that arrived during rebalancing. In the *Idle* state it delegates to *list* on *idle*.

```
list (Copy (Current ext _ _ tar) aux new n) = ext @ rev (take (tar − n) aux) @ new
list (Idle _ idle) = list idle
```

The abstraction function *list_current* simply delegates to its counterpart on *current*.

Counting of the remaining steps is similarly straightforward. In phases *Big*2/*Small*3 the difference between the processed elements and the target remains; the idle state does not need any more steps.

```
remaining_steps (Copy (Current _ _ _ tar) _ _ n) = tar − n
remaining_steps (Idle _ _) = 0
```

5.4.4 States

Putting the two ends together into *states* completes the implementation of the rebalancing procedure. Remember that in Section 5.4.2 phase *Small*1 could not transition to *Small*2 by itself because it needs to synchronize with the end of *Big*1. The *step* function on *states* covers this case by moving from *Small*1 to *Small*2 once *Big*1 has reached 0. The other cases were already covered by the *step* functions on *big_state* and *small_state*.

$$\begin{aligned}&\mathit{step}\ (\mathit{States}\ \mathit{dir}\ (\mathit{Big}1\ \mathit{currentB}\ \mathit{big}\ \mathit{auxB}\ 0)\ (\mathit{Small}1\ \mathit{currentS}\ \_\ \mathit{auxS}))\\ &= \mathit{States}\ \mathit{dir}\ (\mathit{step}\ (\mathit{Big}1\ \mathit{currentB}\ \mathit{big}\ \mathit{auxB}\ 0))\ (\mathit{Small}2\ \mathit{currentS}\ \mathit{auxS}\ \mathit{big}\ []\ 0)\\ &\mathit{step}\ (\mathit{States}\ \mathit{dir}\ \mathit{big}\ \mathit{small}) = \mathit{States}\ \mathit{dir}\ (\mathit{step}\ \mathit{big})\ (\mathit{step}\ \mathit{small})\end{aligned}$$

The joint list abstraction *lists* returns the pair containing the lists for the two ends. It also compensates for the partiality of *list* on the smaller end: *lists* simulates the remaining steps of phase *Small*1 and performs the transition to phase *Small*2, for which *list* is already defined, to create the missing list abstraction for phase *Small*1. For the other phases it calls the respective list abstractions.

$$\begin{aligned}&\mathit{lists}\ (\mathit{States}\ \_\ (\mathit{Big}1\ \mathit{curB}\ \mathit{big}\ \mathit{auxB}\ n)\ (\mathit{Small}1\ \mathit{curS}\ \mathit{small}\ \mathit{auxS}))\\ &= (\mathit{list}\ (\mathit{Big}1\ \mathit{curB}\ \mathit{big}\ \mathit{auxB}\ n),\\ &\quad \mathit{list}\ (\mathit{Small}2\ \mathit{curS}\ (\mathit{rev}\ (\mathit{take}\ n\ (\mathit{list}\ \mathit{small}))\ @\ \mathit{auxS})\ (\mathit{pop}^n\ \mathit{big})\ []\ 0))\\ &\mathit{lists}\ (\mathit{States}\ \_\ \mathit{big}\ \mathit{small}) = (\mathit{list}\ \mathit{big},\ \mathit{list}\ \mathit{small})\end{aligned}$$

Function *lists_current* simply delegates to the big and small end:

$$\mathit{lists\_current}\ (\mathit{States}\ \_\ \mathit{big}\ \mathit{small}) = (\mathit{list\_current}\ \mathit{big},\ \mathit{list\_current}\ \mathit{small})$$

For convenience, we define

$$\mathit{list\_small\_first}\ \mathit{states} = (\mathsf{let}\ (\mathit{big},\ \mathit{small}) = \mathit{lists}\ \mathit{states}\ \mathsf{in}\ \mathit{small}\ @\ \mathit{rev}\ \mathit{big})$$

$$\begin{aligned}&\mathit{list\_current\_small\_first}\ \mathit{states}\\ &= (\mathsf{let}\ (\mathit{big},\ \mathit{small}) = \mathit{lists\_current}\ \mathit{states}\ \mathsf{in}\ \mathit{small}\ @\ \mathit{rev}\ \mathit{big})\end{aligned}$$

and analogously *list_big_first* and *list_current_big_first*.

The invariant is defined as follows:

$$\begin{aligned}&\mathit{invar}\ (\mathit{States}\ \mathit{dir}\ \mathit{big}\ \mathit{small}) =\\ &\quad \mathit{invar}\ \mathit{big} \wedge \mathit{invar}\ \mathit{small}\\ &\wedge\ \ \mathit{list\_small\_first}\ (\mathit{States}\ \mathit{dir}\ \mathit{big}\ \mathit{small})\\ &\quad = \mathit{list\_current\_small\_first}\ (\mathit{States}\ \mathit{dir}\ \mathit{big}\ \mathit{small}) && \textbf{(1)}\\ &\wedge\ \ \mathsf{case}\ (\mathit{big},\ \mathit{small})\ \mathsf{of}\\ &\quad (\mathit{Big}1\ \_\ \mathit{big}\ \_\ n,\ \mathit{Small}1\ (\mathit{Current}\ \_\ \_\ \mathit{old}\ \mathit{tar})\ \mathit{small}\ \_) \Rightarrow\\ &\qquad \mathit{size}\ \mathit{big} - n = \mathit{tar} - \mathit{size}\ \mathit{old} \wedge \mathit{size}\ \mathit{small} \leq n && \textbf{(2,3)}\\ &\quad \mid (\mathit{Big}1\ \_\ \_\ \_\ \_,\ \_) \Rightarrow \mathit{False} && \textbf{(4)}\\ &\quad \mid (\mathit{Big}2\ \_\ ,\ \mathit{Small}1\ \_\ \_\ \_) \Rightarrow \mathit{False} && \textbf{(5)}\\ &\quad \mid (\_\ ,\ \_) \Rightarrow \mathit{True}\end{aligned}$$

(1) Rebalancing preserves the abstract queue (as a list): the list abstraction after the end of rebalancing must be the same as the list abstraction that uses the state before rebalancing.

(2) After phase *Big*1, the bigger end transfers exactly the number of elements missing on the smaller end to reach the target size.

(3) Phase *Big*1 does not finish before *Small*1. This needs to hold because the smaller end transitions from phase *Small*1 to *Small*2 at the end of stage *Big*1.

(4) Phase *Big*1 can only occur together with phase *Small*1.
(5) Phase *Big*2 cannot occur together with phase *Small*1.

The case analysis in the invariant ensures that phase *Big*1 runs in parallel with phase *Small*1, and phase *Big*2 with the phases *Small*2 and *Small*3.

The overall remaining steps are the maximum of the remaining steps of both ends:

```
remaining_steps (States _ big small) =
  max
    (remaining_steps big)
    (case small of
       Small1 (Current _ _ _ tar) _ _ ⇒ let Big1 _ _ _ nB = big in nB + tar + 2
     | Small2 (Current _ _ _ tar) _ _ _ nS ⇒ tar − nS + 1
     | Small3 state ⇒ remaining_steps state)
```

We focus on the smaller end because we covered the bigger end already in Section 5.4.1. The remaining steps for the small end in phase *Small*1 cannot be calculated in isolation because they depend on the big end: Phase *Small*1 needs to wait for phase *Big*1 to finish, which are nB steps. Then it moves via *Small*2 and *Small*3 to *Idle*, counting up until the target size tar is reached. Consequently, the smaller end needs $nB + tar$ steps from phase *Small*1 to finish and 2 more steps for the transitions. In phase *Small*2, the counter is at nS already and hence $tar - nS$ steps remain, plus 1 for the last transition. The remaining steps for phase *Small*3 are already covered in Section 5.4.3.

Finally, we must ensure that the deque re-establishes the size constraints after rebalancing. therefore *size_ok* calculates, relative to the remaining steps, if the size constraints can be met: it is not allowed that one end is more than 3 times larger than the other after rebalancing. Additionally, none of the ends is allowed to be empty at the end. Herefore, it is also important that both ends have enough elements to facilitate all the *dequeue* operations that can potentially happen. Therefore, *size_ok* uses the size measurements implemented for both ends.

```
size_ok states = size_ok' states (remaining_steps states)

size_ok' (States _ big small) steps =
     size_new small + steps + 2 ≤ 3 * size_new big
   ∧ size_new big + steps + 2 ≤ 3 * size_new small
   ∧ steps + 1 ≤ 4 * size small
   ∧ steps + 1 ≤ 4 * size big
```

Note that $(m \leq k * n \wedge n \leq k * m) = (max\ m\ n \leq k * min\ m\ n)$, i.e. we have merely rewritten the size invariant from Section 4.

5.5 Deque

Finally, we can put together all the parts for the overall invariant:

```
invar (Idles l r) =
     invar l ∧ invar r ∧ ¬ is_empty l ∧ ¬ is_empty r
   ∧ size l ≤ 3 * size r ∧ size r ≤ 3 * size l
invar (Rebal states) = (invar states ∧ size_ok states ∧ 0 < remaining_steps states)
invar _ = True
```

In the idle state, the deque must satisfy the invariants of both ends and the size constraints between them. During rebalancing, the deque must satisfy the invariant of the rebalancing process, must ensure that it meets the size constraints after rebalancing, and *remaining_steps*

must correctly predict that there are further steps needed. The other states of the deque fulfill the invariant trivially.

The overall list abstraction function *listL* (Section 3) is composed trivially from the separate states' list abstractions:

```
listL Empty = []
listL (One x) = [x]
listL (Two x y) = [x, y]
listL (Three x y z) = [x, y, z]
listL (Idles left right) = list left @ rev (list right)
listL (Rebal states) = listL states

listL (States L big small) = list_small_first (States L big small)
listL (States R big small) = list_big_first (States R big small)
```

5.5.1 Enqueuing

Function *enqL* enqueues one element on the left end of the deque and returns the resulting deque.

```
enqL x Empty = One x
enqL x (One y) = Two x y
enqL x (Two y z) = Three x y z
enqL x (Three a b c) = Idles (Idle (Stack [x, a] []) 2) (Idle (Stack [c, b] []) 2)
enqL x (Idles l (Idle r nR)) =
  let Idle l nL = push x l in
  if nL ≤ 3 * nR then Idles (Idle l nL) (Idle r nR)
  else let nl = nl − nR − 1;
      nR = 2 * nL + 1;
      big = Big1 (Current [] 0 l nL) l [] nL;
      small = Small1 (Current [] 0 r nR) r [];
      states = States R big small;
      states = step^6 states;
  in Rebal states
enqL x (Rebal (States L big small) =
  let small = push x small;
      states = step^4 (States L big small);
  in case states of
       States L (Big2 (Idle _ big)) (Small3 (Idle _ small)) ⇒ Idle small big
     | _ ⇒ Rebal states
```

```
enqL x (Rebal (States R big small) =
  let big = push x big;
      states = step^4 (States R big small);
  in case states of
       States R (Big2 (Idle _ big)) (Small3 (Idle _ small)) ⇒ Idle big small
     | _ ⇒ Rebal states
```

Function *enqL* advances the constructors *Empty*, *One* and *Two* to the next larger one. For *Three*, it transitions the deque into the idle state by placing two elements at each end.

In the idle state, it enqueues one element on the left end and checks if the size invariant between the two ends still holds. If so, it keeps the deque in the idle state. Otherwise, it initiates rebalancing in the same way as $deqL'$, but in the other direction.

If the deque is already in the rebalancing process, *enqL* enqueues the new element and advances rebalancing by 4 steps. If that finishes rebalancing, it moves back into the idle state.

Function *enqR*, the counterpart of *enqL*, swaps the two ends of the deque, calls *enqL* and swaps the ends back.

```
enqR x d = swap (enqL x (swap d))
```

5.5.2 Dequeuing

The function $deqL'$ dequeues one element from the left end of the deque and returns the dequeued element and the remaining deque. Accordingly, it implements *deqL* and *firstL* simultaneously.

```
deqL' (One x) = (x, Empty)
deqL' (Two x y) = (x, One y)
deqL' (Three x y z) = (x, Two y z)
deqL' (Idles l (Idle r nR)) =
  let (x, Idle l nL) = pop l in
  if nR ≤ 3 * nL then (x, Idles (Idle l nL) (Idle r nR))
  else if 1 ≤ nL then
    let nL' = 2 * nL + 1;
        nR' = nR − nL − 1;
        small = Small1 (Current [] 0 l nL') l [];
        big = Big1 (Current [] 0 r nR') r [] nR';
        states = States L big small;
        states = step^6 states;
    in (x, Rebal states)
  else case r of Stack r1 r2 ⇒ (x, small_deque r1 r2)
deqL' (Rebal (States L big small) =
  let (x, small) = pop small;
      states = step^4 (States L big small);
  in case states of
       States R (Big2 (Idle _ big)) (Small3 (Idle _ small)) ⇒ (x, Idle big small)
     | _ ⇒ (x, Rebal states)
```

```
deqL' (Rebal (States R big small) =
  let (x, big) = pop big;
      states = step^4 (States R big small);
  in case states of
       States R (Big2 (Idle _ big)) (Small3 (Idle _ small)) ⇒ (x, Idle big small)
     | _ ⇒ (x, Rebal states)
```

If the deque has less than four elements, $deqL'$ dequeues the leftmost element and transitions to the next smaller constructor (not shown).

In the idle state, $deqL'$ dequeues an element from the left end and checks if the size invariant between the two ends still holds. If so, the deque stays in the idle states. Otherwise, it checks if the left end became empty and transitions to one of the small states using *small_deque* (below) in that case. In the last case, when the left end is not empty and the size constraints are violated, it starts rebalancing. Therefore, it divides the total number of elements into two almost equal halves – the right is one larger because the total number is odd. Then, the phases $Big1$ (for the bigger, right side) and $Small1$ (for the smaller, left side) are initialized with these numbers as target sizes and the state of the respective end. Finally, $deqL'$ starts rebalancing by executing 6 steps.

When the deque is already in the rebalancing state, $deqL'$ dequeues one element from the respective end and advances the rebalancing with 4 more steps. If that finishes rebalancing, it transitions the deque back into the idle state.

$$\begin{array}{ll}
small_deque\ []\ []\ =\ Empty & small_deque\ []\ [x,\ y]\ =\ Two\ y\ x \\
small_deque\ [x]\ []\ =\ One\ x & small_deque\ []\ [x,\ y,\ z]\ =\ Three\ z\ y\ x \\
small_deque\ []\ [x]\ =\ One\ x & small_deque\ [x,\ y,\ z]\ []\ =\ Three\ z\ y\ x \\
small_deque\ [x]\ [y]\ =\ Two\ y\ x & small_deque\ [x,\ y]\ [z]\ =\ Three\ z\ y\ x \\
small_deque\ [x,\ y]\ []\ =\ Two\ y\ x & small_deque\ [x]\ [y,\ z]\ =\ Three\ z\ y\ x
\end{array}$$

Function $deqR'$, analogously to $enqR$, is reduced to $deqL'$ by swapping the ends twice. $deqR$ and $firstR$ are specializations of $deqR'$.

$$deqR'\ deque\ =\ (\mathsf{let}\ (x,\ deque)\ =\ deqL'\ (swap\ deque)\ \mathsf{in}\ (x,\ swap\ deque))$$

5.6 Proof

In this section we explain how the top-level properties of the specification in Section 3 are proved. This is what we proved for $enqL$ and $deqL'$:

$$\begin{array}{l}
invar\ d\ \longrightarrow\ listL\ (enqL\ x\ d)\ =\ x\ \#\ listL\ d \\
invar\ d\ \longrightarrow\ invar\ (enqL\ x\ d)
\end{array} \qquad (*)$$

$$\begin{array}{l}
invar\ d\ \wedge\ listL\ d\ \neq\ []\ \wedge\ deqL'\ d\ =\ (x,\ d')\ \longrightarrow\ x\ \#\ listL\ d'\ =\ listL\ d \\
invar\ d\ \wedge\ \neg\ is_empty\ d\ \longrightarrow\ invar\ (deqL\ d)
\end{array}$$

The proofs are case analyses over all the defining equations of the non-recursive functions $enqL$ and $deqL'$. In each case, the proof is largely by application of the verified properties for the underlying data structures (see Section 5). As an example of these top-level proofs we present one crucial case of $(*)$:

$$listL\ (enqL\ x\ (Rebal\ (States\ L\ big\ small)))\ =\ x\ \#\ listL\ (Rebal\ (States\ L\ big\ small))$$

assuming that the deque stays in the rebalancing state. We start by defining $states\ =\ States\ L\ big\ small$, $small'\ =\ push\ x\ small$ and $states'\ =\ States\ L\ big\ small'$ as shorthands. Then we can unfold the definition of $enqL$:

$$listL\ (enqL\ x\ (Rebal\ (States\ L\ big\ small)))\ =\ listL\ (step^4\ states')$$

Using the property of the *step* functions preserving list abstractions (see Section 5), we can simply ignore the four rebalancing steps:

$$\ldots \quad = listL\ states'$$

This enables us to unfold the definition of *listL*:

$$\begin{aligned}\ldots &= list\_small\_first\ states' \\ &= \mathsf{let}\ (bs,\ ss') = lists\ states'\ \mathsf{in}\ ss' \ @\ rev\ bs\end{aligned}$$

Now, we can utilize that *push* operations prepend the new element to the list abstractions:

$$\begin{aligned}\ldots &= \mathsf{let}\ (bs,\ x\ \#\ ss) = lists\ states'\ \mathsf{in}\ (x\ \#\ ss) \ @\ rev\ bs \\ &= x\ \#\ (\mathsf{let}\ (bs,\ ss) = lists\ states\ \mathsf{in}\ ss \ @\ rev\ bs)\end{aligned}$$

Concluding the proof, we fold the definition of *listL* again:

$$\begin{aligned}\ldots &= x\ \#\ list\_small\_first\ states \\ &= x\ \#\ listL\ (Rebal\ states)\end{aligned}$$

The required properties of *firstL* and *deqL* are simple corollaries of the above properties for *deqL′*. The dual properties of *enqR* and *deqR′* are again corollaries via these additional properties:

$$\begin{aligned}&invar\ d \longrightarrow listR\ (swap\ d) = listL\ d \\ &invar\ d \longrightarrow invar\ (swap\ d)\end{aligned}$$

5.7 Complexity

All operations of our implementation take constant time because they only employ constant-time functions (arithmetic, (#), *hd*, *tl*) and are not recursive. Some of the auxiliary functions used in the verification are not constant-time but this is irrelevant. Our colour schema helps to distinguish the two worlds.

6 Conclusion

We have presented an implementation of a real-time double-ended queue and in particular the key ingredients of its verification: the abstraction functions, the invariants (incl. all auxiliary functions to define them), and the key theorems about the implementation. It would be interesting to investigate if our invariants could be simplified and if semi-automatic theorem provers like Why3 [3] could automate the proof significantly beyond the current level.

Finally note that our deque implementation is fully executable and that Isabelle can generate code in many functional languages (including Haskell and Scala) from it [5].

References

1 F. Warren Burton. An efficient functional implementation of FIFO queues. *Inf. Process. Lett.*, 14(5):205–206, 1982. doi:10.1016/0020-0190(82)90015-1.

2 Tyng-Ruey Chuang and Benjamin Goldberg. Real-time deques, multihead turing machines, and purely functional programming. In *Functional programming languages and computer architecture - FPCA '93*. ACM Press, 1993. doi:10.1145/165180.165225.

3 Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *European Symposium on Programming (ESOP 2013)*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.

4 Alejandro Gómez-Londoño. Hood-Melville queue. *Archive of Formal Proofs*, January 2021. URL: https://isa-afp.org/entries/Hood_Melville_Queue.html.

5 Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming (FLOPS 2010)*, volume 6009 of *LNCS*, pages 103–117. Springer, 2010.

6 Robert Hood. *The Efficient Implementation of Very-High-Level Programming Language Constructs*. PhD thesis, Cornell University, 1982. TR 82-503.

7 Robert Hood and Robert Melville. Real-time queue operation in pure LISP. *Inf. Process. Lett.*, 13(2):50–54, 1981. doi:10.1016/0020-0190(81)90030-2.

8 Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. Contract-based resource verification for higher-order functions with memoization. In Giuseppe Castagna and Andrew D. Gordon, editors, *Symposium on Principles of Programming Languages, POPL 2017*, pages 330–343. ACM, 2017. doi:10.1145/3009837.3009874.

9 Ravichandhran Madhavan and Viktor Kuncak. Symbolic resource bound inference for functional programs. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification, CAV 2014*, volume 8559 of *LNCS*, pages 762–778. Springer, 2014. doi:10.1007/978-3-319-08867-9_51.

10 Tobias Nipkow, editor. *Functional Data Structures and Algorithms. A Proof Assistant Approach*. ACM Books, Forthcoming. 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 Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

13 Chris Okasaki. Simple and efficient purely functional queues and deques. *J. Funct. Program.*, 5(4):583–592, 1995. doi:10.1017/S0956796800001489.

14 Balazs Toth and Tobias Nipkow. Real-time double-ended queue. *Archive of Formal Proofs*, June 2022. , Formal proof development. URL: https://www.isa-afp.org/entries/Real_Time_Deque.html.

Certifying Higher-Order Polynomial Interpretations

Niels van der Weide
Institute for Computing and Information Sciences, Radboud University, Nijmegen, The Netherlands

Deivid Vale
Institute for Computing and Information Sciences, Radboud University, Nijmegen, The Netherlands

Cynthia Kop
Institute for Computing and Information Sciences, Radboud University, Nijmegen, The Netherlands

Abstract

Higher-order rewriting is a framework in which one can write higher-order programs and study their properties. One such property is termination: the situation that for all inputs, the program eventually halts its execution and produces an output. Several tools have been developed to check whether higher-order rewriting systems are terminating. However, developing such tools is difficult and can be error-prone. In this paper, we present a way of certifying termination proofs of higher-order term rewriting systems. We formalize a specific method that is used to prove termination, namely the polynomial interpretation method. In addition, we give a program that processes proof traces containing a high-level description of a termination proof into a formal Coq proof script that can be checked by Coq. We demonstrate the usability of this approach by certifying higher-order polynomial interpretation proofs produced by Wanda, a termination analysis tool for higher-order rewriting.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Equational logic and rewriting

Keywords and phrases higher-order rewriting, Coq, termination, formalization

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.30

Related Version *Preprint version*: `https://arxiv.org/abs/2302.11892` [32]

Supplementary Material *Software (Coq Formalization)*: `github.com/nmvdw/Nijn` [31]
Software (Proof script generator): `github.com/deividrvale/nijn-coq-script-generation` [30]

Funding *Niels van der Weide*: This research was supported by the NWO project "The Power of Equality" OCENW.M20.380, which is financed by the Dutch Research Council (NWO).
Deivid Vale: Author supported by NWO Top project "Implicit Complexity through Higher-Order Rewriting", NWO 612.001.803/7571.
Cynthia Kop: Author supported by NWO Top project "Implicit Complexity through Higher-Order Rewriting", NWO 612.001.803/7571 and the NWO VIDI project "Constrained Higher-Order Rewriting and Program Equivalence", NWO VI.Vidi.193.075.

Acknowledgements The authors thank Dan Frumin for his help with understanding and using `Ltac`.

1 Introduction

Automatically proving termination is an important problem in term rewriting, and numerous tools have been developed for this purpose, such as AProVE [10], NaTT [35], `MatchBox` [33], Mu-Term [12], SOL [13], $\mathsf{T_TT_2}$ [21] and Wanda [16], which compete against each other in an annual termination competition [11]. Aside from basic (first-order) term rewriting, this includes tools analyzing for instance string, conditional, and higher-order rewriting.

Developing termination tools is a difficult and error-prone endeavor. On the one hand, the termination techniques that are implemented may contain errors. This is particularly relevant in higher-order term rewriting, where the proofs are often very intricate due to


© Niels van der Weide, Deivid Vale, and Cynthia Kop;
licensed under Creative Commons License CC-BY 4.0

partial application, type structure, beta-reduction, and techniques often not transferring perfectly between different formalisms of higher-order rewriting. Hence, it should come as no surprise that errors have been found even in published papers on higher-order rewriting. On the other hand, it is very easy for a tool developer to accidentally omit a test whether some conditions to apply specific termination techniques are satisfied, or to incorrectly translate a method between higher-order formalisms.

To exacerbate this issue, termination proofs are usually complex and technical in nature, which makes it hard to assess the correctness of a prover's output by hand. Not only do many benchmarks contain hundreds of rules, modern termination tools make use of various proof methods that have been developed for decades. Indeed, a single termination proof might, for instance, make use of a combination of dependency pairs [9, 19, 3], recursive path orders [20, 5], rule removal, and multiple kinds of interpretations [8, 23, 34, 18]. This makes bugs very difficult to find.

Hence, there is a need to formally certify the output of termination provers, ideally automatically. There are two common engineering strategies to provide such certification. In the first, one builds the certifier as a library in a proof assistant along with tools to read the prover's output and construct a formal proof, which we call *proof script*. The proof script is then verified by a proof assistant. Examples of this system design are the combinations Cochinelle/CiME3 [7] and CoLoR/Rainbow [6]. In the second, the formalization includes certified algorithms for checking the correctness of the prover's output. This allows for the whole certifier to be extracted, using code extraction, and be used as a standalone program. Hence, the generation of proof scripts by a standalone tool is not needed in this approach, but it comes with a higher formalization cost. IsaFoR/CeTA [28] utilizes this approach.

When it comes to higher-order rewriting, however, the options are limited. Both Cochinelle [7] and IsaFoR/CeTA [28] only consider first-order rewriting. CoLoR/Rainbow [6] does include a formalization of an early definition of HORPO [20]. Since here we use a different term formalism compared to that of [20], our results are not directly compatible. See for instance [2, 25] for more formalization results in rewriting.

In this paper, we introduce a new combination Nijn/ONijn for the certification of higher-order rewriting termination proofs. We follow the first aforementioned system design: Nijn is a Coq library providing a formalization of the underlying higher-order rewriting theory and ONijn is a proof script generator that given a minimal description of a termination proof (which we call *proof trace*), outputs a Coq proof script. The proof script then utilizes results from Nijn for checking the correctness of the traced proof. The schematic below depicts the basic steps to produce proof certificates using Nijn/ONijn.

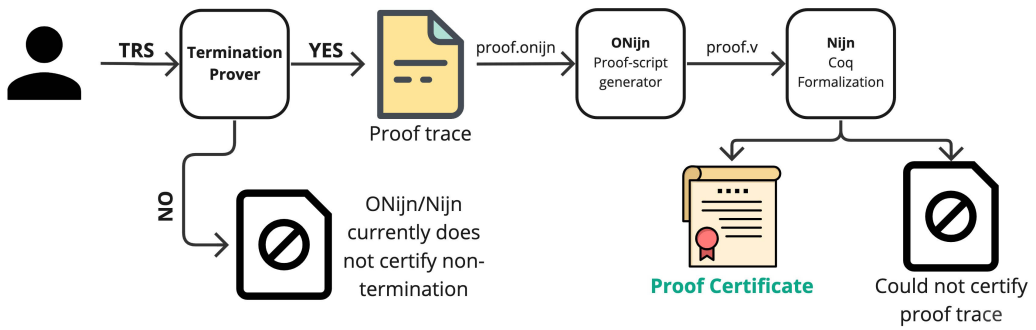


Figure 1 Nijn/ONijn schematics.

While Nijn is the certified core part of our tool since it is checked by Coq, the proof script generation implemented in OCaml (ONijn) is not currently certified and must be trusted. For this reason, we deliberately keep ONijn as simple as possible and no checking or computation is done by it. The only task delegated to ONijn is that of parsing the proof trace given by the termination prover to a Coq proof script. Additionally, checking the correctness of polynomial termination proofs in Coq is an inherently incomplete task, since it would require a method to solve inequalities over arbitrary polynomials, which is undecidable in general.

Contributions. The main contribution of this paper can be summarized as follows:

- we provide a formalization of higher-order algebraic functional systems (Definition 2.6);
- a formal proof of the interpretation method using weakly monotonic algebras (Theorem 3.11);
- a formalization of the higher-order polynomial method (Theorem 4.7);
- a tactic that automatically solves the constraints that arise when using the higher-order polynomial method (Section 4.3);
- an OCaml program that transforms the output of a termination prover into a Coq script that represents the termination proof (Section 5).

Technical Overview. This paper orbits Nijn, a Coq library formalizing higher-order rewriting [31]. The formalization is based on intensional dependent type theory extended with two axioms: *function extensionality* and *uniqueness of identity proofs* [14]. Currently, the termination criterion formalized in the library is *the higher-order polynomial method*, introduced in [8]. The tool `coqwc` counts the following amount of lines of code:

```
spec    proof comments
5497     1985      272 total
```

The higher-order interpretation method roughly works as follows. First, types are interpreted as well-ordered structures (Definition 3.3), compositionally. For instance, we interpret base types as natural numbers (with the usual ordering). Then we interpret a functional type $A \Rightarrow B$ as the set of weakly monotonic functions from $(\!|A|\!)$ to $(\!|B|\!)$ where $(\!|A|\!)$, $(\!|B|\!)$ denote the interpretations of A, B respectively. The second step is to map inhabitants of a type A to elements of $(\!|A|\!)$, which is expressed here by Definition 3.9.

This interpretation, called *extended monotonic algebras* in [8], alone does not suffice for termination. To guarantee termination, we interpret both term application (Definition 4.6) and function symbols as strongly monotonic functionals. In addition, terms must be interpreted in such a way that the rules of the system are strictly oriented, i.e., $[\![\ell]\!] > [\![r]\!]$, for all rules $\ell \to r$. This means that whenever a rewriting is fired in a term, the interpretation of that term strictly decreases. As such, termination is guaranteed. Here we use *termination models* (Definition 3.10) to collect these necessary conditions.

The main result establishing the correctness of this technique in the higher-order case is expressed by Theorem 3.11. To the reader familiar with *the interpretation method* in first-order rewriting, Theorem 4.7 would be no surprise. It is essentially the combination of the Manna–Ness criterion with higher-order polynomials and the additional technicalities that are needed for the higher-order case.

2 The Basics of Higher-Order Rewriting in Coq

In this section, we introduce the basic constructs needed to formalize *algebraic functional systems* (AFSs) like types, contexts, variables, terms, and rewriting rules. We end the section with an exposition on how to express termination constructively in Coq.

2.1 Terms and Rewrite Rules

Let us start by defining *simple types*.

▶ **Definition 2.1** (`ty`). **Simple types** over a type `B` are defined as follows:

```
Inductive ty (B : Type) : Type :=
| Base : B → ty B
| Fun : ty B → ty B → ty B.
```

Elements of `B` are called **base types**. Every inhabitant `b : B` gives rise to a simple type `Base b` and if `A1`, `A2` are simple types then so is `Fun A1 A2`. We write `A1 ⟶ A2` for `Fun A1 A2`.

We need *(variable) contexts* in order to type terms that may contain free variables. Conceptually, a context is a list of variables with their respective types. For instance, $[x_0 : A_0; \ldots; x_n : A_n]$ is the context with variables x_0 of type A_0, …, x_n of type A_n. However, we use nameless variables in our development, so we do not keep track of their names. Consequently, a context is represented by a list of types. Then we only consider the list $[A_0, \ldots, A_n]$. However, we still need to refer to the free variables in terms. In order to do so, we represent them through indexing positions in the context. For instance, in the context $[A_0; \ldots; A_n]$ we have $n + 1$ position indexes $0, 1, \ldots, n$, which we use as variables.

▶ **Definition 2.2** (`con`). The type of **variable contexts** over a type `B` is defined as follows.

```
Inductive con (B : Type) : Type :=
| Empty : con B
| Extend : ty B → con B → con B.
```

We write • for `Empty` and `A ,, C` for `Extend A C`.

▶ **Definition 2.3** (`var`). We define the type `var C A` of **variables** of type `A` in context `C` as

```
Inductive var {B : Type} : con B → ty B → Type :=
| Vz : forall {C : con B} {A : ty B}, var (A ,, C) A
| Vs : forall {C : con B} {A1 A2 : ty B}, var C A2 → var (A1 ,, C) A2.
```

Let us consider an example of a context and some variables. Suppose that we have a base type denoted by `b`. Then we can form the context `Base b ,, Base b ⟶ Base b ,, Empty`. In this context, we have two variables. The first one, which is `Vz`, has type `Base b`, and the second variable, which is `Vs Vz`, has type `Base b ⟶ Base b`, The context that we discussed corresponds to $[x_0 : b; x_1 : b \longrightarrow b]$. The variable `Vz` represents x_0, while `Vs Vz` represents x_1.

In Definition 2.4 below we define the notion of *well-typed terms-in-context* which consists of those expressions such that there is a typing derivation. We use dependent types to ensure well-typedness of such expressions. The type of terms depends on a simple type `A : ty B` (which represents the object-level type of the expression) and context `C : con B` that carries the types of all free variables in the term. We also need to type function symbols. Hence, we require a type `F : Type` of function symbols and `ar : F → ty B`, which maps `f : F` to a simple type `ar f`.

▶ **Definition 2.4** (`tm`). We define the type of **well-typed terms** as follows

```
Inductive tm {B : Type} {F : Type} (ar : F → ty B) (C : con B) : ty B → Type :=
| BaseTm : forall (f : F), tm ar C (ar f)
| TmVar : forall {A : ty B}, var C A → tm ar C A
| Lam : forall {A1 A2 : ty B}, tm ar (A1 ,, C) A2 → tm ar C (A1 ⟶ A2)
| App : forall {A1 A2 : ty B}, tm ar C (A1 ⟶ A2) → tm ar C A1 → tm ar C A2.
```

For every function `f : F` we have a term `BaseTm f` of type `ar f`. Every variable `v` gives rise to a term `TmVar v`. For λ-abstractions, given a term `s : tm ar (A1 ,, C) A2`, there is a term λ `s : tm ar C (A1 ⟶ A2)`, namely `Lam s`. The last constructor represents term application. If we have a term `s : tm ar C (A1 ⟶ A2)` and a term `t : tm ar C A1`, we get a term `s · t : tm ar C A2`, which is defined to be `App s t`.

While it may be more cumbersome to write down terms using de Bruijn indices, it does have several advantages. Most importantly, it eliminates the need for α-equivalence, so that determining equality between terms is reduced to a simple syntactic check.

Our notion of rewriting rules deviates slightly from the presentation in [8]. Mainly, we do not impose the pattern restriction on the left-hand side of rules nor that free variables on the right-hand side occur on the left-hand side. This choice simplifies the formalization effort because when defining a concrete TRS, one does not need to check this particular condition. Note that in IsaFoR [28] the same simplification is used

▶ **Definition 2.5** (`rewriteRule`). The type of **rewrite rules** is defined as follows:

```
Record rewriteRule {B : Type} {F : Type} (ar : F → ty B) :=
 make_rewrite {
    vars_of : con B ;
    tar_of : ty B ;
    lhs_of : tm ar vars_of tar_of ;
    rhs_of : tm ar vars_of tar_of }.
```

The context `vars_of` carries the variables used in the rule, and the type `tar_of` is used to guarantee that both the `lhs_of` and `rhs_of` are terms of the same type.

▶ **Definition 2.6** (`afs`). The type of **algebraic functional systems** is defined as follows

```
Record afs (B : Type) (F : Type) :=
make_afs { arity : F → ty B ; list_of_rewriteRules : list (rewriteRule arity) }.
```

As usual, every AFS induces a rewrite relation on the set of terms, which we denote by $\sim>$. The formal definition is found in `RewritingSystem.v`. The rewrite relation $\sim>$ is defined to be the closure of the one-step relation under transitivity and compatibility with the term constructors. In Coq, we use an inductive type to define this relation. Each rewrite step is represented by a constructor. More specifically, we have a constructor for rewriting the left-hand and the right-hand side of an application, we have a constructor for β-reduction, and we have a constructor for the rewrite rules of the AFS.

▶ **Example 2.7** (`map_afs`). Let us encode $\mathcal{R}_{\mathsf{map}}$ in Coq. It is composed of two rules: $\mathsf{map}\, F\, \mathsf{nil} \to \mathsf{nil}$ and $\mathsf{map}\, F\, (\mathsf{cons}\, x\, xs) \to \mathsf{cons}\, (F\, x)\, (\mathsf{map}\, F\, xs)$. We start with base types.

```
Inductive base_types := TBtype | TList.
Definition Btype : ty base_types := Base TBtype.
Definition List : ty base_types := Base TList.
```

The abbreviations `Btype` and `List` is to smoothen the usage of the base types. There are three function symbols in this system:

```
Inductive fun_symbols := TNil | TCons | TMap.
```

The arity function `map_ar` maps each function symbol in `fun_symbols` to its type.

```
Definition map_ar f : ty base_types
  := match f with
     | TNil ⇒ List
     | TCons ⇒ Btype ⟶ List ⟶ List
     | TMap ⇒ (Btype ⟶ Btype) ⟶ List ⟶ List
     end.
```

So, `TNil` is a list and given an inhabitant of `Btype` and `List`, the function symbol `TCons` gives a `List`. Again we introduce some abbreviations to simplify the usage of the function symbols.

```
Definition Nil {C} : tm map_ar C _ := BaseTm TNil.
Definition Cons {C} x xs : tm map_ar C _ := BaseTm TCons · x · xs.
Definition Map {C} f xs : tm map_ar C _ := BaseTm TMap · f · xs.
```

The first rule, $\mathsf{map}\, F\, \mathsf{nil} \to \mathsf{nil}$, is encoded as the following Coq construct:

```
Program Definition map_nil :=
  make_rewrite
    (_ ,, •) _
    (let f := TmVar Vz in Map · f · Nil)
    Nil.
```

Notice that we only defined the *pattern* of the first two arguments of `make_rewrite`, leaving the types in the context `(_ ,, •)` and the type of the rule unspecified. Coq can fill in these holes automatically, as long as we provide a context pattern of the correct length. In this particular rewrite rule, there is only one free variable. As such, the variable `TmVar Vz` refers to the only variable in the context. In addition, we use iterated `let`-statements to imitate variable names. For every position in the context, we introduce a variable in Coq, which we use in the left- and right-hand sides of the rule. This makes the rules more human-readable. Indeed, the lhs $\mathsf{map}\, F\, \mathsf{nil}$ of this rule is represented as `Map · f · Nil` in code. The second rule for `map` is encoded following the same ideas.

```
Program Definition map_cons :=
  make_rewrite
    (_ ,, _ ,, _ ,, •) _
    (let f := TmVar Vz in let x := TmVar (Vs Vz) in let xs := TmVar (Vs (Vs Vz)) in
    Map · f · (Cons · x · xs))
    (let f := TmVar Vz in let x := TmVar (Vs Vz) in let xs := TmVar (Vs (Vs Vz)) in
    Cons · (f · x) · (Map · f · xs)).
```

Putting this all together, we obtain an AFS, which we call `map_afs`.

```
Definition map_afs := make_afs map_ar (map_nil :: map_cons :: nil).
```

2.2 Termination

Strong normalization is usually defined as the absence of infinite rewrite sequences. Such a definition is sufficient in a classical setting where the law of excluded middle holds. However, we work in a constructive setting, and thus we are interested in a stronger definition.

Therefore, we need a constructive predicate, formulated positively, which implies there are no infinite rewrite sequences. This idea is captured by the following definition

▶ **Definition 2.8** (`WellfoundedRelation.v`). The **well-foundedness predicate** for a relation `R` is defined as follows

```
Inductive isWf {X : Type} (R : X → X → Type) (x : X) : Prop :=
| acc : (forall (y : X), R x y → isWf R y) → isWf R x.
```

A relation is **well-founded** if the well-foundedness predicate holds for every element.

```
Definition Wf {X : Type} (R : X → X → Type) :=
  forall (x : X), isWf R x.
```

Note that this definition has been considered numerous times before, for example in [4] and in CoLoR [6]. An element `x` is well-founded if all `y` such that `R x y` are well-founded. Note that if there is no `y` such that `R x y`, then `x` is vacuously well-founded. From the rewriting perspective, this definition properly captures the notion of strong normalization. Indeed, a term s is strongly normalizing iff every s' such that s rewrites to s' is strongly normalizing.

Well-foundedness contradicts the existence of infinite rewrite sequences, even in a constructive setting. As such, it indeed gives a stronger condition.

▶ **Proposition 2.9** (`no_infinite_chain`). *If R is well-founded, then there is no infinite sequence $s_0, s_1, \ldots$ such that $R(s_n, s_{n+1})$, for all n.*

Next, we define strong normalization using well-founded predicates.

▶ **Definition 2.10** (`is_SN`). An algebraic functional system is **strongly normalizing** if for every context `C` and every type `A` the rewrite relation for terms of type `A` in context `C` is well-founded. We formalize that as follows:

```
Definition isSN {B F : Type} (X : afs B F) : Prop :=
  forall (C : con B) (A : ty B), Wf (fun (t1 t2 : tm X C A) ⇒ t1 ∼> t2).
```

3 Higher-Order Interpretation Method

In this section, we formalize the method of weakly monotonic algebras for algebraic functional systems. We proceed by providing type-theoretic semantics for the syntactic constructions introduced in the last section and a sufficient condition for which such semantics can be used to establish strong normalization.

3.1 Interpreting types and terms

In weakly monotonic algebras, types are interpreted as sets along with a well-founded ordering and a quasi-ordering [24, 8]. For that reason, we start by defining *compatible relations*. Intuitively, these are the domain for our semantics.

▶ **Definition 3.1** (`CompatibleRelation.v`). **Compatible relations** are defined as follows

```
Record CompatRel := {
  carrier :> Type ;
  gt : carrier → carrier → Prop ;
  ge : carrier → carrier → Prop }.
```

We write `x > y` and `x >= y` for `gt x y` and `ge x y` respectively.

The record `CompatRel` consists of the data needed to express compatibility between $>$ and $\geq$. The conditions it needs to satisfy, are in the type class `isCompatRel`, defined below.

```
Class isCompatRel (X : CompatRel) := {
  gt_trans : forall {x y z : X}, x > y → y > z → x > z ;
  ge_trans : forall {x y z : X}, x >= y → y >= z → x >= z ;
  ge_refl : forall (x : X), x >= x ;
  compat : forall {x y : X}, x > y → x >= y ;
  ge_gt : forall {x y z : X}, x >= y → y > z → x > z ;
  gt_ge : forall {x y z : X}, x > y → y >= z → x > z }.
```

Note that the field `gt_trans` in `isCompatRel` follows from `compat` and `ge_gt`. The type `nat` of natural numbers with the usual orders is a first example of data that satisfies `isCompatRel`. We denote this one by `nat_CompatRel`. This type class essentially models the notion of extended well-founded set introduced in [18]. An **extended well-founded set** is a set together with compatible orders $>, \geq$ such that $>$ is well-founded and $\geq$ is a quasi-ordering. This compatibility requirement corresponds to the axiom `compat` in the type class `isCompatRel`. However, since we do not require $>$ to be well-founded in this definition, we instead call it a compatible relation. More specifically, `X` is a compatible relation if it is of type `CompatRel` and satisfies the constraints in the type class `isCompatRel`.

In order to interpret simple types (Definition 2.1), we start by fixing a type `B : Type` of base types and an interpretation `semB : B → CompatRel` such that each `semB b` is a compatible relation. Whenever `semB` satisfies such property we call it an **interpretation key** for `B`. We interpret arrow types as functional compatible relations, i.e., compatible relations such that the inhabitants of their carrier are functional. The class of functionals we are interested in is that of *weakly-monotone maps*.

▶ **Definition 3.2** (`MonotonicMaps.v`)**. Weakly monotone maps** are defined as follows

```
Class weakMonotone {X Y : CompatRel} (f : X → Y) :=
  map_ge : forall (x y : X), x >= y → f x >= f y.

Record weakMonotoneMap (X Y : CompatRel) :=
  make_monotone {
    fun_carrier :> X → Y ;
    is_weak_monotone : weakMonotone fun_carrier }.
```

The class `weakMonotone` says when a function is weakly monotonic, and an inhabitant of the record `weakMonotoneMap` consists of a function together with proof of its weak monotonicity. Then we define `fun_CompatRel` which is of type `CompatRel` and represents the **functional compatible relations** from `X` to `Y`. It is defined as follows:

```
Definition fun_CompatRel (X Y : CompatRel) : CompatRel :={|
  carrier := weakMonotoneMap X Y ;
  gt f g := forall (x : X), f x > g x ;
  ge f g := forall (x : X), f x >= g x |}.
```

In what follows, we write `X →wm Y` for `fun_CompatRel X Y`. The semantics for a type is parametrized by an interpretation key `semB`. It is defined as follows:

▶ **Definition 3.3** (`sem_Ty`). Assume `A : ty B` and `semB` is an interpretation key for `B`. Then

```
Fixpoint sem_Ty (A : ty B) : CompatRel :=
  match A with
  | Base b    ⇒ semB b
  | A1 → A2   ⇒ sem_Ty A1 →wm sem_Ty A2
  end.
```

We also show how to interpret contexts, and to do so, we need to interpret the empty context and context extension. For those, we define the unit and product of compatible relations.

▶ **Definition 3.4** (`Examples.v`). The **unit** and **product** compatible relations:

```
Definition unit_CompatRel :
CompatRel := {|
  carrier := unit ;
  gt _ _ := False ;
  ge _ _ := True |}.
```

```
Definition prod_CompatRel (X Y : CompatRel) :
CompatRel := {|
  carrier := X * Y ;
  gt x y := fst x > fst y ∧ snd x > snd y ;
  ge x y := fst x >= fst y ∧ snd x >= snd y |}.
```

Note that `unit_CompatRel` is the compatible relation on the type with only one element, for which the ordering is trivial. In addition, `prod_CompatRel` is the compatible relation on the product, for which we compare elements coordinate-wise. We write `X * Y` for `prod_CompatRel X Y`.

▶ **Definition 3.5** (`sem_Con`). Contexts are interpreted as follows

```
Fixpoint sem_Con (C : con B) : CompatRel :=
  match C with
  | •       ⇒ unit_CompatRel
  | A ,, C  ⇒ sem_Ty A * sem_Con C
  end.
```

Next, we give semantics to variables and terms. The approach we use here is slightly different from what is usually done in higher-order rewriting. In [8, 18, 24], for instance, context information is lifted to the meta-level and variables are interpreted using the notion of valuations. In contrast, in our setting, the typing context lives at the syntactic level and variables are interpreted as weakly monotonic functions. Consequently, to every term `t : tm C A`, we assign a map from `sem_Con C` to `sem_Ty A`. In the remainder, we need the following weakly monotonic functions.

▶ **Definition 3.6** (`Examples.v`). We define the following weakly monotonic functions.

- Given `y : Y`, we write `const_wm y : X →wm y` for the constant function.
- Given `f : X →wm Y` and `g : Y →wm Z`, we define `g ∘wm f : X →wm Z` to be their composition.
- We have the first projection `fst_wm : X * Y →wm X`, which sends a pair `(x , y)` to `x`, and the second projection `snd_wm : X * Y →wm Y`, which sends `(x , y)` to `y`.
- Given `f : X →wm Y` and `g : X →wm Z`, we have a function `⟨ f , g ⟩ : X →wm (Y * Z)`. For `x : X`, we define `⟨ f , g ⟩ x` to be `(f x , g x)`.
- Given `f : Y * X →wm Z`, we get `λwm f : X →wm (Y →wm Z)`. For every `x : X` and `y : Y`, we define `λwm f y x` to be `f (y , x)`.
- Given `f : X →wm (Y →wm Z)` and `x : X →wm Y`, we obtain `f ·wm x : X →wm Z`, which sends every `a : X` to `f a (x a)`.
- Given `x : X`, we have a weakly monotonic function `apply_el_wm x : (X →wm Y) →wm Y` which sends `f : X →wm Y` to `f x`.

Recall that variables are represented by positions in a context which in turn is interpreted as a weakly monotonic product (Definition 3.5). This allows us to interpret the variable at a position in a context as the corresponding interpretation of the type in that position.

▶ **Definition 3.7** (sem_Var)**.** We interpret variables with the following function

```
Fixpoint sem_Var {C : con B} {A : ty B} (v : var C A) : sem_Con C →wm sem_Ty A
  := match v with
    | Vz ⇒ fst_wm
    | Vs v ⇒ sem_Var v ∘wm snd_wm
    end.
```

We need the following data in order to provide semantics to terms. An arity function `ar : F → ty B`, together with its interpretation `semF : forall (f : F), sem_Ty (ar f)`, and an *application operator* given by

```
semApp : forall (A1 A2 : ty B), (sem_Ty A1 →wm sem_Ty A2) ∗ sem_Ty A1 →wm sem_Ty A2
```

to interpret term application.

▶ **Remark 3.8.** A first, but incorrect, guess to interpret application would have been by interpreting the application of `f : sem_Ty A1 →wm sem_Ty A2` to `x : sem_Ty A1` by `f x`. However, there is a significant disadvantage of this interpretation. Ultimately, we want to deduce strong normalization from the interpretation, and the main idea is that if we have a rewrite `x ∼> x'`, then we have `semTm x > semTm x'`. This requirement would not be satisfied if we interpret application of our terms as actual applications as functions. Indeed, if we have `x < x'`, then one is not guaranteed that we also have `f x < f x'`, because `f` is only weakly monotone.

There are two ways to deal with this. One way is by interpreting function types as strictly monotonic maps [18]. In this approach, this interpretation of application is valid. However, it comes at a price, because the interpretation of lambda abstraction becomes more difficult.

Another approach, which we use here, is also used in [8]. We add a parameter to our interpretation method, namely `semApp`, which abstractly represents the interpretation of application. To deduce strong normalization in this setting, we add requirements about `semApp` in Section 3.2. As a result, in concrete instantiations of this method, we need to provide an actual definition for `semApp`. We see this in Section 4.2.

▶ **Definition 3.9** (sem_Tm)**.** Given a function `semF : forall (f : F), sem_Ty (ar f)`, the semantics of terms is given by

```
Fixpoint sem_Tm {C : con B} {A : ty B} (t : tm ar C A) : sem_Con C →wm sem_Ty A :=
  match t with
  | BaseTm f  ⇒ const_wm (semF f)
  | TmVar v   ⇒ sem_Var v
  | λ f       ⇒ λwm (sem_Tm f)
  | f · t     ⇒ semApp _ _ ∘wm ⟨sem_Tm f , sem_Tm t ⟩
  end.
```

Notice that we could have chosen a fixed way of interpreting application. We follow the same approach used by Fuhs and Kop [8] in our formalization and leave `semApp` abstract. This choice is essential if we want to use the interpretation method for both *rule removal* and the *dependency pair* approach. See [15, Chapters 4 and 6] for more detail.

3.2 Termination Models for AFSs

Now we have set up everything that is necessary to define the main notion of this section: *termination models*. From a termination model of an algebraic functional system, one obtains an interpretation of the types and terms. In addition, every rewrite rule is "satisfied" in this interpretation.

▶ **Definition 3.10** (`Interpretation`). Let $\mathcal{R}$ be an algebraic functional system with base type `B` and function symbols `F`. A **termination model** of $\mathcal{R}$ consists of

- an interpretation key `semB`;
- a function `semF : forall (f : F), sem_Ty (ar f)`;
- a function

```
semApp : forall (A1 A2 : ty B), (sem_Ty A1 →wm sem_Ty A2) * sem_Ty A1 →wm sem_Ty A2
```

such that the following axioms are satisfied

- each `semB b` is well-founded and inhabited;
- if `f > f'`, then `semApp _ _ (f , x) > semApp _ _ (f' , x)`;
- if `x > x'`, then `semApp _ _ (f , x) > semApp _ _ (f , x')`;
- we have `semApp _ _ (f , x) >= f x` for all `f` and `x`;
- for every rewrite rule `r`, substitution `s`, and element `x`, we have

```
semTm (lhs r [ s ]) x > semTm (rhs r [ s ]) x.
```

Whereas the left-hand side of every rewrite rule is greater than its right-hand side, this does not hold for β-reduction in our interpretations. Since rewrite sequences can contain both rewrite rules and β-reduction, such sequences are not guaranteed to strictly decrease. As such, we need more to actually conclude strong normalization, and we follow the method used by Kop [15]. More specifically, Kop uses *rule removal* to show that strong normalization follows from the strong normalization of β-reduction, which is a famous theorem proven by Tait [27]. The strong normalization of β-reduction has been formalized numerous times and an overview can be found in [1]. Now we deduce the main theorem of this section.

▶ **Theorem 3.11** (`afs_is_SN_from_Interpretation`). *Let $\mathcal{R}$ be an algebraic functional system. If we have a termination model of X, then X is strongly normalizing.*

4 The Higher-Order Polynomial Method

4.1 Polynomials

In this section, we instantiate the material of Section 3 to a concrete instance, namely *the polynomial method* [8]. For that reason, we define the notation of *higher-order polynomial*.

▶ **Definition 4.1** (`Polynomial.v`). We define the type `base_poly` of **base polynomials** and `poly` of **higher-order polynomials** by mutual induction as follows:

```
Inductive base_poly {B : Type}
    : con B → Type :=
| P_const : forall {C : con B},
    nat → base_poly C
| P_plus : forall {C : con B},
    (P1 P2 : base_poly C) → base_poly C
| P_mult : forall {C : con B},
    (P1 P2 : base_poly C)→ base_poly C
| from_poly : forall {C : con B} {b : B},
    poly C (Base b) → base_poly C
```

```
with poly {B : Type} : con B → ty B → Type :=
| P_base : forall {C : con B} {b : B},
  base_poly C →poly C (Base b)
| P_var : forall {C : con B} {A : ty B},
  var C A → poly C A
| P_app : forall {C : con B} {₁A ₂A : ty B},
  poly C (₁A ⟶₂ A)→
   poly C ₁A→
   poly C ₂A
| P_lam : forall {C : con B} {₁A ₂A : ty B},
  poly (₁A ,, C) ₂A → poly C (₁A ⟶₂ A).
```

We can make expressions of base polynomials using `P_const` (constants), `P_plus` (addition), and `P_mult` (multiplication). In addition, `from_poly` takes an inhabitant of `poly C (Base b)` and returns a base polynomial in context `C`. Using `P_base`, we can turn a base polynomial into a polynomial of any base type. The constructors, `P_var`, `P_app`, and `P_lam`, are remniscent of the simply typed lambda calculus. We get variables from `P_var`, λ-abstraction from `P_lam`, and application from `P_app`. Note that combining `from_poly` and `P_var`, we can use variables in base polynomials.

Let us make some remarks about the design choices we made and how they affected the definition of polynomials. One of our requirements is that we are able to add and multiply polynomials on different base types. This is frequently used in actual examples, such as Example 4.2. Function symbols might use arguments from different base types, and we would like to use both of them in polynomial expressions.

One possibility would have been to only work with the type `poly` and to add a constructor

```
P_plus : forall {C : con B} (b1 b2 : B),
        poly C (Base b1) → poly C (Base b2) → poly C (Base b1)
```

However, we refrained from doing so: if we were to use `P_const`, then the elaborator would be unable to determine the actual type if we do not tell the base type explicitly. Instead, we used a type of base polynomials that does not depend on the actual base type. This is the role of `base_poly`, which only depends on the variables being used. We can freely add and multiply inhabitants of `base_poly`, and if we were to use a constant, then we do not explicitly need to mention the base type. In addition, we are able to transfer between `base_poly` and `poly C (Base b)`, and that is what `P_base` and `from_poly` enable us to do.

Note that our definition of higher-order polynomials is rather similar to the one given by Fuhs and Kop [8, Definition 4.1]. They define a set $\mathsf{Pol}(X)$, which consists of polynomial expressions, and for every type A a set $\mathsf{Pol}^A(X)$. The set $\mathsf{Pol}^A(X)$ is defined by recursion: for base type, it is the set of polynomials over X and for function types $A_1 \longrightarrow A_2$, it consists of expressions $\Lambda(y : A_1).P$ where P is a polynomial of type A_2 using an extra variable $y : A_1$. Our `base_poly C` and `poly C A` correspond to $\mathsf{Pol}(X)$ and $\mathsf{Pol}^A(X)$ respectively. However, there are some differences. First of all, Fuhs and Kop require variables to be fully applied, whereas we permit partially applied variables. Secondly, Fuhs and Kop define polynomials in such a way that for every two base types b_1, b_2 the types $\mathsf{Pol}^{b_1}(X)$ and $\mathsf{Pol}^{b_2}(X)$ are equal. This is not the case in our definition: instead we use constructors `from_poly` and `P_base` to relate `base_poly C` and `poly C (Base b)`.

In the polynomial method, the interpretation key sends every base type to `nat_CompatRel`, and in what follows, we write ⟦ `C` ⟧`con` and ⟦ `A` ⟧`ty` for the interpretation of contexts and types respectively. Note that every polynomial `P : poly C A` gives rise to a weakly monotonic

function `sem_poly P : ⟦ C ⟧con →wm ⟦ A ⟧ty` and that every base polynomial `P : base_poly C` gives rise to `sem_base_poly P : ⟦C ⟧con →wm nat_CompatRel`. These two functions are defined using mutual recursion and every constructor is interpreted in the expected way: `sem_poly`.

In order to actually use `base_poly C` and `poly C A`, we provide convenient notations for operations on polynomials. More concretely, we define notations $+$, $*$, and $\cdot$`P` that represent addition, multiplication, and application respectively. These operations must be overloaded since we need to be able to add polynomials of different types. To do so, we similarly use type classes to MathClasses [26]. For details, we refer the reader to the formalization.

▶ **Example 4.2** (`map_fun_poly`)**.** We continue with Example 2.7 and provide a polynomial interpretation to the system `map_afs` as follows:

```
Definition map_fun_poly fn_symbols : poly •(arity trs fn_symbols) :=
  match fn_symbols with
  | Tnil ⇒ to_Poly (P_const 3)
  | Tcons  ⇒ λP λP let y1 := P_var Vz in
    to_Poly (P_const 3 + P_const 2 * y1)
  | Tmap  ⇒ λP let y0 := P_var (Vs Vz) in λP let G1 := P_var Vz in
    to_Poly (P_const 3 * y0 + P_const 3 * y0 * (G1 ·P (y0)))
  end.
```

Informally, the interpretation of `nil` is the constant 3. The interpretation of `cons` is the function that sends $y_1 : \mathbb{N}$ to $3 + 2y_1$, and `map` is interpreted as the function that sends $y_0 : \mathbb{N}$ and $G_1 : \mathbb{N} \to_{\text{wm}} \mathbb{N}$ to $3y_0 + 3y_0G_1(y_0)$.

4.2 Polynomial Interpretation

Using polynomials, we deduce strong normalization under certain circumstances using Theorem 3.11. Suppose that for all function symbols `f` we have a polynomial `J : poly • (arity X f)`, and now we need to provide the interpretation for application. Following Fuhs and Kop [8], we use a general method to interpret application. We start by constructing a minimal element in the interpretation of every type.

▶ **Definition 4.3** (`min_el_ty`)**.** For every simple type `A` we define a minimal element of `⟦ A ⟧ty` as follows

```
Fixpoint min_el_ty (A : ty B) : minimal_element ⟦A ⟧ty
  := match A with
     | Base _ ⇒ nat_minimal_element
     | A1 ⟶ A2 ⇒ min_el_fun_space (min_el_ty A2)
     end.
```

Here `nat_minimal_element` is defined to be 0, and `min_el_fun_space (min_el_ty A2)` is the constant function on `(min_el_ty A2)`.

In order to define the semantics of application, we need several operations involving `⟦ A ⟧ty`. First, we consider *lower value functions*.

▶ **Definition 4.4** (`lvf`)**.** We define the **lower value function** as follows

```
Fixpoint lvf {A : ty B} : ⟦ A ⟧ty →wm nat_CompatRel :=
  match A with
    | Base _ ⇒ id_wm
    | A1 ⟶ A2 ⇒ lvf ∘wm apply_el_wm (min_el_ty A1)
  end.
```

Note that we construct `lvf` directly as a weakly monotonic function. In addition, we reuse the combinators defined in Definition 3.6. As such, we do not need to prove separately that this function is monotonic.

In Kop and Fuhs [8], this definition is written down in a different, but equivalent, way. Instead of defining lvf_A recursively, they look at full applications, which would be more complicated in our setting. More specifically, since we are working with simple types, we must have that $A = A_1 \longrightarrow \ldots \longrightarrow A_n \longrightarrow B$. Then they define $\mathsf{lvf}_A(f) := f(\bot_{A_1}, \ldots, \bot_{A_n})$, where $\bot_A$ is the minimum element of the interpretation of A. Next, we define two addition operations on ⟦ `A` ⟧`ty`.

▶ **Definition 4.5** (`plus_ty_nat`)**.** Addition of natural numbers and elements on ⟦ `A` ⟧`ty` is defined as follows

```
Fixpoint plus_ty_nat {A : ty B} : ⟦A⟧ty ∗ nat_CompatRel →wm ⟦A⟧ty
  := match A with
     | Base _ ⇒ plus_wm
     | A1 ⟶ A2 ⇒
       let f := fst_wm ∘wm snd_wm in
       let x := fst_wm in
       let n := snd_wm ∘wm snd_wm in
       λwm (plus_ty_nat ∘wm ⟨f ·wm x , n⟩)
     end.
```

The function `plus_ty_nat` allows us to add arbitrary natural numbers to elements of the interpretation of types. Note that there are two cases in Definition 4.5. First of all, the type `A` could be a base type. In that case, we are adding two natural numbers, and we use the usual addition operation. In the second case, we are working with a functional type `A1` ⟶ `A2`. The resulting function is defined using pointwise addition with the relevant natural number. Now we have everything in place to define the interpretation of application.

▶ **Definition 4.6** (`p_app`)**.** Application is interpreted as the following function

```
Definition p_app {A1 A2 : ty B}
  : ⟦ A1 ⟶ A2 ⟧ty ∗ ⟦ A1 ⟧ty →wm ⟦ A2 ⟧ty
  := let f := fst_wm in
     let x := snd_wm in
     plus_ty_nat ∘wm ⟨f ·wm x , lvf ∘wm x⟩.
```

If both `A1` and `A2` are base types, then `p_app` (`f` , `x`) reduces to `f x` + `x`. Note that `p_app` satisfies the requirements from Theorem 3.11. Hence, we obtain the following.

▶ **Theorem 4.7** (`poly_Interpretation`)**.** *Let $\mathcal{R}$ be an AFS. Suppose that for every function symbol* `f` *we have a polynomial* `p_fun_sym f` *such that for all rewrite rules* `l` ∼> `r` *in $\mathcal{R}$ we have* `semTm l x` > `semTm r x` *for all* `x`*. Then $\mathcal{R}$ has a termination model.*

4.3 Constraint Solving Tactic

Notice that in order to formally verify a proof of termination of a system using Theorem 4.7, we need to provide a polynomial interpretation and show that $[\![\ell]\!] > [\![r]\!]$ holds for all rules $\ell \to r$. This will introduce inequality proof goals into the Coq context that must be solved.

▶ **Example 4.8.** Let us consider a concrete example. We use the polynomials given in Example 4.2 to show strong normalization of Example 2.7. This example introduces two inequalities, one for each rule. Let $G_0 : \mathbb{N} \to_{\text{wm}} \mathbb{N}$ be weakly monotonic. For rule `map_nil`, we need to prove that for all G_0, the constraint $12 + G_0(0) + 9G_0(3) > 3$ holds. For the second rule, `map_cons`, the constraint is: $12 + 4y_0 + 12y_1 + G_0(0) + (3y_0 + 9y_1 + 9)G_0(3 + y_0 + 3y_1) > 3 + y_0 + 12y_1 + 3G_0(0) + G_0(y_0) + 9y_1G_0(y_1)$, for all $y_0, y_1 \in \mathbb{N}$ and G_0.

Finding witnesses for such inequalities is tedious, and we would like to automate this task. For that reason, we developed a tactic (`solve_poly`) that automatically solves the inequalities coming from Theorem 4.7. Essentially, this tactic tries to mimic how one would solve those goals in a pen-and-paper proof, and the same method is used by Wanda.

▶ **Example 4.9.** We show how to solve the constraint arising from `map_cons` mentioned in Example 4.8. The first step is to find matching terms on both sides of the inequality and subtract them. In our example, $3 + y_0 + 12y_1 + G_0(0)$ occurs on both sides, and after subtraction, we obtain the following constraint:

$$9 + 3y_0 + 9y_1 + (3y_0 + 9y_1 + 9)G_0(3 + y_0 + 3y_1) > 2G_0(0) + G_0(y_0) + 9y_1G_0(y_1).$$

The second step is combining the arguments for the higher-order variable G_0 use its monotonicity. Note that each of 0, y_0, and y_1 is lesser than or equal to $3 + y_0 + 3y_1$, because they are natural numbers. Since G_0 is weakly monotonic, we get

$$2G_0(0) + G_0(y_0) + 9y_1G_0(y_1) \leq (9y_1 + 3)G_0(3 + y_0 + 3y_1).$$

Now we can simplify our original constraint to

$$9 + 3y_0 + 9y_1 + (3y_0 + 9y_1 + 9)G_0(3 + y_0 + 3y_1) > (9y_1 + 3)G_0(3 + y_0 + 3y_1).$$

Since $3y_0 + 9y_1 + 9 \geq 9y_1 + 3$, we have

$$(3y_0 + 9y_1 + 9)G_0(3 + y_0 + 3y_1) \geq (9y_1 + 3)G_0(3 + y_0 + 3y_1).$$

This is sufficient to conclude that the constraints for `map_cons` are satisfied.

The tactic `solve_poly` (`solve_poly`) follows the steps described above. Note that we use the tactic `nia`, which is a tactic in Coq that can solve inequalities and equations in nonlinear integer arithmetic. More specifically, `solve_poly` works as follows:

- First, we generate a goal for every rewrite rule, and we destruct the assumptions so that each variable in the context is either a natural number or a function.
- For every variable f that has a function type, we look for pair (x, y) such that $f(x)$ on the left hand side and $f(y)$ occurs on the right-hand side. We try using `nia` whether we can prove $x < y$ from our assumptions. If so, we add $x < y$ to the assumptions, and otherwise, we continue.
- The resulting goals with the extra assumptions are solved using `nia`.

Note that `solve_poly` is not complete, because `nia` is incomplete. As such, if a proof using this tactic is accepted by Coq, then that proof is correct. However, if the proof is not accepted, then it does not have to be the case that the proof is false. With the material discussed in this section, we can write down the polynomials given in Example 4.2, and the tactic is able to verify strong normalization.

5 Generating Proof Scripts

In this section, we discuss the practical aspects of our verification framework. In principle one can manually encode rewrite systems as Coq files and use the formalization we provide to verify their own termination proofs. However, this is cumbersome to do. Indeed, in Example 2.7 we used abbreviations to make the formal description of $\mathcal{R}_{\mathsf{map}}$ more readable. A rewrite system with many more rules would be difficult to encode manually. Additionally, to formally establish termination we also need to encode proofs. We did this in Example 4.2. The full formal encoding of $\mathcal{R}_{\mathsf{map}}$ and its termination proof is found in the file `Map.v`.

5.1 Proof traces for polynomial interpretation

This difficulty of manual encoding motivates the usage of proof traces. A proof trace is a human-friendly encoding of a TRS and the essential information needed to reconstruct the termination proof as a Coq script. Let us again consider $\mathcal{R}_{\mathsf{map}}$ as an example. The proof trace for this system starts with `YES` to signal that we have a termination proof for it. Then we have a list encoding the signature and the rules of the system.

```
YES
Signature: [
  cons : a -> list -> list ;
  map : list -> (a -> a) -> list ;
  nil : list
]
Rules: [
  map nil F => nil ;
  map (cons X Y) G => cons (G X) (map Y G)
]
```

Notice that the free variables in the rules do not need to be declared nor their typing information provided. Coq can infer this information automatically. The last section of the proof trace describes the interpretation of each function symbol in the signature.

```
Interpretation: [
  J(cons) = Lam[y0;y1].3 + 2*y1;
  J(map)  = Lam[y0;G1].3*y0 + 3*y0 * G1(y0);
  J(nil)  = 3
]
```

We can fully reconstruct a formal proof of termination for $\mathcal{R}_{\mathsf{map}}$, which uses Theorem 4.7, with the information provided in the proof trace above. The full description of proof traces can be found in [29], the API for ONijn. Proof traces are not Coq files. So we need to further compile them into a proper Coq script. The schematics in Figure 1 describe the steps necessary for it. We use ONijn to compile proof traces to Coq script. It is invoked as follows:

```
onijn path/to/proof/trace.onijn -o path/to/proof/script.v
```

Here, the first argument is the file path to a proof trace file and the `-o` option requires the file path to the resulting Coq script. The resulting Coq script can be verified by Nijn as follows:

```
coqc path/to/proof/script.v
```

Instructions on how to locally install ONijn/Nijn can be found at [29].

5.2 Verifying Wanda's Polynomial Interpretations

It is worth noticing that the termination prover is abstract in our certification framework. This means that we are not bound to a specific termination tool. So we can verify any termination tool that implements the interpretation method described here and can output proof traces in ONijn format.

Since Wanda [16] is a termination tool that implements the interpretation method in [8], it is our first candidate for verification. We added to Wanda the runtime argument `--formal` so it can output proof traces in ONijn format. In [16] one can find details on how to invoke Wanda. For instance, we illustrate below how to run Wanda on the map AFS.

```
./wanda.exe -d rem --formal Mixed_HO_10_map.afs
```

The setting `-d rem` sets Wanda to disable rule removal. The option `--formal` sets Wanda to only use polynomial interpretations and output proofs to ONijn proof traces. Running Wanda with these options gives us the proof trace we used for $\mathcal{R}_{\mathsf{map}}$ above. The latest version of Wanda, which includes this parameter, is found at [17].

The table below describes our experimental evaluation on verifying Wanda's output with the settings above. The benchmark set consists of those 46 TRSs that Wanda outputs `YES` while using only polynomial interpretations and no rule removal. The time limit for certification of each system is set to 60 seconds.

The experiment was run in a machine with M1 Pro 2021 processor with 16GB of RAM. Memory usage of Nijn during certification ranges from 400MB to 750MB. We provide the experimental benchmarks at https://github.com/deividrvale/nijn-coq-script-generation.

Table 1 Experimental Results.

	Wanda			Nijn/ONijn		
Technique	# YES	Pct.	Avg. Time	# Certified	Perc.	Avg. Time
Poly, no rule removal	46	23%	0.07s	46	100%	4.06s

Hence, we can certify all TRSs proven SN by Wanda using only polynomial interpretations.

6 Conclusions and Future Work

We presented a formalization of the polynomial method in higher-order rewriting. This not only included the basic notions, such as algebraic functional systems, but also the interpretation method and the instantiation of this method to polynomials. In addition, we showed how to generate Coq scripts from the output of termination provers. This allowed us to certify their output and construct a formal proof of strong normalization. We also applied our tools to a concrete instance, namely to check the output of Wanda.

There are numerous ways to extend this work. First, one could formalize more techniques from higher-order rewriting, such as tuple interpretations [18] and dependency pairs [19, 22]. One could also integrate HORPO into our framework [20]. Second, in the current formalization, the interpretation of application is fixed for every instance of the polynomial method. One could also provide the user with the option to select their own interpretation. Third, currently, only Wanda is integrated with our work. This could be extended so that there is direct integration for other tools as well.

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 Ariane Alves Almeida and Mauricio Ayala-Rincón. Formalizing the dependency pair criterion for innermost termination. *Sci. Comput. Program.*, 195:102474, 2020. doi:10.1016/j.scico.2020.102474.

3 Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.*, 236(1-2):133–178, 2000. doi:10.1016/S0304-3975(99)00207-8.

4 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.

5 Frédéric Blanqui, Jean-Pierre Jouannaud, and Albert Rubio. The computability path ordering. *Log. Methods Comput. Sci.*, 11(4), 2015. doi:10.2168/LMCS-11(4:3)2015.

6 Frédéric Blanqui and Adam Koprowski. CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Math. Struct. Comput. Sci.*, 21(4):827–859, 2011. doi:10.1017/S0960129511000120.

7 Evelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, and Xavier Urbain. Automated certified proofs with cime3. In Manfred Schmidt-Schauß, editor, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, May 30 - June 1, 2011, Novi Sad, Serbia*, volume 10 of *LIPIcs*, pages 21–30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011. doi:10.4230/LIPIcs.RTA.2011.21.

8 Carsten Fuhs and Cynthia Kop. Polynomial Interpretations for Higher-Order Rewriting. In Ashish Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA'12) , RTA 2012, May 28 - June 2, 2012, Nagoya, Japan*, volume 15 of *LIPIcs*, pages 176–192. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPIcs.RTA.2012.176.

9 Carsten Fuhs and Cynthia Kop. A Static Higher-Order Dependency Pair Framework. 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 752–782. Springer, 2019. doi:10.1007/978-3-030-17184-1_27.

10 Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Analyzing program termination and complexity automatically with aprove. *J. Autom. Reason.*, 58(1):3–31, 2017. doi:10.1007/s10817-016-9388-y.

11 Jürgen Giesl, Albert Rubio, Christian Sternagel, Johannes Waldmann, and Akihisa Yamada. The termination and complexity competition. In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 156–166, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-17502-3_10.

12 Raúl Gutiérrez and Salvador Lucas. mu-term: Verify termination properties automatically (system description). 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 436–447. Springer, 2020. doi:10.1007/978-3-030-51054-1_28.

13 Makoto Hamana. Theory and practice of second-order rewriting: Foundation, evolution, and SOL. In Keisuke Nakano and Konstantinos Sagonas, editors, *Functional and Logic Programming - 15th International Symposium, FLOPS 2020, Akita, Japan, September 14-16,*

2020, Proceedings, volume 12073 of *Lecture Notes in Computer Science*, pages 3–9. Springer, 2020. doi:10.1007/978-3-030-59025-3_1.

14 Martin Hofmann and Thomas Streicher. The groupoid model refutes uniqueness of identity proofs. In *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994*, pages 208–212. IEEE Computer Society, 1994. doi:10.1109/LICS.1994.316071.

15 Cynthia Kop. *Higher Order Termination: Automatable Techniques for Proving Termination of Higher-Order Term Rewriting Systems*. PhD thesis, Vrije Universiteit Amsterdam, 2012.

16 Cynthia Kop. WANDA - a higher order termination tool (system description). In Zena M. Ariola, editor, *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 36:1–36:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.FSCD.2020.36.

17 Cynthia Kop. Wanda's source code repository, 2023. URL: https://github.com/hezzel/wanda.

18 Cynthia Kop and Deivid Vale. Tuple Interpretations for Higher-Order Complexity. In Naoki Kobayashi, editor, *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference)*, volume 195 of *LIPIcs*, pages 31:1–31:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.FSCD.2021.31.

19 Cynthia Kop and Femke van Raamsdonk. Dynamic Dependency Pairs for Algebraic Functional Systems. *Log. Methods Comput. Sci.*, 8(2), 2012. doi:10.2168/LMCS-8(2:10)2012.

20 Adam Koprowski. Coq formalization of the higher-order recursive path ordering. *Appl. Algebra Eng. Commun. Comput.*, 20(5-6):379–425, 2009. doi:10.1007/s00200-009-0105-5.

21 Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp. Tyrolean Termination Tool 2. In Ralf Treinen, editor, *Rewriting Techniques and Applications, 20th International Conference, RTA 2009, Brasília, Brazil, June 29 - July 1, 2009, Proceedings*, volume 5595 of *Lecture Notes in Computer Science*, pages 295–304. Springer, 2009. doi:10.1007/978-3-642-02348-4_21.

22 Keiichirou Kusakari, Yasuo Isogai, Masahiko Sakai, and Frédéric Blanqui. Static dependency pair method based on strong computability for higher-order rewrite systems. *IEICE Trans. Inf. Syst.*, 92-D(10):2007–2015, 2009. doi:10.1587/transinf.E92.D.2007.

23 Friedrich Neurauter and Aart Middeldorp. Revisiting matrix interpretations for proving termination of term rewriting. In Manfred Schmidt-Schauß, editor, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, May 30 - June 1, 2011, Novi Sad, Serbia*, volume 10 of *LIPIcs*, pages 251–266. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011. doi:10.4230/LIPIcs.RTA.2011.251.

24 J.C. van de Pol. *Termination of Higher-order Rewrite Systems*. PhD thesis, University of Utrecht, 1996. URL: https://www.cs.au.dk/~jaco/papers/thesis.pdf.

25 José-Luis Ruiz-Reina, José-Antonio Alonso, María-José Hidalgo, and Francisco-Jesús Martín-Mateos. Formalizing Rewriting in the ACL2 Theorem Prover. In John A. Campbell and Eugenio Roanes-Lozano, editors, *Artificial Intelligence and Symbolic Computation, International Conference AISC 2000 Madrid, Spain, July 17-19, 2000, Revised Papers*, volume 1930 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2000. doi:10.1007/3-540-44990-6_7.

26 Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Math. Struct. Comput. Sci.*, 21(4):795–825, 2011. doi:10.1017/S0960129511000119.

27 William W. Tait. Intensional Interpretations of Functionals of Finite Type I. *J. Symb. Log.*, 32(2):198–212, 1967. doi:10.2307/2271658.

28 René Thiemann and Christian Sternagel. Certification of Termination Proofs Using CeTA. 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 452–468. Springer, 2009. doi:10.1007/978-3-642-03359-9_31.

29 Deivid Vale and Niels van der Weide. Onijn documentation, 2022. URL: https://deividrvale.github.io/nijn-coq-script-generation/onijn/index.html.

30 Deivid Vale and Niels van der Weide. deividrvale/nijn-coq-script-generation: First Release of public API, May 2023. doi:10.5281/zenodo.7915736.

31 Niels van der Weide and Deivid Vale. nmvdw/nijn: 1.0.0, May 2023. doi:10.5281/zenodo.7913023.

32 Niels van der Weide, Deivid Vale, and Cynthia Kop. Certifying higher-order polynomial interpretations. *CoRR*, abs/2302.11892, 2023. doi:10.48550/arXiv.2302.11892.

33 Johannes Waldmann. Matchbox: A tool for match-bounded string rewriting. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings*, volume 3091 of *Lecture Notes in Computer Science*, pages 85–94. Springer, 2004. doi:10.1007/978-3-540-25979-4_6.

34 Akihisa Yamada. Multi-dimensional interpretations for termination of term rewriting. 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 273–290. Springer, 2021. doi:10.1007/978-3-030-79876-5_16.

35 Akihisa Yamada, Keiichirou Kusakari, and Toshiki Sakabe. Nagoya Termination Tool. In Gilles Dowek, editor, *Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8560 of *Lecture Notes in Computer Science*, pages 466–475. Springer, 2014. doi:10.1007/978-3-319-08918-8_32.

Slice Nondeterminism

Niels F. W. Voorneveld
Tallinn University of Technology, Estonia

Abstract

This paper studies a technique for describing and formalising nondeterministic functions, using slice categories. Results of a nondeterministic function are modelled by an object of the slice category over the codomain of the function, which is an indexed family over the codomain. Two such families denote the same set of results if slice morphisms exist between them in both directions. We formulate the category of nondeterministic functions by expressing a set of possible results as an equivalence class of objects. If we allow families to use any indexing set, this category will be equivalent to the category of relations. When we limit ourselves to a smaller universe of indexing sets, we get a subcategory which more closely resembles nondeterministic programs. We compare this category with other representations of the category of relations, and see how many properties can be carried over, such as its product, coproduct and other monoidal structures. We can describe inductive nondeterministic structures by lifting free monads from the category of sets. Moreover, due to the intensional nature of the slice representation, nondeterministic processes are easily represented, such as interleaving concurrency and labelled transition systems. This paper has been formalised in Agda.

2012 ACM Subject Classification Theory of computation → Categorical semantics; Theory of computation → Type theory

Keywords and phrases Category theory, Agda, Slice category, Nondeterministic functions, Powerset

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.31

Supplementary Material *Software*: `https://github.com/Voorn/Slice-Nondeterminism` archived at `swh:1:dir:540737f2bdc27c9a2bad796cd6713d39de325e94`

Funding *Niels F. W. Voorneveld*: Supported by the ESF funded Estonian IT Academy research measure (project 2014-2020.4.05.19-0001)

1 Introduction

Nondeterminism slips into many facets of computation, be it a run-time optimiser making decisions based on unknown facts, a machine learning algorithm which outputs preferences from beyond a black box, or dependencies of a program on an ever-changing and chaotic environment. If you want to verify correctness of programs, you may not be able to precisely model every aspect, and will have to embrace the fact that to an extent, a program may behave unpredictably.

In essence, nondeterminism is the potential for a program to produce different results in different runs, even in situations when all known external conditions in both runs appear to be the exact same. These multiple possible results are often gathered in a set, a subset of all plausible results allowed by the program's type. This can be seen as an element of the powerset over plausible results.

$$\{a \in A \mid P(a)\} \qquad \text{with } P \text{ a predicate on } A \tag{1}$$

A subset of a set A is commonly described using a predicate on the set. Given a predicate P on A, we can define the subset of all elements of A satisfying P. This is a fundamental construction in set theory. However, it is not directly suited as a tool for gathering possible results of a program from a constructive perspective. Given a program it is often difficult to find a computable way to test for the fact that a certain result is produced. On the other


© Niels F. W. Voorneveld;
licensed under Creative Commons License CC-BY 4.0

hand, given a predicate, it is difficult to find a program which nondeterministically produces the exact collection of results satisfying the predicate. The two concepts seem misaligned computationally.

Instead, a program tends to behave nondeterministically based on some unknown cause, unspecified state, or unpredictable decision procedure. It makes sense to describe this unknown with some indexing set I which captures a set of circumstances, for instance potential states, decisions, or transitions, and a function $f : I \to A$ which associates to each circumstance the result a program will produce.

$$\{f(i) \mid i \in I\} \qquad \text{with } f \text{ a function } I \to A \tag{2}$$

In other words, we define a *family* over A, which is an object in the *slice category* over A. These families should be endowed with a notion of equivalence, as different objects in the slice category could represent the same subset of A. The natural solution here lies in the morphisms of the slice category over A; an object is a subset of another object if there is a morphism between them, and they give the same subset if there are morphisms both ways. The subset property can be shown by *any* morphism, not just the injective ones, since we do not want to differentiate between families with different multiplicities of elements.

A nondeterministic function $f : A \to B$ is given by a map which associates to each argument an indexing set and a function from the indexing set to the set of results. This representation lands it firmly between two other representations of relations: *extensional relations* and *spans*.

$$A \to B \to \mathsf{Set} \qquad A \to \Sigma_{I:\mathsf{Set}}\,(I \to B) \qquad \Sigma_{I:\mathsf{Set}}\,(I \to A \times B) \tag{3}$$

In case of $A \to B \to \mathsf{Set}$, we see Set as a space of propositions, in which we will only care about whether the set is inhabited or not. In the other two statements, we use Set as a universe of indexing sets.

Each instance forms a *bicategory* [7], where a 2-morphism from one to another tells us that the relation modelled by one is a subrelation of the relation modelled by the other. We turn this into a *setoid-enriched category* [12], sometimes called a *$\mathcal{E}$-category* [3, 13], which is a category whose morphisms are endowed with an equivalence relation, and whose categorical equations hold up to that equivalence relation. In each instance, the equivalence on morphisms is given by the symmetrisation of the relation induced by the 2-morphisms; they are equivalent if there is a 2-morphism in both directions. E.g., two relations modelled by $R, S : A \to B \to \mathsf{Set}$ are equivalent if for each $a \in A$, and $b \in B$, $R\,a\,b$ is inhabited if and only if $S\,a\,b$ is inhabited. Similarly, two spans are deemed equivalent[1] if they point out the same set of pairs of A and B. Taking the quotient over the equivalence relation of morphisms in an $\mathcal{E}$-category, we get a category. Each of the three categories gives an equivalent representation of the category of relations Rel.

In this paper, we focus on the approach of the second category. Instead of using Set, we specify a smaller inductive universe of indexing sets $\mathbb{U}$, which is sufficient for modelling many nondeterministic processes. The collection of families over A will form a set if we restrict indexing sets to this universe $\mathbb{U}$, and as such we get an endofunctor on Set. Relative to the aforementioned equivalence relation on families, the endofunctor forms a monad, which acts like the powerset monad. The Kleisli category over this monad is what we call the category of *slice nondeterministic functions* $\mathsf{SNF}_{\mathbb{U}}$, and can be put in the form of $\mathcal{E}$-category.

[1] Not equivalent in the category of spans, but in the sense that they model the same relation.

The category $\mathsf{SNF}_{\mathbb{U}}$ forms a wide subcategory of Rel, since it is a substructure of the second representation of relations given in (3). This category still retains many of the properties of Rel, while deviating from it in ways more in-line with nondeterministic processes. In this paper, we shall explore the properties of $\mathsf{SNF}_{\mathbb{U}}$, see how it models a variety of nondeterministic processes, and we shall briefly compare this approach with the other two approaches from (3). In particular, the category $\mathsf{SNF}_{\mathbb{U}}$ is better at dealing with composition and iteration.

In programming theory, a prevalent way of representing nondeterministic computations is in the form of *algebraic effect* [19]. There we consider nondeterministic choices as operations in the programming language. These choices can be enumerated in the indexing set of the family, encoding how the choices should be resolved. Each index gives us a way of handling the algebraic operations in the sense of *effect handlers* [20].

The resulting denotation using slices is similar to historical models of nondeterminism in *domain theory* [9, 2]. Domains are sets with an additional preorder which captures a notion of computational approximation, and *powerdomains* [18] consider different nondeterministic extensions to the domain. These form more usual models for nondeterministic processes, as for instance done in *guarded powerdomains* [17]. Slice nondeterminism offers an alternative light-weight formalisation approach, allowing us to model a lot of nondeterministic processes.

We use category theory in this work, in particular focussing on emulating properties of the category of relations in our framework (see e.g. [16]). The contributions of this paper can be split into three parts; the presentation of a novel model for nondeterministic functions in terms of slices, an exploration of the properties of the associated category, and examples of concepts and processes which it can capture with relative ease.

This paper is formalised in Agda: `https://github.com/Voorn/Slice-Nondeterminism`. See in particular the `ITP-paper.agda` file for links to terms corresponding to the definitions and results of this paper. Most of the code uses only the Agda standard library, though there are some optional files linking the results to the Agda categories library.

2 Powerset via the Slice Category

The main idea is to use a family over A to represent a subset of A. Such a family consists of an indexing set I together with a function $h : I \to A$, and is used to represent the subset $\{a \in A \mid \exists i \in I.\, h(i) = a\}$. It is useful to put a limit on what kind of set I can be. To this end, we specify a universe of sets $\mathbb{U}$.

2.1 A Universe for Indexing Sets

We could simply take $\mathbb{U}$ to be the universe of sets itself. There are some drawbacks to this. The first being that the collection of families using any indexing set does not form a set itself, and hence cannot be used as a basis for an endofunctor in the category of sets. Secondly, the more indexing sets we allow the more cases we need to check when formalising general properties about them. Thirdly, nondeterministic programs do not tend to have the abilities that models using arbitrary sets have. For instance, a program cannot in general nondeterministically produce any result from its codomain, nor can it decide to produce results based on incomputable tests (like equality checking) on its argument.

Instead, we construct an inductive universe of sets which is a set itself, and is closed with respect to a few chosen constructions we would like to use.

▶ **Definition 1.** *The universe of sets $\mathbb{U}$ are defined inductively as:*

$$\frac{}{\bot : \mathbb{U}} \quad \frac{}{\top : \mathbb{U}} \quad \frac{}{\mathbb{N} : \mathbb{U}} \quad \frac{A, B : \mathbb{U}}{A \to B : \mathbb{U}} \quad \frac{A, B : \mathbb{U}}{A \uplus B : \mathbb{U}} \quad \frac{A, B : \mathbb{U}}{A \times B : \mathbb{U}} \quad \frac{A : \mathbb{U} \qquad f : A \to \mathbb{U}}{\Sigma_{a:A}\, f(a) : \mathbb{U}} \quad \frac{A : \mathbb{U} \qquad f : A \to \mathbb{U}}{\Pi_{a:A}\, f(a) : \mathbb{U}}$$

Here $\bot$ is the empty set, $\top$ the single element set, and $\mathbb{N}$ the natural numbers. $A \to B$ is the function space, as specified by the ambient logic (e.g. Agda functions), $A \uplus B$ is disjoint union, and $A \times B$ the Cartesian product on sets. Lastly, $\Sigma_{a:A} f(a)$ is a sigma type whose elements consists of pairs $(a : A, b : f(a))$, and $\Pi_{a:A} f$ a pi type whose elements are dependent functions $(a : A) \to f(a)$. It should be noted that a large portion of the development up to Section 5 works without function spaces. Hence the universe could be limited further, effectively only using *enumerable sets*. For elements of product sets and sigma sets, we write proj_1 and proj_2 for the projections into the first and second components respectively, and for elements of sum sets, we write inj_1 and inj_2 for the first and second injections respectively.

Note that with the inclusion of Sigma types and Pi types in $\mathbb{U}$, the property of whether a set of $\mathbb{U}$ is inhabited is not decidable, since Sigma and Pi types function as existential and universal quantifiers. So in the representation of nondeterministic function, it may not be decidable whether the function outputs a result.

In formalisation, $\mathbb{U}$ is simply a collection of names which each is given a denotation according to the above definitions. For simplicity, we shall write results in this paper directly in terms of the indexing sets, using the closure properties of $\mathbb{U}$ specified above.

▶ **Definition 2.** *The set of $\mathbb{U}$-families over A, denoted $\mathsf{SL}_{\mathbb{U}}\, A$ is given by: $\Sigma_{I:\mathbb{U}} (I \to A)$.*

Hence, a family $S = (I, m) \in \mathsf{SL}_{\mathbb{U}}\, A$ is given by an indexing set $I : \mathbb{U}$ together with a function $m : I \to A$. In other words, it specifies a domain which we may use for indexing elements of A, and a function which associates to each index a result. For a family S, we denote S_I and S_m for the associated indexing set and map respectively.

In the conclusion we will also consider other universes of indexing sets C, and consider families SL_C using those. However, the main development focusses on using $\mathbb{U}$. With this universe, we know that $\mathsf{SL}_{\mathbb{U}}\, A$ is in fact also a set. Hence, the construction is entirely contained within the universe of sets, and $\mathsf{SL}_{\mathbb{U}}$ can be made into an endofunctor on Set. As a functor, it sends a function $f : A \to B$ to a function on families $\mathsf{SL}_{\mathbb{U}}(f) : \mathsf{SL}_{\mathbb{U}}\, A \to \mathsf{SL}_{\mathbb{U}}\, B$ sending (S_I, S_m) to $(S_I, f \circ S_m)$. In the next subsection, we shall define an equivalence relation on $\mathsf{SL}_{\mathbb{U}}\, A$ telling us whether two families represent the same set.

2.2 Relations on Families

Though we work towards an alternative construction of relations for describing nondeterministic functions, we still require the use of relations as they are more traditionally represented for reasoning purposes. An *endorelation* $\mathcal{R}$ on X is a predicate $\mathcal{R} : X \to X \to \mathbb{P}$, where $\mathbb{P}$ is a suitable space of propositions, e.g. Set as used in Agda. We write $a\,\mathcal{R}\,b$ to say $\mathcal{R}$ relates a and b. For two relations $\mathcal{R}$ and $\mathcal{S}$ on X, we write $\mathcal{R} \subset \mathcal{S}$ if $a\,\mathcal{R}\,b$ implies $a\,\mathcal{S}\,b$. Given an endorelation $\mathcal{R}$ on X, we write $\overline{\mathcal{R}}$ for the symmetrisation of $\mathcal{R}$, which relates a and b if both $a\,\mathcal{R}\,b$ and $b\,\mathcal{R}\,a$.

We use the theory of *relators* [15, 21] to guide us in building relations on families. We limit ourselves to relators on endorelations, which is a common adaptation.

▶ **Definition 3.** *Given a relation $\mathcal{R}$ on X, we define a relation $\Gamma_{\mathbb{U}}(\mathcal{R})$ on $\mathsf{SL}_{\mathbb{U}}\, X$ by relating (I, a) and (J, b) if there is a function $h : I \to J$ such that $\forall i \in I.\ a(i)\ \mathcal{R}\ b(h(i))$.*

This construction has the following properties:

- If $\mathcal{R}$ is reflexive, then $\Gamma_{\mathbb{U}}(\mathcal{R})$ is reflexive.
- If $\mathcal{R}$ is transitive, then $\Gamma_{\mathbb{U}}(\mathcal{R})$ is transitive.
- If $\mathcal{R} \subset \mathcal{S}$, then $\Gamma_{\mathbb{U}}(\mathcal{R}) \subset \Gamma_{\mathbb{U}}(\mathcal{S})$.
- If $\forall x, y.(x\,\mathcal{R}\,y) \implies (f(x)\,\mathcal{S}\,g(y))$, then $a\ \ \Gamma_{\mathbb{U}}(\mathcal{R})\ b \implies \mathsf{SL}_{\mathbb{U}}(f)(a)\ \ \Gamma_{\mathbb{U}}(\mathcal{S})\ \ \mathsf{SL}_{\mathbb{U}}(g)(b)$.

As a consequence, if $\mathcal{R}$ is an equivalence relation, then $\overline{\Gamma_{\mathbb{U}}(\mathcal{R})}$ is an equivalence relation. Our notion of equality on families $\mathsf{SL}_{\mathbb{U}}\, X$ shall be given by taking the appropriate notion of equality $=$ on X and lifting it to an equivalence relation on $\mathsf{SL}_{\mathbb{U}}\, X$ given by $\overline{\Gamma_{\mathbb{U}}(=)}$. In formalisation, the $=$ will be instantiated by propositional equality $\equiv$. We will first observe some more general properties.

2.3 A Monad on Setoids

We can see $\mathsf{SL}_{\mathbb{U}}$ as an endofunctor on setoids, as well as an endofunctor on sets. A setoid is a pair $(X, \mathcal{R})$ consisting of a set X and an equivalence relation $\mathcal{R}$ on X. A morphism between setoids $(X, \mathcal{R})$ and $(Y, \mathcal{S})$ is a relation preserving function between X and Y. We have the slice function F on setoids sending $(X, \mathcal{R})$ to $(\mathsf{SL}_{\mathbb{U}}\, X, \overline{\Gamma_{\mathbb{U}}(\mathcal{R})})$.

Given a set X, we define a set function $\langle - \rangle_X : X \to \mathsf{SL}_{\mathbb{U}}\, X$ as given by sending x to $(\top, \lambda * . x)$. This forms a natural transformation in Set; given $f : X \to Y$, then $\langle - \rangle_Y \circ f = \lambda x.(\top, \lambda * . f(x)) = \mathsf{SL}_{\mathbb{U}}(f) \circ \langle - \rangle_X$. Moreover, given a relation $\mathcal{R}$ on X, then $a \,\mathcal{R}\, b$ implies $\langle a \rangle \,\Gamma_{\mathbb{U}}\, \langle b \rangle$, hence this forms a natural transformation on setoids as well.

We have a set function $\bigsqcup_X : \mathsf{SL}_{\mathbb{U}}(\mathsf{SL}_{\mathbb{U}}(X)) \to \mathsf{SL}_{\mathbb{U}}(X)$ which does the following. Given a family $(I, a) \in \mathsf{SL}_{\mathbb{U}}(\mathsf{SL}_{\mathbb{U}}(X))$, then for each $i \in I$ we write $(J_i, b_i) = a(i) \in \mathsf{SL}_{\mathbb{U}}(X)$. $\bigsqcup_X$ sends (I, a) to (K, c) where: $K = \Sigma_{i:I}\, J_i$, meaning a pair (i, j) such that $i \in I$ and $j \in J_i$, and $c(i, j) = b_i(j)$. Similar to $\langle - \rangle$, this forms a natural transformation in both set and setoid:

- Given $f : X \to Y$, $\bigsqcup_Y \circ \mathsf{SL}_{\mathbb{U}}(\mathsf{SL}_{\mathbb{U}}(f)) = \mathsf{SL}_{\mathbb{U}}(f) \circ \bigsqcup_X$.
- Given a relation $\mathcal{R}$ on X, $u \,\Gamma_{\mathbb{U}}(\Gamma_{\mathbb{U}}(\mathcal{R}))\, v$ implies $\bigsqcup_X(u) \,\Gamma_{\mathbb{U}}(\mathcal{R})\, \bigsqcup_Y(v)$.

Importantly, we get the following result.

▶ **Proposition 4.** *$(\mathsf{SL}_{\mathbb{U}}, \langle - \rangle, \bigsqcup)$ forms a monad in the category of setoids.*

Together with the results of Subsection 2.5, we see that the monad actually satisfies the axioms of a powertheory on setoids [18], and as such is a candidate for *powersetoid*.

It is difficult to formalise things in the category of setoids, since one needs to prove many coherences and use higher-order rewrite techniques. As such, the previous proposition is as far as we go in this direction. We can however use the result to make similar claims about the category of sets in the next subsection.

2.4 A Kleisli Triple on Sets

As discussed before, we can define an equivalence relation $\equiv^{\mathbb{U}}_X$ on $\mathsf{SL}_{\mathbb{U}}(X)$ as $\overline{\Gamma_{\mathbb{U}}(=)}$. Consider the endofunctor $\mathsf{SL}^{\equiv}_{\mathbb{U}}$ which sends a set X to the quotient $\mathsf{SL}_{\mathbb{U}}(X)/\equiv^{\mathbb{U}}_X$.

Proposition 4 implies that $\mathsf{SL}_{\mathbb{U}}(-)/\equiv^{\mathbb{U}}$ forms a monad in Set. The associated multiplication operation for X is defined on domain $\mathsf{SL}_{\mathbb{U}}(\mathsf{SL}_{\mathbb{U}}(X)/\equiv^{\mathbb{U}})/\equiv^{\mathbb{U}}$. A morphism whose domain is given by a quotient is defined as a morphism invariant under the equivalence relation of the quotient. In formalisation, this means the morphism would need to be equipped with a proof that they are well defined. In the spirit of getting flexibility in defining nondeterministic function, we bypass this necessity by avoiding the use of quotients in the domain of morphisms. Instead, we look at the Kleisli triple corresponding to the monad structure, which is as follows:

- The unit $\langle - \rangle_X : X \to \mathsf{SL}_{\mathbb{U}}(X)$ is as given before.
- The Kleisli lifting $(-)^* : (X \to \mathsf{SL}_{\mathbb{U}}(Y)) \to (\mathsf{SL}_{\mathbb{U}}(X) \to \mathsf{SL}_{\mathbb{U}}(Y))$ sends f to $\bigsqcup_Y \circ \mathsf{SL}_{\mathbb{U}}(f)$.

We see that this satisfies the appropriate properties:

- For $f, f' : X \to \mathsf{SL}_{\mathbb{U}}(Y)$, such that $\forall x.\, f(x) \equiv^{\mathbb{U}}_X f'(x)$, then $\forall a, a' \in \mathsf{SL}_{\mathbb{U}}(X).\, a \equiv^{\mathbb{U}}_X a' \implies f^*(a) \equiv^{\mathbb{U}}_Y f'^*(a')$.

- For all $a \in \mathsf{SL}_{\mathbb{U}}(X)$, $a \equiv^{\mathbb{U}}_X \langle - \rangle^*(a)$.
- For $f : X \to \mathsf{SL}_{\mathbb{U}}(Y)$ and $x \in X$, $f^*(\langle x \rangle) \equiv^{\mathbb{U}}_Y f(x)$.
- For $f : X \to \mathsf{SL}_{\mathbb{U}}(Y)$, $g : Y \to \mathsf{SL}_{\mathbb{U}}(Z)$ and $a \in \mathsf{SL}_{\mathbb{U}}(X)$, $g^*(f^*(a)) \equiv^{\mathbb{U}}_Z (g^* \circ f)^*(a)$.

2.5 Semilattice Structure

Before we continue to defining nondeterministic functions, we look at one last useful structure. We have an order $\sqsubseteq^{\mathbb{U}}$ on families over X given by applying $\Gamma_{\mathbb{U}}$ to the equality relation on X. This is a preorder which implements the subset relation discussed before, and whose symmetrisation gives the equivalence relation $\equiv^{\mathbb{U}}$ on families telling us that the two families model the same subset. Specifically, $(I, u) \sqsubseteq^{\mathbb{U}} (J, v)$ if for any $i \in I$ there is a $j \in J$ such that $u(i) = v(j)$. In other words, the images are subsets, $u(I) \subseteq v(J)$.

Alternatively, we may define a relation $\in^{\mathbb{U}}$ between X and $\mathsf{SL}_{\mathbb{U}}(X)$, which models inhabitance by relating x to S if there is an $i \in S_I$ such that $S_m(i) = x$. This gives rise to an external subset endorelation $\subseteq^{\mathbb{U}}$ on $\mathsf{SL}_{\mathbb{U}}(X)$ which relates S and Z if for any $x \in X$, $x \in^{\mathbb{U}} S \implies x \in^{\mathbb{U}} Z$. It turns out that the relations $\subseteq^{\mathbb{U}}$ and $\sqsubseteq^{\mathbb{U}}$ are equivalent.

Like for powersets, $\sqsubseteq^{\mathbb{U}}$ comes equipped with a join semi-lattice structure [9]. As such, we can see it as a model of *angelic* nondeterminism, or lower powertheory [10].

▶ **Proposition 5.** *For sets A and I, where I is from $\mathbb{U}$, and $f : I \to \mathsf{SL}_{\mathbb{U}}(A)$, there is an element $\bigvee f \in \mathsf{SL}_{\mathbb{U}}(A)$ giving the supremum of $f(I)$ under the order $\sqsubseteq^{\mathbb{U}}$. In other words, $\forall i \in I.\, f(i) \sqsubseteq^{\mathbb{U}} \bigvee f$, and for $S \in \mathsf{SL}_{\mathbb{U}}(A)$, if $\forall i \in I.\, f(i) \sqsubseteq^{\mathbb{U}} S$ then $\bigvee f \sqsubseteq^{\mathbb{U}} S$.*

Proof. We define $\bigvee f$ as $(\bigvee f)_I = \Sigma_{i:I} f(i)_I$ and $(\bigvee f)_m(i, j) = f(i)_m(j)$.

Then for any $i \in I$, $f(i) \sqsubseteq^{\mathbb{U}} \bigvee f$ since for any $j \in f(i)_I$, there is an index $k = (i, j) \in (\bigvee f)_I$ such that $f(i)_m(j) = (\bigvee f)_I(k)$.

Let $(K, u) \in \mathsf{SL}_{\mathbb{U}}(A)$ such that $\forall i \in I.\, f(i) \sqsubseteq^{\mathbb{U}} (K, u)$. Then for any $(i, j) \in (\bigvee f)_I$, since $i \in I$ there is a $k \in J$ such that $f(i)_m(j) = u(k)$, hence there is a $k \in J$ such that $(\bigvee f)_m((i, j)) = f(i)_m(j) = u(k)$. So $\bigvee f \sqsubseteq^{\mathbb{U}} S$. ◀

We distinguish three special kinds of joins.

- Let $!_A = \lambda() : \emptyset \to A$ be the unique function from the empty set to A. We define the *empty join* $\oslash_A \in \mathsf{SL}_{\mathbb{U}}(A)$ as $\oslash_A = (\emptyset, !_A)$. This is the smallest element of $\mathsf{SL}_{\mathbb{U}}(A)$, and it holds that $\oslash_A \equiv_A \bigvee !_{\mathsf{SL}_{\mathbb{U}}(A)}$.
- For $U, V \in \mathsf{SL}_{\mathbb{U}}(A)$, we define the *binary join* $U \vee V$ as $(U_I \uplus V_I, \lambda\{\mathsf{inj}_1(i) \mapsto U_m(i), \mathsf{inj}_2(j) \mapsto V_m(i)\})$. This is equivalent to $\bigvee f$ where $f : \{0, 1\} \to \mathsf{SL}_{\mathbb{U}}(A)$ with $f(0) = U$ and $f(1) = V$.
- For $f : \mathbb{N} \to \mathsf{SL}_{\mathbb{U}}(A)$ we can take the *countable join* $\bigvee f$.

Lastly, note that for $f : I \to J \to \mathsf{SL}_{\mathbb{U}}(A)$, $\bigvee(\lambda i.\, \bigvee f(i)) \equiv \bigvee(\lambda(i, j).\, f(i)(j))$. Due to the limitations of our universe of indexing sets, we do not have a complementing meet semilattice structure, which would require us to check for equality of results within indexing sets.

3 Nondeterministic Functions

To model nondeterministic functions, we use the Kleisli category associated to the Kleisli triple defined in Subsection 2.4. Since this is a Kleisli triple relative to an equivalence relation, what we end up constructing is a *Setoid-enriched category*, sometimes called an *$\mathcal{E}$-category*.

3.1 $\mathcal{E}$-categories

In an $\mathcal{E}$-category, we use setoids to describe collections of morphisms. A common interpretation of this is to see the setoid relation as a 2-morphism between these morphisms, creating a *bicategory* whose spaces of 2-morphisms have at most one element. We lay out precisely the conditions of such a structure.

▶ **Definition 6.** *An* $\mathcal{E}$-category $\mathcal{C}$ *is given by:*

- *A set of objects* O*.*
- *For any two objects* $X, Y \in O$*, a set of morphisms* $M(X,Y)$ *equipped with an equivalence relation* $\equiv$*.*
- *For each* $X \in O$ *an identity morphism* $id_X \in M(X,X)$*.*
- *For* $X, Y, Z \in O$*,* $f \in M(X,Y)$*,* $g \in M(Y,Z)$ *a morphism* $f\,g \in M(X,Z)$*.*

such that:

- *For any* $X, Y \in O$*,* $f \in M(X,Y)$*,* $id_X\, f \equiv f \equiv f\, id_Y$*.*
- $\forall X, Y, Z, W \in O$*,* $f \in M(X,Y)$*,* $g \in M(Y,Z)$*,* $h \in M(Z,W)$*,* $(f\,g)\,h \equiv f\,(g\,h)$*.*
- $\forall X, Y, Z \in O$*,* $f, f' \in M(X,Y)$*,* $g, g' \in M(Y,Z)$*, if* $f \equiv f'$ *and* $g \equiv g'$ *then* $f\,g \equiv f'\,g'$*.*

The $\mathcal{E}$-category method matches the approach of the Agda categories library. There, categories are formalised using equivalence relations on morphisms. By taking the quotient over the equivalence relations on the homsets, we get a category. Hence, we may see any $\mathcal{E}$-category as a category as well, and properties on the $\mathcal{E}$-category directly translate to properties on the resulting category. For instance, a functor between $\mathcal{E}$-categories forms a functor between their corresponding categories.

The focus of this paper is the following setoid-enriched category.

▶ **Definition 7.** *The* $\mathcal{E}$*-category of* slice nondeterministic functions*, denoted* $\mathcal{E}\mathsf{SNF}_{\mathbb{U}}$*, is the category consisting of:*

- *Objects are sets.*
- *The set of morphisms between set* A *and* B*, denoted* $A \multimap B$*, are functions* $A \to \mathsf{SL}_{\mathbb{U}}(B)$*. Two morphisms* $f, g : A \multimap B$ *are equivalent, denoted as* $f \sim g$*, if* $\forall a \in A.\, f(a) \equiv_B^{\mathbb{U}} g(a)$*.*
- *For a set* A*, the identity morphism is given by* $\langle - \rangle_A : A \multimap A$*.*
- *For two morphisms* $f : A \multimap B$ *and* $g : B \multimap C$*, the composition is given by* $f; g := f\, g^*$*.*

We write $\mathsf{SNF}_{\mathbb{U}}$ for the associated quotient category to the $\mathcal{E}$-category $\mathcal{E}\mathsf{SNF}_{\mathbb{U}}$.

The appropriate properties are satisfied as observed in the previous chapter. For $f : A \multimap B$, remember we may write $f(a)_I$ for the indexing set associated to $f(a)$, and for $i \in f(a)_I$, $f(a)_m(i)$ the element of B associated to i via $f(a)$.

3.2 Functors and Variations

Let us look at some variations on models of nondeterministic functions. First we revisit the category of relations Rel, which can be cast into the form of an $\mathcal{E}$-category as well.

▶ **Definition 8.** *The* $\mathcal{E}$-category of relations*, denoted* $\mathcal{E}\mathsf{Rel}$ *is given by:*

- *Objects are sets.*
- *Morphisms between* A *and* B *are predicates* $\mathcal{R} : A \to B \to \mathbb{P}$*, with equivalence relation:* $\mathcal{R} \equiv \mathcal{S}$ *if* $\forall a \in A, b \in B.\, a\,\mathcal{R}\,b \iff a\,\mathcal{S}\,b$*.*
- *The identity relation is the equality predicate.*
- *Composition is given by:* $a\,\mathcal{R}\mathcal{S}\,c$ *if* $\exists b.\, a\,\mathcal{R}\,b \wedge b\,\mathcal{S}\,c$*.*

Note that the type of propositions $\mathbb{P}$ needs to be flexible enough to contain both equality on sets and existential quantification. In Agda, this is instantiated by Set, with the double implication $\iff$ describing the existence of a function in both directions.

There exists a functor $[\![-]\!]$ from $\mathcal{E}\mathsf{SNF}_{\mathbb{U}}$ to $\mathcal{E}\mathsf{Rel}$, which preserves objects and sends a morphism $f : A \multimap B$ to $[\![f]\!]$ with the following property:

For $a \in A$, $b \in B$, $a \,[\![f]\!]\, b$ holds if and only if there is an $i \in f(a)_I$ such that $f(a)_m(i) = b$.

There is no functor going from Rel to $\mathsf{SNF}_{\mathbb{U}}$, since our universe of indexing sets $\mathbb{U}$ is not large enough to capture the two things that are necessary to formulate such a thing; it would need Sigma types over any set, and equality types on arbitrary sets. If however our universe $\mathbb{U}$ of indexing sets was taken to be Set itself, then Rel and $\mathsf{SNF}_{\mathsf{Set}}$ are equivalent.

In Subsection 3.3 we shall do a qualitative comparison between the two approaches of formalising nondeterministic functions. There we will see that when composing nondeterministic functions, it is easier to verify that the composite gives a certain result when it is formalised in $\mathsf{SNF}_{\mathbb{U}}$ than when it is formalised in Rel or the category of spans.

We look at four properties a slice nondeterministic function $f : A \multimap B$ could satisfy.

- f is *total* if for any $a \in A$ the indexing set $f(a)_I$ is inhabited.
  This is synonymous to saying that $\forall a \in A.\, \exists b \in B.\, a \,[\![f]\!]\, b$.
- f is *deterministic* (single-valued) if for any $a \in A$ and $i, j \in f(a)_I$, $f(a)_m(i) = f(a)_m(j)$.
  This is synonymous to saying that $\forall a \in A, b, b' \in B,\, a \,[\![f]\!]\, b \wedge a \,[\![f]\!]\, b' \implies b = b'$.
- f is *surjective* if for any $b \in B$, there are $a \in A$ and $i \in f(a)_I$ such that $f(a)_m(i) = b$.
  This is synonymous to saying that $\forall b \in B.\, \exists a \in A.\, a \,[\![f]\!]\, b$.
- f is *injective* (sometimes called modest) if $\forall a, a' \in A.\, i \in f(a)_I$, and $j \in f(a')_I$, if $f(a)_m(i) = f(a')_m(j)$ then $a = a'$.
  This is synonymous to saying that $\forall a, a' \in A, b \in B,\, a \,[\![f]\!]\, b \wedge a' \,[\![f]\!]\, b \implies a = a'$.

All four properties are satisfied by the identity morphism of $\mathsf{SNF}_{\mathbb{U}}$, are preserved by composition and invariant under equivalence of morphisms. Hence, any subset of properties specifies a *wide subcategory* of $\mathsf{SNF}_{\mathbb{U}}$.

If a morphism is surjective and deterministic, it is an epimorphism. If a morphism is total and injective, it is a monomorphism. Lastly, the wide subcategory of total and deterministic morphisms is equivalent to the category Set of sets.

3.3 Method Comparison

We can compare the method of formalising nondeterministic processes using $\mathsf{SNF}_{\mathbb{U}}$ with other representations of the category of relations. We do this with code from the Agda proof assistant. Let us consider a particular example. Suppose we toss five coins, and want to prove that it is possible to get three heads. So, starting with zero, if we nondeterministically add zero heads or one head a total of five times, it should be possible to get three heads. We compare extensional relations, spans and slice nondeterminism, and we first define a nondeterministic function on $\mathbb{N}$ representing a coin toss.

```
Erel-ℕ-toss : ℕ → ℕ → Set
Erel-ℕ-toss n m = (n ≡ m) ⊎ (suc n ≡ m)

Span-ℕ-toss : Σ Set λ I → I → ℕ × ℕ
Span-ℕ-toss = (ℕ × Bool) , λ {(n , false) → n , n ;
    (n , true) → n , (suc n)}

Slic-ℕ-toss : ℕ → Σ Set λ I → (I → ℕ)
Slic-ℕ-toss n = Bool , (λ { false → n ; true → suc n})
```

Note that though we use Set in the slice example, the indexing set is from $\mathbb{U}$. In each case, we construct a multi-toss operation inductively, by composing the identity relation with a number of toss operations specified by the argument. We look at the examples, and prove that 5 tosses may get 3 heads, using a sequence of two tails and three heads.

```
Erel-ℕ-example : Erel-ℕ-test (Erel-ℕ-multi-toss 5) 0 3
Erel-ℕ-example = 0 , (inj₁ refl , 0 , (inj₁ refl) , 1 , (inj₂ refl) ,
2 , (inj₂ refl) , 3 , (inj₂ refl) , refl)

Span-ℕ-example : Span-ℕ-test (Span-ℕ-multi-toss 5) 0 3
Span-ℕ-example = (((0 , false) , ((0 , false) , ((0 , true) , ((1 , true) ,
((2 , true) , 3) , refl) , refl) , refl) , refl) , refl) , refl

Slic-ℕ-example : Slic-ℕ-test (Slic-ℕ-multi-toss 5) 0 3
Slic-ℕ-example = (false , false , true , true , true , tt) , refl
```

In the first method, since composition is defined using existential quantification on the intermediate argument, we need to specify all the intermediate results in the sequence. So, after one toss we can keep 0, given that we choose a tail signified by the inj_1. Then still 0 after another tail, followed by 1, 2 and 3, each with a heads result signified by inj_2. For spans, a similar proof is necessary, though we specify transitions using both the input state and a truth value denoting the result of the coin toss. Though in a slightly different way, both endorelations and spans need the same information in their proof.

Only in the slice nondeterministic proof do we neither need to specify intermediate states, nor have to check for equality at each step. The proof there is simply a sequence of choices with a single final verification that it gives the right result. This example illustrates the relative ease of using slice nondeterminism in these situations.

Many nondeterministic processes are results of compositions of many subprocesses. Therefore, slice nondeterministic functions are particularly well suited to capture such processes. In the rest of the paper, we will see that it moreover still retains a lot of the useful structures and properties that the category of relations exhibits.

4 Categorical Structures

We look at some of the structures that can be found in $\mathsf{SNF}_{\mathbb{U}}$. Most of these reflect similar structures from the category of relations, though not all can be replicated. In most cases though, this inability to express some structures is more faithful to the limitations of nondeterministic programs compared to relations.

Some of the structures we look at are lifted from the category of sets using the unit $\langle - \rangle$ of $\mathsf{SL}_{\mathbb{U}}$. Explicitly, we have a functor $|-| : \mathsf{Set} \to \mathsf{SNF}_{\mathbb{U}}$ which keeps objects as is, and sends morphisms $f : A \to B$ to $|f| = \lambda x.\langle f(x) \rangle_B : A \multimap B$.

4.1 Morphisms with Daggers

Given a relation $\mathcal{R}$ between A and B, there is a relation $\mathcal{R}^\dagger$ between B and A, called the *dagger* of $\mathcal{R}$, such that $a \mathcal{R} b \iff b \mathcal{R}^\dagger a$. Such a reversing of morphisms does not exist for $\mathsf{SNF}_{\mathbb{U}}$. Though unfortunate, it is arguably not an unreasonable problem to have when thinking of nondeterministic functions as programs.

For a nondeterministic program P of type $\sigma \to \tau$, there is in general not another program of type $\tau \to \sigma$ which for input $V : \tau$ can nondeterministically give any term W of σ such that $P(W)$ may produce V. This is due to two general concerns:

- Equality in τ may not be checkable by a program, hence there may not be a way of verifying whether $P(W)$ can produce V.
- Programs may not have access to enumerations over terms of σ.

Both these restrictions align closely to the limitations of our universe $\mathbb{U}$.

We can however distinguish those slice nondeterministic functions that have a dagger.

▶ **Definition 9.** *$f : A \multimap B$ and $g : B \multimap A$ are each others dagger, denoted $f \dagger g$, if $[\![f]\!]^\dagger = [\![g]\!]$.*

If a morphism has a dagger, we call it *daggerable*. This property of having a dagger is preserved under composition and the equivalence relation on morphisms. Moreover, the identity morphism is a dagger of itself. We conclude that the daggerable morphisms form a wide subcategory of $\mathsf{SNF}_{\mathbb{U}}$.

Note that being daggerable does not mean that the daggered program is an inverse. We can say the following; let f g be eachother's dagger, then:

- f is total if and only if g is surjective.
- f is deterministic if and only if g is injective.
- If f is total and injective, then $f\,g$ is the identity morphism. In other words, g is a post inverse of f.
- If f is deterministic and surjective, then $g\,f$ is the identity morphism. In other words, g is a pre-inverse of f.

There is a slight difference between the notion of daggerability and the notion of *reversibility*. A nondeterministic function is normally called reversible if it is injective (see e.g. [11]). If we were to match this definition, the appropriate category to consider for reversible nondeterministic functions is the category of daggerable injective (maybe total) morphisms.

4.2 Products and Coproducts

Similar to the category of relations, the category of slice nondeterministic functions is both Cartesian and Cocartesian, with both structures mirroring each other exactly as in a *semi-additive category*.

Firstly, the initial and terminal object of the category is given by the empty set.

▶ **Lemma 10.** *For a set A, any two morphisms $\emptyset \multimap A$ are similar, and any two morphisms $A \multimap \emptyset$ are similar.*

Consider the bifunctor for disjoint union $\uplus$ on sets, with injections $\mathsf{inj}_1 : A \to A \uplus B$ and $\mathsf{inj}_2 : B \to A \uplus B$. We can lift $\uplus$ to the category of nondeterministic functions, where for $f : A \multimap B$ and $g : C \multimap D$, $f \uplus g : A \uplus B \multimap C \uplus D$ sends $\mathsf{inj}_1(a)$ to $(f(a)_I, \lambda i.\mathsf{inj}_1(f(a)_m(i)))$ and $\mathsf{inj}_2(b)$ to $(g(b)_I, \lambda i.\mathsf{inj}_2(g(b)_m(i)))$. Consider the following natural transformations.

- $\curlyvee_1 := |\mathsf{inj}_1| : A \multimap A \uplus B$ and $\curlyvee_2 := |\mathsf{inj}_2| : B \multimap A \uplus B$.
- Given $f : A \multimap C$ and $g : B \multimap C$, we define $(f \curlyvee g) : A \uplus B \multimap C$ as given by $(f \curlyvee g)(\mathsf{inj}_1(a)) = f(a)$ and $(f \curlyvee g)(\mathsf{inj}_2(b)) = g(b)$.

The following proposition establishes that $(\uplus, \curlyvee_1, \curlyvee_2)$ forms a coproduct in $\mathsf{SNF}_{\mathbb{U}}$.

▶ **Proposition 11.** *Let $f : A \multimap C$, $g : B \multimap C$, and $h : A \uplus B \multimap C$ such that $\curlyvee_1; h \sim f$ and $\curlyvee_2; h \sim g$, then $h \sim (f \curlyvee g)$.*

Proof. Let $a \in A$, then $(f \amalg g)(\mathsf{inj}_1(a)) = f(a)$ and $h(\mathsf{inj}_1(a)) = h^*(\langle \mathsf{inj}_1(a) \rangle) = h^*(\amalg_1(a)) = (\amalg_1; h)(a)$. Since $\amalg_1; h \sim f$, $f(a) \equiv^{\mathbb{U}} h(\mathsf{inj}_1(a))$. Similarly, for $b \in B$, $(f \amalg g)(\mathsf{inj}_2(b)) = g(b) \equiv^{\mathbb{U}} (\amalg_2; h)(b) = h(\mathsf{inj}_2(b))$, so $h \sim (f \amalg g)$. ◀

We define $\mathsf{merge}_A = (id_A \amalg id_A) : A \uplus A \to A$, so it holds that $(f \uplus g); \mathsf{merge}_C = (f \amalg g)$.

Remember that for each set A, we have a special kind of family called the *empty family* $\oslash_A \in \mathsf{SL}_{\mathbb{U}}(A)$ given by: $\oslash = (\emptyset, \lambda())$.

- $\amalg_1$ has a dagger $\pi_1 : A \uplus B \multimap A$ defined as: $\pi_1(\mathsf{inj}_1(a)) = \langle a \rangle$ and $\pi_1(\mathsf{inj}_2(b)) = \oslash_A$. Dually, $\amalg_2$ has a dagger $\pi_2 : A \uplus B \multimap B$ defined as: $\pi_2(\mathsf{inj}_1(a)) = \oslash_B$ and $\pi_2(\mathsf{inj}_2(b)) = \langle b \rangle$.
- Given $f : A \multimap B$ and $g : A \multimap C$, there is a function $(f \pi g) : A \multimap B \uplus C$ given by: $(f \pi g)(a) = (f(a)_I \uplus g(b)_I, \lambda\{\mathsf{inj}_1(i) \mapsto \mathsf{inj}_1(f(a)_m(i)), \mathsf{inj}_2(j) \mapsto \mathsf{inj}_2(g(b)_m(j))\})$.

Note that $\amalg_1; \pi_1 \sim id_A$ and $\amalg_2; \pi_2 \sim id_B$. Moreover, if $f^\dagger$ and $g^\dagger$ are daggers of f and g respectively, then $(f^\dagger \amalg g^\dagger)$ is a dagger of $(f \pi g)$. Now, $(\uplus, \pi_1, \pi_2)$ forms a product in $\mathsf{SNF}_{\mathbb{U}}$:

▶ **Proposition 12.** *Let $f : A \multimap B$, $g : A \multimap C$, and $h : A \multimap B \uplus C$ such that $h; \pi_1 \sim f$ and $h; \pi_2 \sim g$, then $h \sim (f \pi g)$.*

We can define $\mathsf{share}_A : A \multimap A \uplus A$ as $(id_A \pi id_A)$, so $\mathsf{share}_A; (f \uplus g) = (f \pi g)$.

4.3 Semilattice Enriched

As explored in Subsection 2.5, each family $\mathsf{SL}_{\mathbb{U}}(A)$ comes equipped with a join semilattice structure. This can be used to define a join semilattice structure on the spaces of morphisms as well, creating a *semilattice enriched* category [12].

Firstly, we define an order $\prec$ on morphisms $A \multimap B$, given by $f \prec g \iff \forall a \in A.\, f(a) \sqsubseteq^{\mathbb{U}} g(a)$. This is a preorder, and by definition $f \prec g \wedge g \prec f$ implies $f \sim g$. Hence, in $\mathsf{SNF}_{\mathbb{U}}$ as a category (the quotient over the $\mathcal{E}$-category), $\prec$ is antisymmetric. Similar to how $\sim$ is preserved over composition and products, $\prec$ is preserved over the same constructions in the $\mathcal{E}$-category $\mathcal{E}\mathsf{SNF}_{\mathbb{U}}$ and the associated category $\mathsf{SNF}_{\mathbb{U}}$.

We can recover the join operation from Subsection 2.5.

▶ **Corollary 13.** *For A, B and I sets, with I from $\mathbb{U}$, and $F : I \to (A \multimap B)$, there is a supremum $\bigvee F : A \multimap B$ of $F(I)$.*

This can be simply proven by taking $(\bigvee F)(a) = (\Sigma_{i:I} F(i,a)_I,\ \lambda(i,j).\, F(i,a)_m(j))$.

Given $F : I \to (A \multimap B)$ and $G : J \to (B \multimap C)$, where I and J from $\mathbb{U}$, then $(\bigvee F); (\bigvee G) \sim (\bigvee(\lambda(i,j) : I \times J.\, F(i); G(j)))$. Note that for $I = J$, $(\bigvee F); (\bigvee G)$ is not necessarily similar to $\bigvee(\lambda i.\, F(i); G(i))$. To get such a property, we look at ω-chains.

▶ **Definition 14.** *An ω-chain of morphisms is an enumeration of morphisms $F : \mathbb{N} \to (A \multimap B)$ such that for any $n \in \mathbb{N}$, $F(n) \prec F(n+1)$.*

Omega chains are helpful as they allow us to more uniformly use preservation of join over constructions like composition and products. Consider the following results, for example.

- For $F : \mathbb{N} \to (A \multimap B)$ and $G : \mathbb{N} \to (B \multimap C)$ two ω-chains, then $H = \lambda n.F(n); G(n) : \mathbb{N} \to (A \multimap C)$ is an ω-chain and $\bigvee H = (\bigvee F); (\bigvee G)$.
- For $F : \mathbb{N} \to (A \multimap B)$ and $G : \mathbb{N} \to (C \multimap D)$ two ω-chains, then $H = \lambda n.F(n) \uplus G(n) : \mathbb{N} \to (A \uplus C \multimap B \uplus D)$ is an ω-chain and $\bigvee H = (\bigvee F) \uplus (\bigvee G)$.

As before, we can have a specific binary join operation which takes $f, g : A \multimap B$ and produces $f \vee g : A \multimap B$. It holds that $(f \vee g) \sim (\mathsf{share}_A; (f \uplus g); \mathsf{merge}_B)$.

4.4 Monoidal

The Cartesian structure on Rel and $\mathsf{SNF}_{\mathbb{U}}$ is different from the Cartesian structure on Set. Set uses the bifunctor $\times$ which collects pairs of elements, with projections $\mathsf{proj}_1 : A \times B \to A$ and $\mathsf{proj}_2 : A \times B \to B$. We can lift this to the monoidal structure $(\times, |\mathsf{proj}_1|, |\mathsf{proj}_2|)$ on $\mathsf{SNF}_{\mathbb{U}}$ using the functor $|-| : \mathsf{Set} \to \mathsf{SNF}_{\mathbb{U}}$. Similarly, we can lift the comonoid structure on $\times$ in Set as well:

- The unique map $u_A : A \to \{*\}$ is lifted to $|u_A| : A \multimap \{*\}$.
- The map $c_a : A \to A \times A$ which duplicates the argument is lifted to $|c_A| : A \multimap A \times A$.

For any morphism $f : A \multimap B$, $f; |u_B| \prec |u_A|$ and $f; |c_b| \prec |c_a|; (f \times f)$. If f is total, then $f; |u_B| \sim |u_A|$, and if f is deterministic, then $f; |c_b| \sim |c_a|; (f \times f)$.

We cannot equip $\times$ with a monoidal closed structure. The best approximation would be to take $(A \Rightarrow B) = (A \multimap B)$, in which case we can do the following: For $F : A \multimap (B \Rightarrow C)$, let $F' : (A \times B) \multimap C$ be the map $F'(a, b) = (\Sigma_{i:F(a)_I} F(a)_m(i, b)_I,\ \lambda(i, j).\, F(a)_m(i, b)_m(j))$. Only in this direction is the similarity relation preserved. We cannot go into the other direction, since in $(B \Rightarrow C)$ we distinguish between similar but syntactically different families.

5 Inductive Nondeterministic Structures

One of the main applications of the formalisation via slice functions is the relative ease in which it can be used to give a specification for a variety of inductive nondeterministic structures. Suppose in particular we have some inductive structure generated over some signature, for instance generated by algebraic effect operations [19] or a container [1]. Such a signature forms a free monad in the category of sets, which we can lift to $\mathsf{SNF}_{\mathbb{U}}$.

A $\mathbb{U}$-*container* C is an element of the sigma type $\Sigma_{O:\mathsf{Set}}(O \to \mathbb{U})$. It consists of a set of operations O, and a function $ar : O \to \mathbb{U}$ which associates to every operation and arity.

▶ **Definition 15.** *Given a $\mathbb{U}$-container $C = (O, ar)$, the* free monad over C *in Set, is a monad which sends each set A to the set $F_S A$ defined inductively:*

- *For each $a \in A$, there is an element $\mathsf{leaf}(a) \in F_S A$.*
- *For each $\sigma \in O$ and $c : ar(\sigma) \to F_S A$, there is an element $\mathsf{node}_\sigma(c) \in F_S A$.*

It sends a function $f : X \to Y$ to a function $F_S(f) : F_S(X) \to F_S(Y)$ replacing the values at the leaves. The monad unit is given by $\eta^C_A : A \to F_C A$ sending a to $\mathsf{leaf}(a)$, and the monad multiplication $\mu^C_A : F_S F_S A \to F_S A$ is defined inductively as:

- $\mu^C_A(\mathsf{leaf}(t)) = t$.
- $\mu^C_A(\mathsf{node}_\sigma(c)) = \mathsf{node}_\sigma(\lambda i.\mu^C_A(c(i)))$.

We will show that this monad can be lifted to $\mathsf{SNF}_{\mathbb{U}}$ in the next subsection.

5.1 Distributivity

In Set, the free monad distributes over the powerset monad, in the sense that there is a distributivity law between monads [6]. This can be adapted to show that the free monad distributes over the monad $\mathsf{SL}_{\mathbb{U}}$ using a natural transformation $F_S(\mathsf{SL}_{\mathbb{U}} A) \to \mathsf{SL}_{\mathbb{U}}(F_S A)$. We construct this by specifying a set of leaf-positions for each element of $F_S A$, following ideas from the theory of containers [1, 4]. An element $t \in F_S(\mathsf{SL}_{\mathbb{U}} A)$ has at each of its positions a leaf which contains a set of values (a family). If we make a choice of element for each of the leaves of t, we can construct an element of $F_S(A)$. Such a combination of choices is given by associating to each position a choice, which we can do with a dependent map.

We use this idea to lift the Set-endofunctor F_S to an $\mathsf{SNF}_{\mathbb{U}}$-endofunctor. We need a way to transform a morphism $f : A \to \mathsf{SL}_{\mathbb{U}}B$ to $F_SA \to \mathsf{SL}_{\mathbb{U}}(F_SB)$, which accepts a term t as input and replaces each leaf $\mathsf{leaf}(a)$ with some choice value from $f(a)$. We specify a set of indices which collects all possible combinations of choices that can be made.

▶ **Definition 16.** *Given a set A and a function $f : A \to \mathsf{Set}$ assigning a set of choices to each element $a \in A$, we inductively define a function $\mathsf{Pos}(f) : F_SA \to \mathsf{Set}$ assigning to an element $t \in F_SA$ a set of combinations of choices:*

- $\mathsf{Pos}(f)(\mathsf{leaf}(a)) = f(a)$.
- $\mathsf{Pos}(f)(\mathsf{node}_\sigma(c)) = (\Pi_{i:ar(\sigma)}\, \mathsf{Pos}(f)(c(i)))$.

With this function, we can specify our endofunctor on $\mathsf{SNF}_{\mathbb{U}}$, which we call T_C. On objects, T_CA is given by F_CA.

▶ **Definition 17.** *Given a function $f : A \multimap B$, we define $T_Cf : T_CA \multimap T_CB$, with indexing set $T_Cf(t)_I = \mathsf{Pos}(\lambda a.f(a)_I)(t)$ and indexing map where:*

- $T_Cf(\mathsf{leaf}(a))_m = \lambda i : f(a)_I.\, \mathsf{leaf}(f(a)_m(i))$.
- $T_Cf(\mathsf{node}_\sigma(c))_m = \lambda h : ((i : ar(\sigma)) \to T_Cf(c(i))_I).\, \mathsf{node}_\sigma(\lambda i.\, T_Cf(c(i))_m(h(i)))$.

With this definition, T_C forms an endofunctor in $\mathsf{SNF}_{\mathbb{U}}$.

Let us consider whether the above recipe gives us what we want. We independently specify a relation which tells us whether a term is a result of making nondeterministic choices at the leaves. This is akin to how one would lift the free monad to the category of relations. Then, we can check soundness and completeness; meaning the result of our endofunctor contains precisely the choices specified by the relation.

Given $f : A \to \mathsf{SL}_{\mathbb{U}}B$, the *$f$-choice relation* is a relation between T_CA and T_CB inductively defined as follows:

- It relates $\mathsf{leaf}(a)$ to $\mathsf{leaf}(b)$ if b is a result of $f(a)$.
- It relates $\mathsf{node}_\sigma(c)$ to $\mathsf{node}_\sigma(d)$ if for every $i \in ar(\sigma)$, it relates $c(i)$ to $d(i)$.

We can show that our endofunctor T_C is complete over the choice relation.

▶ **Theorem 18.** *Given $f : A \to \mathsf{SL}_{\mathbb{U}}B$, $t \in T_CA$ and $r \in T_CB$, then:*
$\exists i \in T_Cf(t)_I.\, T_Cf(t)_m = r$ if and only if t is related to r by the f-choice relation.

5.2 Monad and Comonad Structure

The free functor forms a monad in the category of relations. The appropriate natural transformations are constructed by lifting the unit and multiplication map of the free monad from Set to $\mathsf{SNF}_{\mathbb{U}}$ by composing it with the unit transformation $\langle - \rangle$ for families. Unit and multiplication are given by $T_C^\eta = |\eta_A^C| : A \multimap T_CA$ and $T_C^\mu = |\mu_A^C| : T_C(T_CA) \multimap T_CA$.

▶ **Proposition 19.** *For each $\mathbb{U}$-container C, (T_C, T_C^η, T_C^μ) is a monad in $\mathsf{SNF}_{\mathbb{U}}$.*

It is possible to reverse the structure of this monad, and construct a comonad.

- Let $T_C^\varepsilon : T_CA \multimap A$ be the morphism sending $\mathsf{leaf}(a)$ to $\langle a \rangle$, and $\mathsf{node}_\sigma(c)$ to $\oslash_A$, then $T_C^\eta \dagger T_C^\varepsilon$.
- Let $T_C^\delta : T_CA \multimap T_C(T_CA)$ be the morphism sending $\mathsf{leaf}(a)$ to $\langle \mathsf{leaf}(\mathsf{leaf}(a)) \rangle$, and $\mathsf{node}_\sigma(c)$ to $\langle \mathsf{leaf}(\mathsf{node}_\sigma(c)) \rangle \vee N$, where $N = (N_I, N_m)$ is given by $N_I = \Pi_{i:ar(\sigma)}\, T_C^\delta(c(i))_I$ and $N_m = \lambda(i,j).\, \mathsf{node}_\sigma(T_C^\delta(c(i))_m(j))$. Then $T_C^\mu \dagger T_C^\delta$.

Hence $(T_C, T^\varepsilon, T^\delta)$ is a comonad. Suppose for instance $T_C A$ describes a set of computational processes, with η creating a process that immediately terminates and gives a result, and μ merges a two-staged process into a single process. Then ε extracts from a process any result that is immediately produced, and δ nondeterministically splits a process into two stages. The monad and comonad structure interacts in interesting ways.

- $T^\eta_C; T^\varepsilon_C \sim id$; we can extract a result from a process that immediately terminates.
- $T^\delta_C; T^\mu_C \sim id$; when we split a process into two stages, and merge the stages, we get the original process back.
- $T^\mu_C; T^\varepsilon_C \sim T^\varepsilon_C; T^\varepsilon_C$; extracting results from a process is like extracting from its stages.
- $T^\eta_C; T^\delta_C \sim T^\eta_C; T^\eta_C$; splitting a terminating process gives two terminating stages.

6 Example Processes

We look at some examples of nondeterministic processes we can represent with slice nondeterministic functions.

6.1 Interleaving Concurrency

Let us first look at *interleaving concurrency*, which can be modelled by the interleaving operation on actions. Consider lists X^* over X inductively defined as $[] \in A^*$ and for $x \in X$ and $l \in X^*$, $x :: l \in X^*$.

▶ **Definition 20.** *The* interleaving operations *on lists* $\|, \|^l, \|^r \colon X^* \times X^* \multimap X^*$ *are inductively defined as:*

- $(a \parallel b) = (a \parallel^l b) \vee (a \parallel^r b)$.
- $([] \parallel^l a) = \langle a \rangle = (a \parallel^r [])$.
- $(x :: a \parallel^l b) = \mathsf{SL}_{\mathbb{U}}(\lambda c.x :: c)(a \parallel b) = (a \parallel^r x :: a)$.

The definition of parallel operation is split into three clauses in order to satisfy Agda's termination checker. In the axioms of *process algebra* (see e.g. [8]), the parallel operation was already split into two clauses corresponding to our $\parallel$ and $\parallel^l$, in order for the induction principle to apply. Here, we separately define $\parallel^r$, since without it, a non-trivial order on arguments need to be established in order to prove termination of the function.

The parallel operation satisfies some interesting properties, which can be summarised in the following proposition.

▶ **Proposition 21.** $(\|\colon X^* \times X^* \multimap X^*, e : \top \multimap X^*)$ *where* $e(*) = \langle [] \rangle$ *forms a commutative monoid in* $\mathsf{SNF}_{\mathbb{U}}$

▶ Remark. The approach of using slices for modelling nondeterminism was used at first in order to prove properties like associativity of the parallel operation. Earlier efforts to model interleaving concurrency used the representation of the finite powerset monad as a free idempotent commutative monoid. Proving equations of the interleaving operation required precise rewriting of terms using associativity and commutativity of the monoid. Nondeterminism via slices on the other hand uses a case analysis on the elements of the indexing set, which in this case represents the exact nondeterministic choices made by the parallel operation. Though the proof of associativity is still not simple, the technique of using slice nondeterminism greatly reduces the level of bureaucracy necessary.

It can be verified that this model satisfies the equations of process algebra as for instance specified in [8]. The development in Agda also contains a formalisation of previous work from the author [24] which considers a categorical model for interleaving algebraic effects using $\mathsf{SNF}_{\mathsf{Set}}$.

6.2 Iterated Processes

As our second example, we study a simple state automata which is given by a state space, and a nondeterministic transition function into the set of states and outputs. We can iterate over chains of transitions using a natural number, and then take the supremum over this natural number to get a single collection of all possible outputs.

We define the iteration function $\mathsf{Iter}_{A,B} : (A \multimap B \uplus A) \to \mathbb{N} \to (A \multimap B)$, which takes a morphism $H : A \multimap B \uplus A$ and a natural number $n \in \mathbb{N}$, and iterates H n-times, gathering all results from B it produces. Defined inductively on $n \in \mathbb{N}$, it does the following:

- $\mathsf{Iter}_{A,B}(H, 0) = \lambda a.\oslash_B$.
- $\mathsf{Iter}_{A,B}(H, n+1) = H; (id_B \uplus \mathsf{Iter}_{A,B}(H, n))$.

$\mathsf{Iter}_{A,B}(H, n+1)(a)$ looks at $H(a)$, which is a collection of elements of $B \uplus A$. It keeps all results from B and joins them with elements of $\mathsf{Iter}_{A,B}(H, n)(a')$ for any result a' from A. The function $\mathsf{Iter}_{A,B}(H, -)$ forms an ω-chain.

We define $\mathsf{Iter}\omega_{A,B} : (A \multimap B \uplus A) \to (A \multimap B)$ by taking the countable join of $\mathsf{Iter}_{A,B}$. We have the following results:

- Both $\mathsf{Iter}_{A,B}$ and $\mathsf{Iter}\omega_{A,B}$ preserve the similarity relation $\sim$ on morphisms, and hence are well defined as functions on morphisms in $\mathsf{SNF}_{\mathbb{U}}$.
- For $f : A \multimap B$, $H : B \multimap C \uplus A$, and $n \in \mathbb{N}$, $\mathsf{Iter}_{A,B}(f; H, n) \sim f; (\mathsf{Iter}_{A,B}(H; (id_C \uplus f), n))$, hence $\mathsf{Iter}\omega_{A,B}(f; H) \sim f; (\mathsf{Iter}\omega_{A,B}(H; (id_C \uplus f)))$.
- For $f : B \multimap C$, $H : A \multimap B \uplus A$, and $n \in \mathbb{N}$, $\mathsf{Iter}_{A,B}(H; (f \uplus id_A), n) \sim \mathsf{Iter}_{A,B}(H, n); f$, and hence $\mathsf{Iter}\omega_{A,B}(H; (f \uplus id_A), n) \sim \mathsf{Iter}\omega_{A,B}(H, n); f$.
- For $H : A \multimap B \uplus A$, $\mathsf{Iter}\omega_{A,B}(H) \sim H; (id_B \uplus \mathsf{Iter}\omega_{A,B}(H, n))$.
- For $H : A \multimap B \uplus A$ and $K : B \multimap C \uplus B$, $\mathsf{Iter}\omega_{A,B}(H); \mathsf{Iter}\omega_{B,C}(K) = \bigvee(\lambda n.\, \mathsf{Iter}_{A,B}(H, n); \mathsf{Iter}_{B,C}(K, n))$.
- $\mathsf{Iter}\omega_{A,A}(\mathsf{inj}_1) \sim \langle - \rangle_A$.

The $\mathsf{Iter}\omega$ operation can be used as a basis for a traced monoidal [5] operation for $\mathsf{SNF}_{\mathbb{U}}$, and can moreover be used to model recursive processes using ω-chains. As such, it could form a basis for *domain theoretic* denotations using streams as in previous work from the author [23]. There, such streams were used to define denotational equivalence of recursive effectful programs.

6.3 Labelled transition systems

We consider a common nondeterministic process called the *labelled transition system*. Here each possible transition is given a corresponding label designating some input, and a decidable predicate on states specifying final states. We leave the choice of initial state as a parameter.

▶ **Definition 22.** *A* labelled transition system *(lts) over a set of labels $A : \mathbb{U}$ is specified by a triple (S, t, e) consisting of a set of states S, a transition map $t : S \times A \multimap S$ and a function checking for ending states $e : S \to \mathbb{B}$.*

We denote the set of labelled transition systems over A as $\mathtt{LTS}(A)$. We are often interested in checking which series of labels would lead to a final state. A list of labels l is *accepted* by an lts (S, t, e) on some initial state $s \in S$ if there is a path of labelled transitions corresponding to the labels of l, from s to some final state from $e^{-1}(\mathsf{true})$. There are two ways to specify which lists of labels are accepted. We can either directly enumerate them using accepted paths, or we can inductively check whether a list is accepted.

We can generate a collection of paths which will lead to termination inductively as a dependent function $\mathsf{LTSCol}(A) : ((S, t, r) : \mathsf{LTS}(A)) \to \mathbb{N} \to (S \multimap \mathsf{SL}_{\mathbb{U}}(A^*))$, which for a natural number $n \in \mathbb{N}$ collects all accepted lists of length n.

- $\mathsf{LTSCol}(A)(S,t,e)(0)(s) = \begin{cases} \langle [] \rangle & \text{if } (e(s) = \mathsf{true}) \\ \varnothing_{A^*} & \text{if } (e(s) = \mathsf{false}) \end{cases}$
- $\mathsf{LTSCol}(A)(S,t,e)(n+1)(s) = (I, h)$ where I is a set of triples consisting of $a \in A$, $i \in t(a,s)_I$ and $j \in \mathsf{LTSCol}(A)(S,t,e)(n)(t(s,a)_m(i))_I$, and $h(a,i,j) = a :: \mathsf{LTSCol}(A)(S,t,e)(n)(t(s,a)_m(i))_m(j)$.

We then define $\mathsf{LTSCol}\omega$ as the supremum over $\mathbb{N}$ of LTSCol.

Alternatively, we can define a predicate which checks whether a certain list is accepted. $\mathsf{LTSaccept}(A) : ((S, t, e) : \mathsf{LTS}(A)) \to S \to A^* \to \mathsf{Set}$ where:

- $[]$ is accepted on initial state s if $e(s) = \mathsf{true}$.
- $a :: l$ is accepted on initial state s if there is a state z in the collection $t(s, a)$ such that l is accepted on initial state z.

▶ **Proposition 23.** *The collection of lists generated by* $\mathsf{LTSCol}\omega$ *consists exactly of the lists accepted according to* $\mathsf{LTSaccept}$.

7 Conclusions

We have looked at a formalism for constructively representing denotations of nondeterministic processes. These models are inherently intensional as they send inputs to outputs dependent on concrete nondeterministic choices, as opposed to giving an external predicate for checking whether an output is possible. The model is moreover directional, as only a subset of well-behaved morphisms have daggers. As such, the model stays closer to natural occurrences of computational nondeterministic processes. Nondeterminism is often guided by a source of unpredictability, like sampling spaces in probability theory. In terms of this model, the nondeterministic sampling space takes the form of an indexing set of a family. Potential extensions to probabilistic models could be considered in the future.

We focused mainly on a specific universe of indexing sets $\mathbb{U}$. Earlier versions of the work used the collection of sets as a universe. Other possible universes may be explored in the future too. Varying the universe allows us to model different categories, for instance:

- Taking the universe $\{\top\}$, we effectively retrieve the category of sets Set.
- Taking the universe $\{\bot, \top\}$, we model the category of partial functions Par.
- Taking finite sets as a universe, we get finite nondeterministic functions.
- Excluding functions and Π constructions from $\mathbb{U}$, we get countably nondeterministic functions, e.g. programs which can sample natural numbers nondeterministically.

If there is an inclusion of universes, there is a functor between their respective categories. There are more variations to consider, since we can restrict our sets of morphisms to those satisfying a chosen subset of properties; total, deterministic, surjective, injective, daggerable.

Note that in the above formulation of partial functions, it is decidable whether a function gives a result or not. If you want to capture undecidability, you should instead use the subcategory of deterministic slice functions, where inhabitance of an indexing set is undecidable. This does however require you to show that all possible results are equal.

The slice nondeterminism framework is in some aspects much easier to work with then algebraic representations of nondeterminism. For instance, equations like idempotency, associativity and commutativity of nondeterministic choice (implemented by the join operation) are easily proven using case distinctions on the indexing set. Moreover, proofs that normally need such equations can be easily shown without needing to explicitly call these equations, avoiding the need to specify a concrete sequence of rewrites. In particular, the formalisation of properties for interleaving concurrency was hampered by the need of intensive equational reasoning when working with a monoidal representation of the finite powerset monad. But switching over to slices, many of these former difficulties were easily avoided.

7.1 Agda Categories Library

The formalisation of this paper is grounded in the framework of setoid enriched categories ($\mathcal{E}$-categories), like the development of the Agda Categories library. The core of the formalisation only uses Agda's standard library, for ease of adaptability. In separate files, some of the results are linked up with the Agda Categories library, and concrete properties are shown. These properties are as follows.

We have shown the following things concerning $\mathsf{SNF}_{\mathbb{U}}$.

- $\mathsf{SNF}_{\mathbb{U}}$ is a category.
- $\mathsf{SNF}_{\mathbb{U}}$ is symmetric monoidal with respect to the product and disjoint union bifunctor.
- $\mathsf{SNF}_{\mathbb{U}}$ is Cartesian and Cocartesian

Secondly, we have shown that the endofunctor $\mathsf{SL}_{\mathbb{U}}$ can be used to form a monad in the category of setoids. This is the perfect candidate for a powerset monad over setoids, which could be called the *powersetoid* monad, since it is a model of the powertheory. Formalisation in terms of setoids specifically may be a future avenue for research.

Lastly, consider the slice category with indexing sets ranging over Set instead of $\mathbb{U}$, then:

▶ **Theorem 24.** *The category $\mathit{SNF}_{\mathsf{Set}}$ is equivalent to the category of relations.*

7.2 Denotational Semantics

Some examples were considered in Section 6, regarding finite automata and process algebra. An example for the future is to formalise functional nondeterministic languages as studied by Lassen [14]. Given the fact that families are closed under taking limits (see iterated processes in Section 6), it should be possible to create a denotational model for nondeterministic functional languages, like the one employed in previous work using streams [23]. A simple example has been worked out, providing a denotational semantics for an untyped call-by-value lambda calculus with added nondeterminism.

More generally, using a specification by nondeterministic runner [22, 24], we can model a variety of stateful and nondeterministic effects. When described in this framework, we use an indexing set to enumerate all possible handlers [20] for the effect, which can then be used to extract a set of possible results of the computation. Part of the theory surrounding this is formalised in the current development, and studying such algebraic models using the formalisation presented in this paper is subject to future research.

References

1 Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005. doi:10.1016/j.tcs.2005.06.002.

2 Samson Abramsky. Domain theory in logical form. *Ann. Pure Appl. Log.*, 51(1–2):1–77, 1991. doi:10.1016/0168-0072(91)90065-T.

3 Peter Aczel. Galois: a theory development project, 1993. In: A report on work in progress for the Turin meeting on the Representation of Logical Frameworks.

4 Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor Mcbride, and Peter Morris. Indexed containers. *Journal of Functional Programming*, 25:e5, 2015. doi:10.1017/S095679681500009X.

5 Joyal Andre, Ross Street, and Dominic Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119:447–468, April 1996. doi:10.1017/S0305004100074338.

6 Jon Beck. Distributive laws. In B. Eckmann, editor, *Seminar on Triples and Categorical Homology Theory*, pages 119–140, Berlin, Heidelberg, 1969. Springer Berlin Heidelberg. doi:10.1007/BFb0083084.

7 Jean Bénabou. Introduction to bicategories. In *Reports of the Midwest Category Seminar*, pages 1–77, Berlin, Heidelberg, 1967. Springer Berlin Heidelberg.

8 J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985. doi:10.1016/0304-3975(85)90088-X.

9 Gerhard Gierz, Karl Heinrich Hofmann, Klaus Keimel, Jimmie Lawson, Michael Mislove, and Dana S. Scott. *Continuous Lattices and Domains*. Cambridge University Press, Cambridge, 2003.

10 C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978. doi:10.1145/359576.359585.

11 Markus Holzer and Martin Kutrib. Reversible nondeterministic finite automata. In *International Workshop on Reversible Computation*, pages 35–51, May 2017. doi:10.1007/978-3-319-59936-6_3.

12 G. Kelly. The basic concepts of enriched category theory. *Reprints in Theory and Applications of Categories [electronic only]*, 2005, January 2005.

13 Yoshiki Kinoshita. A bicategorical analysis of E-categories. *Mathematica Japonica*, 47:157–170, 1998.

14 Søren B. Lassen. *Relational Reasoning about Functions and Nondeterminism*. PhD thesis, University of Aarhus, 1998.

15 Paul Blain Levy. Similarity quotients as final coalgebras. In Martin Hofmann, editor, *FoSSaCS 2011*, volume 6604 of *LNCS*, pages 27–41. Springer, 2011. doi:10.1007/978-3-642-19805-2_3.

16 Saunders MacLane. *Categories for the Working Mathematician*. Springer-Verlag, New York, 1971. Graduate Texts in Mathematics, Vol. 5.

17 Rasmus Ejlers Møgelberg and Andrea Vezzosi. Two guarded recursive powerdomains for applicative simulation. In Ana Sokolova, editor, *Proceedings of 37th Conference on Mathematical Foundations of Programming Semantics, MFPS 2021*, volume 351 of *Electron. Proc. in Theor. Comput. Sci.*, pages 200–217, 2021. doi:10.4204/EPTCS.351.13.

18 Gordon D. Plotkin. A powerdomain construction. *Siam J. Comput.*, 5(3):452–487, 1976. doi:10.1137/0205035.

19 Gordon D. Plotkin and John Power. Adequacy for algebraic effects. In Furio Honsell and Marino Miculan, editors, *FoSSaCS 2001*, volume 2030 of *LNCS*, pages 1–24, 2001. doi:10.1007/3-540-45315-6_1.

20 Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Log. Methods Comput. Sci.*, 9(4, article 23):1–36, 2013. doi:10.2168/lmcs-9(4:23)2013.

21 Albert Marchienus Thijs. *Simulation and Fixpoint Semantics*. PhD thesis, University of Groningen, 1996. URL: `https://research.rug.nl/en/publications/simulation-and-fixpoint-semantics`.

22 Tarmo Uustalu. Stateful runners of effectful computations. *Electron. Notes Theor. Comput. Sci.*, 319:403–421, 2015. `doi:10.1016/j.entcs.2015.12.024`.

23 Niccolò Veltri and Niels F. W. Voorneveld. Streams of approximations, equivalence of recursive effectful programs. In Ekaterina Komendantskaya, editor, *Mathematics of Program Construction - 14th International Conference, MPC 2022, Tbilisi, Georgia, September 26-28, 2022, Proceedings*, volume 13544 of *Lecture Notes in Computer Science*, pages 198–221. Springer, 2022. `doi:10.1007/978-3-031-16912-0_8`.

24 Niels F. W. Voorneveld. Runners for interleaving algebraic effects. In Helmut Seidl, Zhiming Liu, and Corina S. Pasareanu, editors, *Theoretical Aspects of Computing – ICTAC 2022*, pages 407–424, Cham, 2022. Springer International Publishing.

Foundational Verification of Stateful P4 Packet Processing

Qinshi Wang
Princeton University, NJ, USA

Mengying Pan
Princeton University, NJ, USA

Shengyi Wang
Princeton University, NJ, USA

Ryan Doenges
Cornell University, Ithaca, NY, USA

Lennart Beringer
Princeton University, NJ, USA

Andrew W. Appel
Princeton University, NJ, USA

Abstract

P4 is a standardized programming language for the network data plane. But P4 is not just for routing anymore. As programmable switches support stateful objects, P4 programs move beyond just stateless forwarders into new stateful applications: network telemetry (heavy hitters, DDoS detection, performance monitoring), middleboxes (firewalls, NAT, load balancers, intrusion detection), and distributed services (in-network caching, lock management, conflict detection). The complexity of stateful programs and their richer specifications are beyond what existing P4 program verifiers can handle.

Verifiable P4 is a new interactive verification framework for P4 that (1) allows reasoning about multi-packet properties by specifying the per-packet relation between initial and final states; (2) performs modular verification, especially providing a modular description for stateful objects; (3) is foundational, i.e., with a machine-checked soundness proof with respect to a formal operational semantics of $P4_{16}$ (the current specification of P4) in Coq. In addition, our framework includes a proved-correct reference interpreter.

We demonstrate the framework with the specification and verification of a stateful firewall that uses a sliding-window Bloom filter on a Tofino switch to block (most) unsolicited traffic.

2012 ACM Subject Classification Security and privacy → Logic and verification

Keywords and phrases Software Defined Networking, Verifiable P4, Stateful data plane programming

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.32

Supplementary Material *Software*: `github.com/verified-network-toolchain/VerifiableP4`
archived at `swh:1:dir:fc1600d9109a91d9b355358af11dd9aaebd311d6`

Funding This material is based upon work supported by DARPA Contracts HR001120C0160 and HR001120C0107, and by NSF grant FMiTF-1918396.

1 Introduction

The data plane for software-defined network switches is increasingly programmed in P4 [3]. But P4 is a quirky language, and programs are often contorted to fit within the constraints of a particular target architecture, so the correctness of these programs has become a concern. To address that concern, there are several verification tools for P4 programs (see §7).

© Qinshi Wang, Mengying Pan, Shengyi Wang, Ryan Doenges, Lennart Beringer, and Andrew W. Appel;
licensed under Creative Commons License CC-BY 4.0

In many classic P4 applications, processing a packet does not change the state of the switch. However, recent applications are *stateful* and go far beyond making routing decisions: the processing of a packet may alter the state of registers or result in the installation of new forwarding rules, and thus affect the processing of following packets. Stateful applications include network telemetry systems (SketchLib [17], BeauCoup [2], FlowRadar [14]), network functions (SilkRoad [16]), and distributed services (NetCache [12], NetLock [26]). Unfortunately, existing tools have limited reasoning capabilities for registers or multi-packet policies.

To begin addressing these shortcomings, we present a foundational framework for specifying and reasoning about data-plane packet processing on a stateful P4 switch. Implemented in the Coq proof assistant, our system facilitates semi-interactive verification of stateful P4 programs and is justified w.r.t. a precise operational semantics of $\mathrm{P4}_{16}$.

As an example for a multi-packet policy, consider a stateful firewall that protects the internal network from unsolicited traffic. External packets may pass through the firewall only if they are responses to recent outgoing requests to the same IP address, modulo a small tolerable rate of false positives. But *no valid incoming responses may be blocked.*

To specify the latter property more precisely, let T be the valid response time window, h be the history of packets processed, p be the current packet, and r the action on p (forward or drop). We want that for every integer i, we have

$$\begin{array}{l} p.\mathrm{dir} = \mathrm{in} \wedge h[i].\mathrm{dir} = \mathrm{out} \wedge h[i].\mathrm{dst} = p.\mathrm{src} \\ \wedge\ h[i].\mathrm{src} = p.\mathrm{dst} \wedge p.t - h[i].t \leq T \end{array} \implies r \neq \mathrm{drop} \tag{1}$$

To maintain the window of length T, a P4 implementation necessarily maintains state; but existing verifiers either do not handle state at all, have specification languages that are too weak to relate the pre-state to the post-state after processing a packet (see §7), or do not permit reasoning about multi-packet properties.

Property 1 holds even for a firewall admits every packet, but we prove the P4 program correct with respect to a functional model, for which properties such as 1 can be derived.

Our verifier connects P4 verification with reasoning about multi-packet policies using assertions that are syntactically constrained but permit reference to arbitrary Coq constructions. The user must equip each P4 function with a specification that asserts adherence to a model-level counterpart, i.e. a Coq function (or proposition) describing its effect in terms of a semantic model of packet processing. We justify the program logic in Coq by a soundness proof w.r.t. the operational semantics.

Our operational semantics builds on Petr4 [5] but is defined as an inductive relation in Coq rather than Petr4's executable Ocaml program. Indeed, part of our effort consisted in understanding aspects related to nondeterminism and partly uninitialized data structures, that are not modeled in Petr4 and are specified partially at best in the P4 manual [3]. We report numerous inaccuracies etc. in said manual later in the paper. We also provide our own interpreter and prove (in Coq) that it exhibits behavior consistent with our semantics, e.g. by resolving determinism in an appropriate manner.

Contributions

1. We present *Verifiable P4*, a system for verifying stateful specifications of P4 data-plane packet processing. Our logic does not cover packet parsing and deparsing, which we leave for future work. We provide automation support for proving that P4 code correctly implements a functional model; users prove interactively that a functional model establishes a high-level property, such as "no valid responses are blocked".

2. We propose a hierarchical representation of states used in semantics, specification, and verification that improves modularity; unlike some previous P4 semantics, we enforce a phase distinction between *instantiation* (that populates this hierarchy) and *run-time packet processing*.
3. We formalize the operational semantics of P4 (incl. parsing and deparsing) in Coq, and prove soundness of our verifier and of a reference interpreter. Our operational semantics is the first formal specification to include abstract methods (a P4 feature that encodes a P4 program in an extern object), and Verifiable P4 is the first verifier to support them. The development of our semantics uncovered several inaccuracies, bugs, and ambiguities in the $P4_{16}$ reference manual that had also escaped the Petr4 semantics and tools. We contributed bug-fixes as pull requests to the manual and discussed them with the $P4_{16}$ committee.
4. We develop a model-level axiomatization of temporal windows that can serve as a blueprint for other finite-horizon data structures over streams implemented in P4 or other languages. We give a concise specification of a sliding window Bloom filter (SBF) and a high-performance implementation in P4, with an application to a stateful firewall. We show how to connect model-level firewall verification to P4 verification in Coq, and briefly discuss other examples.

The source files associated with this paper are available at
`https://github.com/verified-network-toolchain/VerifiableP4/tree/Feb2023`

2 Example: A stateful firewall

The data-plane of a software-defined network (SDN) switch processes about 10^9 packets per second (or 3200 Gb per second) in each "pipeline" [13]. To accomplish that, each packet is allowed only one trip through a highly pipelined "match-action unit". To program this unconventional model of computation, the P4 language was designed with some inspiration from Verilog. It combines logical with architectural aspects but is reasonably modular – one can divide programs into reusable libraries and the client programs that make use of them.

P4 is a horrible but useful programming language

P4 was designed to support packet routing, so all of its control structures are designed around match-action tables. One generally cannot access the same (persistent) data more than once per packet, so things must be accomplished in a read-modify-write with user-specifiable "modify." Except for the parser component, P4 code does not contain loops, hence every packet must process in n pipeline stages (so at least we can mostly use a big-step semantics!). But P4 is *useful* because it gives a reasonably portable way to program high-performance network switches from different manufacturers.

The main elements of a typical P4 program are as follows.

Parsers extract packet headers and the payload from a packet. Headers – and typical auxiliary data structures throughout the code – are represented as `structs` that can be accessed as in the C language.

Control blocks describe how headers are processed. A control block contains a list of declarations, plus the control flow. Declarations include

extern objects, which are architecture-specific constructs such as registers, cryptographic operations, or other domain-specific functions,

actions, which assign flags (e.g. DROP) or other values (e.g. a specific output port) to suitable header fields,

match-action tables, which match header fields against predefined keys and trigger actions or extern objects accordingly.

The control flow is specified by an imperative program ("apply function") that governs under which conditions or order ("pipeline") the match-action tables are invoked on a header.

Deparsers reasssemble the processed packet headers with the payload and emit the packet to the network.

Modularity of P4 arises from the ability to instantiate registers, other extern objects, and control blocks multiple times, and to define them in a style reminiscent of classes in e.g. Java, with instantiatable parameters.

The low-level complexities of P4 make programs difficult to read and write, so code (including our running example) is increasingly not written directly in P4 but in one of the various front-end languages that target P4 (to synthesize our SBF/firewall we used CatQL, a functional-style language under development by the second author). These experimental languages don't yet have what we would call a formal semantics or even a real reference manual, so we verify the P4 code rather than verify the ultimate source programs.

The difficulty of writing and reading P4 programs is a strong motivation to formally verify P4 with respect to model-level or higher-level specifications, which (even though in Coq) will be more accessible to engineers than the source code. Our tool can verify both reusable libraries – like the Bloom filter control block – and their clients – like the firewall, which is a control block separate from the Bloom filter. Where the library implements nontrivial algorithms, users may need more expertise, but verifying simple clients (even those using sophisticated libraries) is easier.

The Sliding Window Boom Filter and its Firewall Client

We continue with our running example. The stateful firewall must remember recent IP addresses in a data structure that can fit in the switch's persistent registers and can be accessed in constant time within the switch's pipeline constraints. To implement this in a P4 switch *running entirely in the data plane,* we use a succinct constant-time-access data structure: a sliding-window Bloom filter (SBF).

A Bloom filter is a hash table, without collision detection, in which value i is hashed with r different functions and a bit is set to 1 at those indexes $f_1(i), f_2(i), \ldots f_r(i)$. Lookup($i$) returns 1 if *all* of the $f_j(i)$ are 1; the probability of a false negative is 0, and the probability of a false positive is small (the product of the individual false-positive probabilities). On the Tofino switch [10], one cannot make r different accesses to the same array, so we implement the Bloom filter with r separate arrays.

A firewall that runs for a long time does not want its hash tables to fill up with stale data, but standard Bloom filters do not (soundly) support deletion; so we use a *sliding window* Bloom filter [30], as we will now explain.

A SBF has k "panes". Each pane is a Bloom filter containing r rows with S slots each. At any given time, $k-1$ panes store the "recent outgoing packets;" we add new IP addresses only to the most recent pane, and the extra pane is in the process of being incrementally cleared to prepare for reuse. We use $T = 60$ seconds as our window, i.e. the period for which we guarantee absence of false negatives. We rotate panes every $T/(k-2)$ seconds. That is because, in the worst case of querying, the most recent pane has just started but the

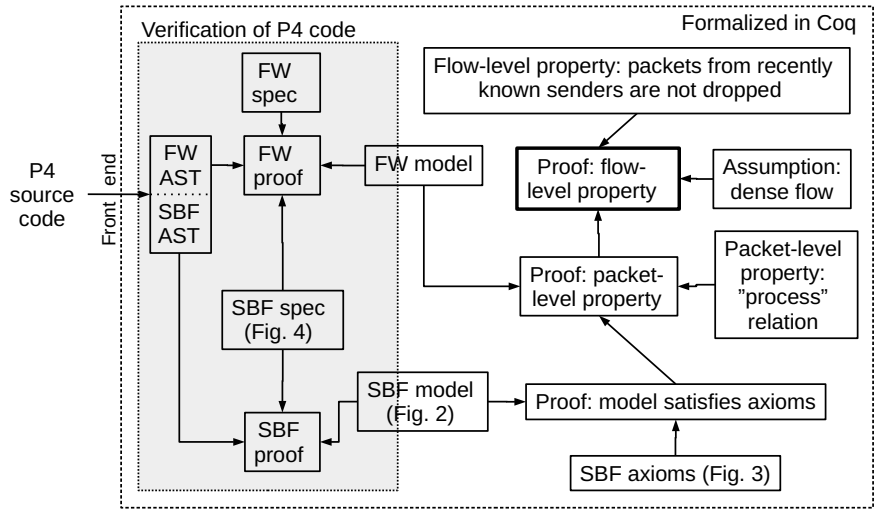


Figure 1 Overview of Stateful Firewall Verification.

remaining $k-2$ panes can still guarantee that keys inserted within T seconds are present. The client must incrementally clear the slots in the timed-out pane at least once every $T/C \approx 100\,\mu$s, where $C = (k-2)S$, so that the oldest pane will be fully cleared before reuse.

Our SBF has $k = 4$ and $r = 3$, and is lightly modular: the pane control contains r row instances – a row being a register of width $S = 2^{18}$ –, the SBF control contains k pane instances, and the SBF itself is a control that is instantiated once by the firewall client to make decisions about which packets to drop. Our verification reflects this structure, exploiting the phase distinction between control-instantiation and packet-processing. The length of our program is largely independent of the number of panes in each window and the number of rows in each pane.

In total, our example is a 600-line P4 program that you really don't want to see, and we will accommodate you for the most part.

2.1 Proof organization and functional model

Fig. 1 shows the specifications and proofs, concluding with the main theorem: on any "dense flow", the firewall satisfies the property shown as equation (1)[1]

The *packet-level correctness property* is a relation, $process(s, p, s', r)$, where s, s' are the switch's state before and after processing the packet p and r is the result: forward or drop. To reason about multi-packet policies, we define (purely in Coq, independent of P4) a notion

[1] We model time in correspondence with the timestamp in metadata from the switch, which we assume the switch inserts correctly. The time $p.t$ in our flow-level specification is an unbounded mathematical integer; the timestamp in the packet header is a 48-bit unsigned integer measured in nanoseconds, $p.t \bmod 2^{48}$. Our proof that the program is correct does allow the timestamp to cross the 2^{48} boundary. A *dense flow* is one in which the mathematical timestamps are monotonically increasing – this is guaranteed by the switch ingress hardware – and in which there is never a gap greater than $100\mu s$ (that is, T/C) between packets. We need this assumption to ensure that the P4 program keeps up with its incremental clearing obligation. We can guarantee a dense flow using the Tofino switch's *packet generator*, a component just before the P4 pipeline that can be configured to insert extra packets at the desired ($100\mu s$) intervals.

Listing 1 Functional model of an SBF, written in Coq's functional language (excerpt).

```
Definition row := list bool.
Definition row_insert (r : row) (i : Z) : row := upd_Znth i r true.

Definition pane := list row.
Definition pane_insert (a:pane) (is:list Z):pane:= map2 row_insert a is.

Record SBF := mk_SBF
  { SBF_panes : list pane; SBF_clear_index : Z; SBF_timer : Z * bool; }.

Definition get_clear_pane (t : Z * bool) : Z := fst t / pane_tick_tocks.

Definition SBFinsert (f : SBF)(tick : bool)(is : list Z):SBF :=
  let '(mk_SBF panes clear_index timer) := f in
  let new_clear_index := update_clear_index clear_index in
  let timer := update_timer timer tick in
  let c := get_clear_pane timer in
  let i := get_insert_pane c in
  let panes := panes[c := pane_clear panes[c] clear_index] in
  let panes := panes[i := pane_insert panes[i] is] in
  mk_SBF panes new_clear_index timer.
```

of transitive closure of *process* that maintains the histories of incoming and outgoing packets, the latter list in fact operating over optional packets to model *drop*. Thus, all that remains is to prove that the P4 program, on a single packet, satisfies the *process* relation.

Our tool's front end generates separate abstract syntax trees (ASTs) for separate control blocks, as indicated in Fig. 1. We use Verifiable P4 to prove semi-automatically that the SBF code implements the SBF model and that the firewall (FW) code implements the FW model. We prove directly in Coq that the SBF model obeys certain abstract SBF axioms (discussed below) which in turn feeds into the proof that the FW model satisfies the *process* relation.

Listing 1 shows the data structures of the SBF functional model, and the function modeling the insertion operation. The model mirrors the hierarchy of control blocks in the P4 code: a *row* is a list of Booleans (modeling the contents of a hash table), a *pane* is a list of rows, and an *SBF* is a list of panes, together with fields for the timer and the clearing maintenance. The full model contains additional functions `SBFquery`, `SBFclear`, and the initial state `SBFempty`.

The functional model will be referred to in specifications in the following section, but the proof of the packet-level policy only relies on certain axioms that specify these operations and express the "no recent false negatives" property; see Listing 2. For example, **QueryInsertSame** has the premise $t \leq t' \leq t + T$, saying that an element inserted into the table at time t will be retrievable up to T seconds later. The three final laws express that the client must perform an incremental-clear (or an insert) at least every T/C seconds.

▶ **Theorem 1** (Coq). *The functional model of the SBF (Listing 1) satisfies the axioms given in Listing 2.*

2.2 Function specifications

Our logic equips each P4 function with a *function specification*, as illustrated in Listing 3 for the SBF insertion operation. The specification components may be understood as follows:

Listing 2 Axioms of an SBF, written in Coq.

```
"ok_until f t" means that the client has performed its
incremental clearing obligation at least through time t.

If state f is OK until (at least) deadline t, you insert IP address h at time t
and then look up h at time t' no more than T seconds later, it will be present.
QueryInsertSame : ∀f t t' h, ok_until f t → t ≤ t' ≤ t+T →
     SBFquery (SBFinsert f (t, h)) (t', h) = Some true.

If you could find IP address h' in the state, and you insert (perhaps different)
IP address h, then h' is still in there.
QueryInsertOther : ∀f t t' h h', SBFquery f (t', h') = Some true →
     ok_until f t → t ≤ t' → SBFquery (SBFinsert f (t, h)) (t', h') = Some true.

Doing a clear-step won't affect any query results.
QueryClear : ∀f t t' h, ok_until f t → t ≤ t' →
     SBFquery f (t', h) = SBFquery (SBFclear f t) (t', h).

If state f is OK until deadline t, you can extend its
deadline by up to 100 microseconds (T/C) by [insert].
OkInsert : ∀f t t' h, ok_until f t → t ≤t' ≤ t+T/C →
     ok_until (SBFinsert f (t,h)) t'.

If state f is OK until deadline t, you can extend its
deadline by up to 100 microseconds (T/C) by [clear].
OkClear : ∀f t t', ok_until f t → t ≤t' ≤ t+T/C → ok_until (SBFclear f t) t'.

The initial state is OK until its preset deadline.
OkEmpty : ∀t, ok_until (SBFempty t) t.
```

MOD: the insertion code modifies stack variables with no restriction and modifies (only) within the external object rooted at path p (but not other objects).

WITH: the universally quantified abstract (logical) variables bound here (k, tstamp, f) have scope that extends to the end of the postcondition.

PRE: The precondition describes the state before executing the insertion, in three clauses – ARG: function parameter values, MEM: program stack variables (e.g. headers and structs, empty in our example), and EXT: external object contents (persistent registers).

POST: The postcondition describes the state after function execution, also divided into three parts (with return-value together with out-parameter-value).

In this case, the precondition says, "The arguments are the key, an operation code with constant value INSERT (this argument selects which SBF operation is performed), a time-stamp *tstamp*, and an 8-bit value 0; there is no header or struct to take note of; and the external registers rooted at p represent a SBF with contents f." Similarly, the postcondition says, "The out parameter is 8-bit value 0, and the external registers rooted at p represent an updated SBF as described by the SBFinsert function from the functional model."

Similar to representation predicates in separation logic [21], SBF_repr is a user-defined predicate that relates the abstract view of an SBF to the corresponding P4 control, as laid out in Tofino registers. The predicate's definition again mirrors the nesting structure of controls and is detailed in §2.3 below.

Listing 3 Specification of SBF insertion.

```
Definition INSERT_spec : func_spec :=
  MOD None [p]
  WITH (k : key) (tstamp : Z) (f : filter),
    PRE (ARG [key_to_sval k; P4Bit 8 INSERT; P4Bit 48 tstamp; P4Bit 8 0]
          (MEM [ ]
          (EXT [SBF_repr p f])))
    POST (ARG_RET [P4Bit 8 0] ValBaseNull
           (MEM [ ]
           (EXT [SBF_repr p (SBFinsert f (tstamp, k))]))).
```

To prove that the insertion operation of the SBF control satisfies INSERT_spec, our tool creates a symbolic state in Coq as described by the precondition. The user then steps through the P4 code, applying forward-mode Hoare rules from our program logic; when reaching the end of the control block one proves that the resulting symbolic state implies the postcondition. The proof proceeds mostly automatically, but in some places the user will have to direct it what to do.

▶ **Theorem 2** (Coq). *The SBF control block of our P4 program correctly implements the functional model (Listing 1).*

This is more than a *safety proof*: it proves not just "this implementation won't crash," it guarantees that the program really behaves like a lookup table with no false negatives.

2.3 Hierarchical State Assertions

According to the P4 language specification, each control or parser declaration is a class definition with local variable, object, and function members. The local variables are temporary for each call to the class, so they are considered as stack variables. Objects include tables, external objects, and control/parser instances. Stateful external objects (e.g., registers) have a persistent state that is preserved between packets.

In P4_{16}, controls and parsers are *instantiated* recursively. Each object has a global fully qualified name, generated by appending its local name to the global name of the parent object. Thus, instantiation forms a tree.

We exploit this hierarchy during specification and proof to make it easy to associate *abstract data types* to P4 state. We encapsulate all objects instantiated inside a root object in a single representation predicate, so that a change in the root object's implementation does not affect client verification. For this purpose, we define predicates along the structure of the hierarchical state, associating each predicate with a path-prefix containing an abstraction of the path names of all sub-objects.

The simplest assertion is of the form $p \mapsto r$, which means the register whose global name is p has value r. The path prefix of this assertion is $\{p\}$. A control block may have multiple registers. For assertions $P_1, \ldots, P_n$ with path prefixes $S_1, \ldots, S_n$, the path prefix of the assertion $P_1 \wedge \cdots \wedge P_n$ is $\bigcup_i S_i$. To encapsulate the state of an object rooted at p, we want to say an assertion has path prefix at most $\{p\}$, without mentioning its contents. So we define a "wrap" operator that wraps an assertion with a more abstract path prefix. Formally, for assertion P whose path prefix is S_1, $\text{wrap}(S_2, P)$ is a valid assertion if $S_1 \sqsubseteq S_2$, that is, every mapping in S_1 is also in S_2. A path p "covers" all of its subtrees, for example $\{p.x\} \sqsubseteq \{p\}$.

The assertions for an SBF containing k panes with r rows each are defined as:

$$\begin{aligned}
\texttt{row_repr}(p, \mathit{row}) &:= \text{wrap}([p], p.\texttt{reg} \mapsto \mathit{row}),\\
\texttt{pane_repr}(p, \mathit{pane}) &:= \text{wrap}([p],\ \texttt{row_repr}(p.\texttt{row1}, \mathit{pane}.\texttt{rows}[0]) \ \wedge\ \ldots\ \wedge\\
&\qquad\qquad \texttt{row_repr}(p.\texttt{rowR}, \mathit{pane}.\texttt{rows}[r-1])\\
\texttt{SBF_repr}(p, \mathit{filter}) &:= \text{wrap}([p],\ \texttt{timer_repr}(p.\texttt{timer}, \mathit{filter}.\texttt{t}) \wedge\\
&\qquad\qquad \texttt{pane_repr}(p.\texttt{pane1}, p.\texttt{panes}[0]) \ \wedge\ \ldots\ \wedge\\
&\qquad\qquad \texttt{pane_repr}(p.\texttt{paneK}, p.\texttt{panes}[k-1])).
\end{aligned}$$

This concludes the description of our example.

3 How the verifier works

Our verifier contains a verification-oriented P4 representation with a parser, an operational semantics, a program logic, and automation tactics.

3.1 P4light abstract syntax, front end

When designing our intermediate language, *P4light*, our desiderata were to

1. include a type-annotation at every expression node;
2. fully disambiguate names, i.e. distinguish local versus global variables;
3. avoid side-effect expressions inside subexpressions;[2]
4. and yet, stay sufficiently close to source-level P4 so that every P4light program can be pretty-printed as a legal and compilable P4 program.

Front end. We adapted Petr4's front end [5] (including its type-checker) to produce P4light ASTs from P4 source programs.

Hierarchical name space. To distinguish names in different scopes, we decorate names with *locator* annotations. A name has locator *glob* p if it is defined in the global scope, or locator *inst* p if it is defined inside a parser/control declaration. In either case, p is a qualified name (*path*) such as `myIngress.x`, so objects are uniquely named. Qualified names can further expand into fully qualified names in the instantiation phase (§4).

Unnesting expressions. To simplify reasoning about programs in P4light, we hoist expressions out of function calls, adding extra local variables in the process.

Representation. Our ASTs are expressed using Coq's inductive data types, with one type for each syntax class of the AST grammar (expression, statement, function, etc.).

3.2 Operational semantics

We define P4 execution as a big-step operational semantics that operates over states, defined as pairs of a stack frame and an external state. A stack frame (resp. external state) is a partial mapping from paths (i.e. fully qualified names) to values (resp. external object values).

$$\begin{aligned}
\texttt{StackFrame} &:= \texttt{Path} \rightharpoonup \texttt{Value}\\
\texttt{ExternState} &:= \texttt{Path} \rightharpoonup \texttt{ExternObject}\\
\texttt{State} &:= \texttt{StackFrame} \times \texttt{ExternState}
\end{aligned}$$

[2] This point simplifies the development of operational semantics and program logic, and improves the interaction experience of our verifier. But P4 always evaluates expressions from left to right, not like C, whose evaluation order is unspecified. So this transformation does not add restrictions to the semantics.

Only the external state persists from one packet to another. Because P4 does not have explicit pointers, we don't need to mention memory addresses, only paths.

Let Γ be the global static environment produced in §4 below. As P4 program statements are mostly inside controls and parsers, we need to know where a statement is in order to execute it. We use a path p to indicate the path of the object that the program is currently in (p is an empty path if not in any object). Let s be a state. We write our big-step semantic judgments as

$$\Gamma, p, s \vdash e \Downarrow v \qquad \text{(expression, 18 rules)}$$
$$\Gamma, p, s \vdash e \Downarrow lv \qquad \text{(l-expression, 5 rules)}$$
$$\Gamma, p, s \vdash stmt \Downarrow (s', sig) \qquad \text{(statement, 16 rules)}$$
$$\Gamma, p, s \vdash e \Downarrow (s', sig) \qquad \text{(call-expression, 2 rules)}$$
$$\Gamma, p, s \vdash f, a_{\text{in}} \Downarrow (s', a_{\text{out}}, sig) \qquad \text{(function, 3 rules)}$$

For example, the judgment $\Gamma, p, s \vdash e \Downarrow v$ reads as "in global environment Γ, with object path p, in state s, the P4light expression e evaluates to value v." Judgment $\Gamma, p, s \vdash \text{stmt} \Downarrow (s', \text{sig})$ reads as "for Γ and p, from state s, the execution of the P4 statement stmt results in state s' and signal sig." Signal is used to mark control flow, like return and exit statements.

As usual in operational semantics, each of these judgments is an inductive relation. If no rule applies, the operational semantics is *stuck*, a technical representation of *undefined behavior*. P4 is designed so that programs that type-check cannot have undefined behavior – a formal proof that this is indeed the case is under current development.

The operational semantics of a control block is given by that of its variable initialization, followed by that of its apply function. More details on the operational semantics presented in the forthcoming PhD thesis of the first author [25].

We briefly discuss two differences between P4 and more traditional languages, to illustrate the challenges we faced.

Undefined Values. According to the P4 specification, reading an uninitialized field or an invalid header yields an undefined value. As of 2021 the official description in the P4_{16} reference manual was ambiguous, but the P4 committee clarified that each bit of such a field can be either 0, 1, or uninitialized. To characterize this in assertions and our tool's symbolic execution, we use an abstract interpretation over bits. The abstract domain for an n-bit field is $\{0, 1, \bot\}^n$, where 0 and 1 characterize the two fully determined values and $\bot$ means the bit's value can be arbitrary, including undefined.

P4 data structures (headers, structs, bitfields) may be partially uninitialized, but P4 expressions and subexpressions are fully defined. That means, when reading from a data structure the semantics must choose arbitrary 0s and 1s for the undefined bits, so we use a *havoc* construct at the appropriate places.[3] Exactly when and how this happens was unclear in the P4_{16} document, but in discussions with the committee we clarified the document and formalized these clarifications in our semantics.

Our treatment of partially defined values in our operational semantics carries over to the logic and verification tool. Other verification tools for P4 do not treat this exactly; they either assume that all uninitialized bits are 0 (which is unsound) or cannot reason about uninitialized fields at all (which is incomplete).

[3] "Havoc" is a standard term in operational-semantic reasoning to indicate arbitrary behavior where it is not necessary to know which choice a program will make.

Compiler-rejected programs. In many languages, if a program is legal then you can expect that the compiler will compile it. In P4 that's explicitly not the case: a legal P4 program may violate architecture-specific pipeline and resource constraints, and be rejected by the compiler. For this reason, our operational semantics (and verifier) does not model architecture-specific constraints, so one should really read our soundness theorem as, "If your compiler agrees to compile the program *and* you prove some correctness property in *Verifiable P4*, then your program will indeed respect that property."[4]

Validating the operational semantics, and debugging P4

We validated that the operational semantics accurately captures the behavior of real compilers and hardware (e.g., Tofino, V1model) using three approaches: correspondence to the $P4_{16}$ reference document; correctness of a P4 reference interpreter w.r.t. our semantics; and testing against Tofino. We comment on the former two activities. A fourth approach, validating our reference interpreter against existing P4 test suites, is ongoing work.

Comparing to the $P4_{16}$ reference manual. We claim that our formalization specifies the same as what the official P4 standard means and what the commercial compilers do. Our process has been to rigorously formalize what is written down informally in the specification; when we find ambiguities, errors, or disagreements between $P4_{16}$ and commercial compilers, we discuss those with the P4 committee – so the official English-language standard gets refined, or bugs get fixed in the commercial compilers. See Table 1. By the time our process has finished, there is real evidence that the formal semantics has meaningful utility, and agrees with the $P4_{16}$ specification.

3.3 Reference Interpreter

In order to execute P4 programs and test our semantics, we wrote a reference interpreter in Coq's embedded functional language. We proved correctness of the interpreter with respect to our operational semantics and used extraction to obtain an executable OCaml program.

The main task in programming the interpreter was translating inductive relations (from the big-step semantics) to Coq functions. Many of the inductive relations are nondeterministic, so the interpreter determinizes them uniformly. Wherever an undefined value is involved in a computation, the interpreter initializes it using zero bits. This is sound, but means that some behaviors exhibited by the big-step semantics cannot be observed in the interpreter.

The interpreter shares some code with the operational semantics, which uses functions for many important (but deterministic) subroutines. For instance, instantiation (see §4) is handled by a functional program in the operational semantics. The interpreter reuses this program, preserving the phase distinction between instantiation and evaluation.

P4 allows "architectural extensions" such as V1model registers or Tofino's registers (which are different from each other). Our semantics handles those extensions as a kind of plug-in. Our verification tool and the reference interpreter treat the extensions by directly using this plug-in, so the reference interpreter can serve for core P4, for V1model P4, or for Tofino P4. Our example (the stateful firewall) happens to be a Tofino P4 program.

[4] And if your compiler is correct, then your program *as compiled* will respect the property too; our formalization of the operational semantics should also support the development of compiler proofs.

Table 1 Specification issues clarified during our operational-semantic formalization.

Section refers to section numbers in the $P4_{16}$ manual [3]. *Git* refers to our issue reports in the repo for that manual, `https://github.com/p4lang/p4-spec`. *Status* "pending" means under discussion with the steering committee; "released" means that a published version of the manual incorporates our merged pull-request. As indicated, some of these issues also reflect bugs in the p4c compiler, some of which have now been corrected.

	Section	Issue	Git	Status	p4c Bug
Expression	8.5, 8.6	Types of the bit slicing index are vague.	955	Released	No
	8.5	Concatenation is missing from the operations on the bit type.	956	Released	No
	8.5, 8.6	Concatenation is not excluded from the binary operations that require same-type operands.	956	Released	No
	8.9.2	Concatenation and shift are not excluded from the binary operations that allow implicit casts.	957	Released	No
	8.9.2	Unclear where implicit casts of serializable enum are allowed.	958	Released	Yes
	8.7	Right operand types of integer shifts are vague.	959	Released	No
	8.10-12 8.14-15	Allowed comparisons between lists, tuples, structs, and headers are unexplained.	960	Pending	Yes
	8.11, 8.12	Implicit casts between lists, tuples, structs, and headers are unexplained.	953	Pending	No
	8.10	Explicit casts between non-base types are unexplained.	961	Released	No
	8.7	Slicing integers is not allowed.	1015	Pending	No
	8.22	Reading uninitialized values is confusing and vaguely defined during argument passing.	988	Pending	Maybe
	8.13	Types of set operations are vague.	969	Pending	Maybe
Function	10.3.1	Abstract extern methods open multiple back doors, e.g., allowing recursion and invoking parsers in controls.	973, 976, 979	Pending	Maybe
	App. F	Parameter restrictions for extern functions are missing.	972	Released	No
	6.7.2	Optional parameters are not allowed in parser and control types.	977	Released	Maybe
Instantiation	11.3,A.H	Instantiation should not be a statement.	975	Released	Yes
	17.2	The concept of instantiation-time known constants is missing.	932	Pending	Maybe
	12.10, 13.4	The difference between stateful and stateless instantiations in parsers and controls is poorly explained.	926	Pending	No
Table	13.2	Default action is not set as NoAction when undefined.	933	Released	No
	13.1	The possible sources of action data are defined in a misleading way.	914	Released	No
Name	17.3	Value sets are not included in control plane objects.	962	Released	No
	6.8	Name duplication and name shadowing is undefined.	974	Pending	Maybe
	6.4	Inconsistent name style for built-in methods & fields.	1004	Pending	Yes

3.4 Program Logic and Proof Automation

We have designed a program logic for P4 (except for packet parsing[5]), as a set of proof rules that have been proven sound in Coq with respect to our operational semantics. As all P4 programs are terminating, the distinction between partial and total correctness vanishes, but P4's hardware-orientation, and the goal to not be overly specific to a concrete architectural model make the logic challenging to design. Concrete challenges arose in the treatment of headers, from the fact that the category of values includes `structs`, and from the combination of hierarchical path names and instantiation. We do not show the proof rules in this paper, and also omit a detailed discussion of automation support that uses Coq's tactic language *Ltac*. As explained in the previous section for the firewall example, the typical user must provide data representation predicates (how the abstract values of interest are represented in the data structures of P4 programs), function specification (preconditions, postconditions), specifications of control blocks (i.e., pre/postconditions of apply functions), and specifications of registers or other architecture-specific externals.

Our logic is modular in the sense that for any control block definition in the code we can write a single proof script that can be adapted to multiple instantiations. Thus, a library module such as our SBF can be proven to satisfy its specification (set-membership check with no false negatives) quite independently of the correctness proof of its client.

▶ **Theorem 3** (Coq). *(Soundness) The proof rules of our program logics are sound w.r.t. the operational judgments in §3.2.*

Soundness of our verifier then follows, as the verifier's tactics build a machine-checked proof using the logic's inference rules.

4 Instantiation

In accordance with Sec. 18 of the P4 specification [3], we divide evaluation into two phases: instantiation and execution. In the instantiation phase, constructor calls are evaluated to produce a static environment of *objects* (instances of control blocks, tables, parsers, packages, and external objects). This instantiation phase determines all relationships between objects and thus avoids any kind of dynamic allocation or closure passing. Indeed, instantiation more closely resembles creation of statically allocated objects/structs in C-like languages and instantiation of Verilog modules than dynamic object allocation.

The instantiation phase assigns a distinct *fully qualified name* to each object. For tables and external objects, it is convenient that this name is the same as the *control plane name* described in the $P4_{16}$ reference manual, which can be used by the control plane to access the objects, *e.g.* updating table entries and reading registers. Then for each object, its name is bound to its record of code and references to generate the static environment. The static environment is passed later into the execution phase, where it is read from but never modified. This is only possible because P4 is designed to avoid dynamic allocation.

In contrast, Petr4 [5] combines the two phases, using closures and dynamic allocation in the evaluation semantics. As a consequence, Petr4 requires a more complex state model with stores (mapping locations to values) and environments (mapping names to locations).

The (simplified) pseudocode of the instantiation phase is in Listing 4. Function `instantiate_prog` iterates over all declarations in the program. For control and parser declarations, it puts them in `decl_env` to look them up when encountering their instances.

[5] Verification of P4 parsing is the subject of a separate project [6], with which we expect to connect.

Listing 4 Pseudocode of instantiation.

```
global ge := ...
global decl_env := ...

procedure instantiate(p, decl) :=
    inst_name := instance name of decl
    class_name := class name of decl
    body := decl_env[class_name]
    ge := ge[p.inst_name ↦ record of the instance]
    for each decl in body
        if decl is an instantiation then instantiate(p.inst_name, decl)

procedure instantiate_prog(prog) :=
    for each decl in prog
        if decl is an instantiable declaration then
            decl_env := decl_env[(name of decl) ↦ decl]
        else if decl is an instantiation then instantiate(ϵ, decl)
```

For instantiation declarations, it calls `instantiate` with the empty path ϵ, which indicates to instantiate at the top level. `instantiate` takes a path `p` and an instance declaration `decl` to instantiate `decl` that appears in `p`. Let `inst_name` be the local name of the instance of `decl`. Then its fully qualified name is `p.inst_name`. `instantiate` allocates the object at `p.inst_name`, and then allocates its inner declarations recursively.

This procedure allocates the fully qualified names and constructs the global static environment at the same time. Inner declarations are allocated under the path `p` and they have distinct local names, so their global fully qualified names are distinct and different from names from other parts of the program. It also provides a local view under path `p`.

We have implemented the instantiation phase as a function in Coq. Users can just use Coq's computation mechanism to evaluate P4 programs.

4.1 Abstract Methods

Suppose you want to increment a (persistent) register on a Tofino P4 switch. You might read the register into a local variable, add one, and write back to the register. But that violates a Tofino pipeline constraint: accessing the same register in two different pipeline stages.

The $P4_{16}$ spec permits architectural extensions including stateful objects on which the P4 program may invoke so-called *abstract methods*. Tofino's registers are such extern objects, and provide a read-modify-write abstract method: the user specifies what *modify* to perform (like incrementing a value, as in our SBF insertion), but not the surrounding register read/write operations – these are implicitly performed by Tofino's invocation mechanism. Without abstract methods, Tofino registers are not much use, but their "almost-object-oriented" realization is arguably even more complex than function pointers in C or virtual or abstract methods in C++ or Java.

Our operational semantics is the first formal specification of P4 to include abstract methods, and *Verifiable P4* is the first verifier to support them.

5 Proof statistics

Our verification of the sliding-window Bloom filter (SBF) is summarized in Table 2. The P4 proof of "Filter" is so large in part because the P4 program has many branches about which we need to reason. The high-level "Filter" proof is large for a different reason: in part because the correctness argument for sliding-window Bloom filters is not trivial; and because we use two intermediate models to facilitate reasoning, which may not have been the best way to organize that proof.

Table 2 Lines of P4 code, specifications, Coq proofs. For each module of the P4 program we have (in the order shown), a functional model, Verifiable P4 function-specifications with preconditions and postconditions, a Coq proof script showing that the P4 code implements the model, and a proof that the model satisfies the high-level specification.

	P4 code	Func. Model	Funcspecs	P4 proof	High-level proof
Row	53	85	165	140	106
Pane	22	62	235	140	87
Filter	341	333	858	1579	1068
Total	416	480	1258	1859	1261

6 Count-min-sketch

We also verify count-min-sketch (CMS) [4] to illustrate that the existing proof script can be easily adapted to verify similar data structures. A CMS is essentially the same as a counting Bloom filter, which counts the frequency of the different types of events in the stream.

Most parameters of a simple CMS data structure are the same as a simple Bloom filter, such as r rows and S slots. The difference is that each slot now indicates how many times an event occurred rather than whether it occurred at all. The insert operation thus increments the relevant slots, *saturating* at `Int<k>.max` if each slot is k bits wide. The query operation returns the minimum value of the matching slots in all rows.

Our CMS implementation employs the same sliding window mechanism as the SBF, and similarly for the functional model. The axiomatization employs the same predicate `ok_until` and reuses most axioms, except for the two axioms shown in Listing 5. From these axioms one can derive the analogue of the Bloom filter's "no false negatives", that CMSquery returns a number not less than the true count. In the function specification (not shown here but analagous to Listing 3) one can see that the value returned is the minimum of this count and the saturation value `Int<k>.max`.

Listing 5 Axioms of Count-Min-Sketch (excerpt).

```
QueryInsertSame : ∀f t t' h k, ok_until f t → t ≤ t'≤ t+T →
     CMSquery f (t', h) = Some k →
     CMSquery (CMSinsert f (t, h)) (t', h) = Some (k + 1).

QueryInsertOther : ∀f t t' h h' k, ok_until f t → t ≤ t' →
     CMSquery f (t', h') = Some k →
     CMSquery (CMSinsert f (t, h)) (t', h') = Some k ∨
     CMSquery (CMSinsert f (t, h)) (t', h') = Some (k + 1).
```

7 Discussion

Related work. Vera [23], p4v [15], and ASSERT-P4 [18] are automatic P4 verifiers for simple properties – including safety properties, architectural constraints, and simple stateless application properties. They translate the P4 program into a guarded command language (p4v), into a network verification language called SEFL (Vera), or into C (ASSERT-P4). ASSERT-P4 does not claim to support stateful objects. Vera [23] uses an extremely expensive encoding that is proportional to the size of registers (impractical when the register contains an entire hash table). Although p4v supports stateful objects, we cannot find any evidence that p4v can relate initial and final states.

Aquila [24] supports a more convenient assertion language, multi-pipeline control, more time-efficient verification, and bug localization when the verification claims a bug. But Aquila oversimplifies registers into fields without indexes, so it could not verify programs such as the stateful firewall. Aquila also reduces the risk of bugs in the verifier by translation validation – checking whether its intermediate representation is equivalent to the counterpart generated by Gauntlet [22] (which is a tool for finding bugs in P4 compilers). But that does not address other software bugs, e.g. the bugs in manipulating assertions, especially when we need a rich and modular assertion language.

Π4 [8] is a dependent refinement type system for a language similar to a subset of P4. Although described as a type system, it includes assertions in types and requires an SMT solver to check the types, so it is very similar to a verifier with assertions in middle of programs. Of all these verifiers, it is the only one that supports modular specification and verification function-by-function – its dependent function type is equivalent to a function contract. But Π4 expands the instantiation hierarchy before verifying, but does not exploit the similarity of the resulting instances. Π4 does not support stateful objects.

There are some other tools that focus on particular properties. bf4 [7] is a tool that automatically infers constraints of tables to make a P4 program safe. Leapfrog [6] is a parser equivalence checker for P4.

Petr4 [5] is a study of P4's formal semantics, which gives us an important reference. But Petr4's semantics does not have a machine-checkable formalization, and mixes instantiation and execution, which means it "instantiates at runtime" and has to define each control instance as a closure. This makes the semantics less straightforward and makes it much more challenging to prove the program logic and the type system sound. We improve this with a phase distinction between instantiation and execution (§4). We also identify and fix some bugs in Petr4 during the formalization in Coq.

The approach taken by Vigor [28, 20, 27] is conceptually similar to our envisioned separation into verification of reusable data structures and more abstract packet- or flow-level network functionality. Vigor is a verifier for C programs on top of DPDK; it employs Verifast [11] for the reusable data structures and the Klee symbolic execution engine [1] for the C code that is a client of these data structures. Vigor has been applied to a number of data structures and network functions, with more automation than our framework currently supports. We expect that as top-level policies get more complex, the expressivity of a general-purpose proof assistant (that we use) will prove beneficial. The use of ADT operations as the interface between the two parts of a Vigor proof is conceptually similar to our use of functional or axiomatic models, but – unlike in our approach – there is no machine-checkable proof that connects these layers together. Continuing this line of work, KLINT [19] applies to binary code, with a focus on the code that implements network functions on top of a small fixed set of map-like data structures whose implementations are assumed correct.

Gravel [29] verifies C++ implementations of Click-style middlebox functions against functional specifications (represented as Python programs), using symbolic execution on LLVM and an SMT solver backend. Like Vigor and KLINT, Gravel achieves a higher degree of automation than our work but has a larger trusted code base, does not support expressive reasoning about specifications, and does not apply to line-rate processing on a P4 switch.

Gopinathan and Sergey [9] show how to prove in Coq that a Bloom filter (functional model) has not only a zero false-negative rate but also a small false-positive rate. Our system could accommodate that style of proof: use Verifiable P4 to prove (as we do) that the Bloom filter exactly implements its functional model; then prove (as we did not) that this functional model has a small false-positive rate, using the method that they demonstrate.

Future work. We currently achieve modular verification by writing a proof script for one instance of a control block (such as *pane* or *row*) that can be re-used unchanged for all the other instances. In future work, we expect to make this more formal, to *guarantee* that each control needs only one proof of correctness.

To support switches with multiple pipelines (each with its own persistent register state), we may either modify the program logic, or treat the independent pipelines as separate switches and account for concurrency at the model level, when proving the relation between the single-packet correctness property and the flow-level property.

Some stateful applications mentioned in Section 1 involve packet recirculation, but we have not explored this in detail, especially the issue of buffering synthesized packets.

Finally, a long-term goal may be to augment tools that synthesize P4 code with mechanisms to generate *Verifiable P4* specifications, or even proofs. For example, we use CatQL to synthesize our bloom filter programs but have no tools to derive specifications or proofs for the synthesized P4 code from CatQL directly.

Our formalization assumes that each P4 packet is handled atomically, which is a justifiable assumption for single Tofino-like pipelines.[6] But Tofino (and other P4-programmable switches) may have multiple parallel pipelines with independent register sets, and multiple sequential pipelines (i.e., ingress and egress pipelines) with nontrivial scheduling between them. Reasoning about packet management *outside* of P4 is future work – though it may not require any changes to our P4 semantics itself.

Conclusion. P4-programmable switches enable a new generation of applications that use sophisticated data structures, modular software engineering, and persistent state. The constraints of P4-capable hardware often mean that those programs are written in a somewhat contorted way that makes them difficult to reason about. Therefore, verification tools are even more useful for P4 than they are for conventional languages.

We have built a verifier for P4 and demonstrated it on a sophisticated application. Our verifier is the first one that can handle rich specifications or nontrivial uses of persistent state. It is the first to be proved sound with respect to a formal semantics of P4. And our formal semantics is the first one that attempts to be complete with respect to the English-language $P4_{16}$ reference manual.

[6] But not a *trivial* assumption: a multistage pipeline may have stateful registers at different pipeline stages, and we can treat the whole pipeline atomically only because (1) each register is accessed only at a given pipeline stage by *all* packets that access it and (2) packets cannot overtake each other within the pipeline.

References

1 Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008. URL: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.

2 Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, pages 226–239, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3387514.3405865.

3 The P4 Language Consortium. $P4_{16}$ language specification, version 1.2.3, July 2022. URL: https://p4.org/p4-spec/docs/P4-16-v1.2.3.pdf.

4 Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005. doi:10.1016/j.jalgor.2003.12.001.

5 Ryan Doenges, Mina Tahmasbi Arashloo, Santiago Bautista, Alexander Chang, Newton Ni, Samwise Parkinson, Rudy Peterson, Alaia Solko-Breslin, Amanda Xu, and Nate Foster. Petr4: Formal foundations for P4 data planes. *Proc. ACM Program. Lang.*, 5(POPL):1–32, 2021. doi:10.1145/3434322.

6 Ryan Doenges, Tobias Kappé, John Sarracino, Nate Foster, and Greg Morrisett. Leapfrog: certified equivalence for protocol parsers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 950–965. ACM, 2022.

7 Dragos Dumitrescu, Radu Stoenescu, Lorina Negreanu, and Costin Raiciu. bf4: towards bug-free P4 programs. In Henning Schulzrinne and Vishal Misra, editors, *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 571–585. ACM, 2020. doi:10.1145/3387514.3405888.

8 Matthias Eichholz, Eric Hayden Campbell, Matthias Krebs, Nate Foster, and Mira Mezini. Dependently-typed data plane programming. *Proc. ACM Program. Lang.*, 6(POPL):40:1–40:28, 2022. doi:10.1145/3498701.

9 Kiran Gopinathan and Ilya Sergey. Certifying certainty and uncertainty in approximate membership query structures. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Proceedings, Part II*, volume 12225 of *LNCS*, pages 279–303. Springer, 2020. doi:10.1007/978-3-030-53291-8_16.

10 Intel. Intel® Tofino™ programmable ethernet switch ASIC. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html. Accessed: 2023-01-18.

11 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 Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Proceedings*, volume 6617 of *LNCS*, pages 41–55. Springer, 2011. doi:10.1007/978-3-642-20398-5_4.

12 Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 121–136. ACM, 2017. doi:10.1145/3132747.3132764.

13 Patrick Kennedy. Intel Tofino2 next-gen programmable switch detailed. https://www.servethehome.com/intel-tofino2-next-gen-programmable-switch-detailed/, August 2020.

14 Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 311–324. USENIX Association, 2016.

15 Jed Liu, William T. Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Calin Cascaval, Nick McKeown, and Nate Foster. p4v: Practical verification for programmable data planes. In Sergey Gorinsky and János Tapolcai, editors, *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM 2018)*, pages 490–503. ACM, 2018. doi:10.1145/3230543.3230582.

16 Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 15–28. ACM, 2017. doi:10.1145/3098822.3098824.

17 Hun Namkung, Zaoxing Liu, Daehyeok Kim, Vyas Sekar, and Peter Steenkiste. SketchLib: Enabling efficient sketch-based monitoring on programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 743–759, Renton, WA, April 2022. USENIX Association. URL: https://www.usenix.org/conference/nsdi22/presentation/namkung.

18 Miguel Neves, Lucas Freire, Alberto Schaeffer-Filho, and Marinho Barcellos. Verification of P4 programs in feasible time using assertions. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, pages 73–85. ACM, 2018.

19 Solal Pirelli, Akvile Valentukonyte, Katerina J. Argyraki, and George Candea. Automated verification of network function binaries. In Amar Phanishayee and Vyas Sekar, editors, *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, pages 585–600. USENIX Association, 2022. URL: https://www.usenix.org/conference/nsdi22/presentation/pirelli.

20 Solal Pirelli, Arseniy Zaostrovnykh, and George Candea. A formally verified NAT stack. *Comput. Commun. Rev.*, 48(5):77–83, 2018. doi:10.1145/3310165.3310176.

21 John Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2002: IEEE Symposium on Logic in Computer Science*, pages 55–74, July 2002.

22 Fabian Ruffy, Tao Wang, and Anirudh Sivaraman. Gauntlet: Finding bugs in compilers for programmable packet processing. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 683–699, 2020.

23 Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging P4 programs with Vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 518–532, 2018.

24 Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, et al. Aquila: a practically usable verification system for production-scale programmable data planes. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 17–32, 2021.

25 Qinshi Wang. *Foundationally Verified Data Plane Programming.* PhD thesis, Princeton University, 2023. In preparation.

26 Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. Netlock: Fast, centralized lock management using programmable switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, pages 126–138, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3387514.3405857.

27 Arseniy Zaostrovnykh, Solal Pirelli, Rishabh R. Iyer, Matteo Rizzo, Luis Pedrosa, Katerina J. Argyraki, and George Candea. Verifying software network functions with no verification

expertise. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 275–290. ACM, 2019. doi:10.1145/3341301.3359647.

28 Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina J. Argyraki, and George Candea. A formally verified NAT. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 141–154. ACM, 2017. doi:10.1145/3098822.3098833.

29 Kaiyuan Zhang, Danyang Zhuo, Aditya Akella, Arvind Krishnamurthy, and Xi Wang. Automated verification of customizable middlebox properties with gravel. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 221–239. USENIX Association, 2020. URL: https://www.usenix.org/conference/nsdi20/presentation/zhang-kaiyuan.

30 Linfeng Zhang and Yong Guan. Detecting click fraud in pay-per-click streams of online advertising networks. *2008 The 28th International Conference on Distributed Computing Systems*, pages 77–84, 2008.

Dependently Sorted Theorem Proving for Mathematical Foundations

Yiming Xu
Australian National University, Canberra, Australia

Michael Norrish
Australian National University, Canberra, Australia

Abstract

We describe a new meta-logical system for mechanising foundations of mathematics. Using dependent sorts and first order logic, our system (implemented as an LCF-style theorem-prover) improves on the state-of-the-art by providing efficient type-checking, convenient automatic rewriting and interactive proof support. We assess our implementation by axiomatising Lawvere's Elementary Theory of the Category of Sets (ETCS) [5], and Shulman's Sets, Elements and Relations (SEAR) [17]. We then demonstrate our system's ability to perform some basic mathematical constructions such as quotienting, induction and coinduction by constructing integers, lists and colists. We also compare with some existing work on modal model theory done in HOL4 [20]. Using the analogue of type-quantification, we are able to prove a theorem that this earlier work could not. Finally, we show that SEAR can construct sets that are larger than any finite iteration of the power set operation. This shows that SEAR, unlike HOL, can construct sets beyond $V_{\omega+\omega}$.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases first order logic, sorts, structural set theory, mechanised mathematics, foundation of mathematics, category theory

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.33

Supplementary Material *Other (Source Code)*: `https://github.com/u5943321/DiaToM`
archived at `swh:1:dir:4358f11ed6de22aabb16cfe4b51c769fb9fd6e1d`

Acknowledgements We appreciate all the anonymous reviewers for their time and effort in pointing out typos and suggesting improvements to this paper. We would also like to thank James Borger for the name "DiaToM", which we feel beautifully encapsulates the aims and nature of our project.

1 Introduction

Mathematicians claim to work with set theory all the time, but many do so without really having to, or trying to, grapple with set theory's axioms. Moreover, this attitude is not unreasonable: it is not clear that standard ZF set theory should be mathematicians' foundation of choice. Few people are particularly happy with a foundation insisting that, for example, $1 \in 2$. It is not surprising then that a number of different foundations have been proposed in the literature. Considering variants of set theory, some famous examples are Lawvere's ETCS [5], Shulman's SEAR [17], Quine's New Foundation [14], Tarski-Grothendieck set theory [18] and von Neumann–Bernays–Gödel set theory (see, for example, Mendelson's presentation [11]) . Category theory has also been proposed as a mathematical foundation, in McLarty's CCAF [8] and Lawvere's ETCC [6], with the former having been shown capable of capturing many non-trivial results. And, though ETCC is known to be flawed, people have never lost interest in fixing it, and are continuing to work on similar systems.

Axiomatising a foundation for all of mathematics is a project that must be approached with the utmost care. Our belief is that this care should include mechanical support. That is, we should develop a theorem-proving system to serve as a tool for checking proofs in these foundations.


© Yiming Xu and Michael Norrish;
licensed under Creative Commons License CC-BY 4.0

Our second goal is *expressiveness*: we want our system to be flexible enough to capture a variety of systems. At the same time, it is readily apparent that a significant amount of work on mathematical foundations concentrates on first-order logic. Certainly, in all of the examples above, first-order logic is enough. We're quite happy to live with this constraint: a richer logic can conceal foundational decisions that we'd prefer to make apparent in our axioms. In the following, we present a first-order system that gains its expressiveness through a simple notion of dependent sort. Despite its simplicity, our system captures three of the foundational systems mentioned above, and is capable of fairly involved constructions in them all.

Contributions

- We develop a logical system that is able to express various first order axiomatic systems, where sorts can depend on terms. We specialise this ambient logical system so as to capture the foundational systems ETCS and SEAR.
- Building on these foundations, we demonstrate that our system can handle common mathematical constructions such as the development of the algebraic and co-algebraic lists.
- In one example, we also demonstrate SEAR's set-theoretic power by extending an existing example in model theory (done in HOL), and prove a theorem impossible to capture in HOL.
- We provide a proof-of-concept implementation that makes logical developments practical through the development of a number of important, though basic tools. For example, in ETCS, where proofs greatly rely on internal logic, we build a tool to automatically construct the internal logic predicates corresponding to "external" predicates. In both ETCS and SEAR, we automate inductive definitions, and provide tools to help with the construction of quotients.

The paper is structured as follows: we first introduce our fundamental logic, which is used for all three foundations. Then we briefly introduce the two structural set theories, ETCS and SEAR. After discussing the automation of comprehension in ETCS, for reasons of space, we present the remaining proofs in SEAR only. We note that with the exception of the modal model theory result (where the *bounded comprehension* schema is not sufficient) and the construction of the large set, all these formalised SEAR results can be formalised in ETCS as well. The proofs of a SEAR statement and its ETCS counterpart are often identical, in the sense that a proof in one system can be cut and pasted into the other. At the end of the paper, we compare our work with some existing logics developed for related purposes.

2 Logical System

We begin with the syntax of our logical system, which is "three-layered", consisting of sorts, terms, and formulas.

2.1 Sorts and Terms

Every sort depends on a (possibly empty) list of terms. The sorts are all of the form $s(t_0, \cdots, t_n)$, where $t_0, \cdots, t_n$ are terms of some pre-existing sorts and s is the name of the sort. A term is either a variable or a function application:

$$t := \mathsf{Var}(n, s) \mid \mathsf{Fun}(f, s, \vec{t})$$

That is, a variable consists of a name and a sort, and a function term consists of the name of the function symbol, the sort, and the arguments, which is a list of terms. A constant is a nullary function. Each term has a unique sort, carried as a piece of information as an intrinsic property. A sort which does not depend on any term is called a *ground sort*. A term with a ground sort is called a *ground term*.

2.2 Formulas

We are working with a classical logic, and can afford to be minimal with our syntax: a formula Φ is either falsity, a predicate, an implication, a universally quantified formula, or a formula variable.

$$\Phi \ ::= \ \bot \mid \mathsf{Pred}(\mathcal{P}, \vec{t}) \mid \phi_1 \implies \phi_2 \mid \forall n : s.\, \phi \mid \mathsf{fVar}(\mathcal{F}, \vec{s}, \vec{t})$$

In the above, $\mathcal{P}$ is a predicate name, and $\mathcal{F}$ is the name of a formula variable. Boolean operators $\wedge, \vee, \neg$ can hence be built from the implication. We write $\top$ as an abbreviation $\bot \implies \bot$. In the $\forall$ case, the n and s carry the name and sort of the quantified variable. A formula $\mathsf{Pred}(\mathcal{P}, \vec{t})$ is a concrete predicate symbol applied to the argument list $\vec{t}$. Such a predicate symbol is either primitive, which comes together with the axiomatic setting or is defined by the user. A formula variable $\mathsf{fVar}(\mathcal{F}, \vec{s}, \vec{t})$ is analogous to a higher-order lambda expression taking an argument list $\vec{t}$ with sorts $\vec{s}$. We provide an inference rule to instantiate them below in Section 2.3.1. In the following, we will write a predicate formula as $\mathcal{P}(\vec{t})$. For a formula variable with name $\mathcal{F}$ on arguments of sorts $\vec{s}$ applied on $\vec{t}$, we write $\mathcal{F}[\vec{s}](\vec{t})$.

The only primitive predicate embedded in the system is equality between terms of the same sort. However, we need not allow equalities between terms just because they have the same sort. We cannot, for example, write equality between objects in ETCS, or equality between sets in SEAR. Thus, each foundation must record (along with function symbols, predicates symbols and axioms), the list of sorts supporting equality.

2.3 Theorems

A theorem consists of a set of variables Γ, called the context, a finite set A of formulas (the assumptions), and a formula ϕ as the conclusion. A theorem $\Gamma, A \vdash \phi$ reads "for all assignments σ of variables in Γ to terms respecting their sorts, if all the formulas in $\sigma(A)$ hold, then we can conclude $\sigma(\phi)$".

The context is the set of variables we require for the conclusion to be true given the assumptions: it contains at least all the free variables appearing in the assumptions or the conclusion. It can be regarded as a special form of assumption, asserting the existence of terms of certain sorts. We need the context to make sure we cannot use terms before either constructing them or assuming their existence. For instance, there is no arrow from the terminal object 1 to the initial object 0 in either ETCS or SEAR. Using a context, it can be proved that: $\{f : 1 \to 0\} \vdash \exists f : 1 \to 0.\ \top$, but $\vdash \exists f : 1 \to 0.\ \top$ is easily proved to be false.

2.3.1 Proof System

We now introduce the primitive rules. Rules for the propositional connectives are standard, as in Figure 1. The quantifier rules take some extra care of the sort information. When specialising a universal by a term, we need to put all the free variables of such a term into the context. Let $\mathsf{Vars}(t)$ denote the set of variables occurring in the term t, then:

$$\forall\text{-E, } t \text{ is of sort } s \ \frac{\Gamma, A \vdash \forall x : s.\phi(x)}{\Gamma \cup \mathsf{Vars}(t), A \vdash \phi(t)}$$

$$\text{Assume}\ \frac{}{\mathsf{Vars}(\phi), \{\phi\} \vdash \phi} \qquad \text{Ax}\ \frac{}{\mathsf{Vars}(\phi) \vdash \phi}\ \phi \text{ is an axiom}$$

$$\text{CContr}\ \frac{\Gamma, A \cup \{\neg\phi\} \vdash \bot}{\Gamma, A \vdash \phi} \qquad \text{ExF}\ \frac{}{\mathsf{Vars}(A \cup \{\phi\}), A \cup \{\bot\} \vdash \phi}$$

$$\text{Disch}\ \frac{\Gamma, A \vdash \phi}{\Gamma \cup \mathsf{Vars}(\psi), A \setminus \{\psi\} \vdash \psi \implies \phi} \qquad \text{MP}\ \frac{\Gamma_1, A_1 \vdash \phi \implies \psi \qquad \Gamma_2, A_2 \vdash \phi}{\Gamma_1 \cup \Gamma_2, A_1 \cup A_2 \vdash \psi}$$

$$\text{Refl}\ \frac{}{\mathsf{Vars}(a) \vdash a = a} \qquad \text{Sym}\ \frac{\Gamma, A \vdash a = b}{\Gamma, A \vdash b = a} \qquad \text{Trans}\ \frac{\Gamma_1, A_1 \vdash a = b \qquad \Gamma_2, A_2 \vdash b = c}{\Gamma_1 \cup \Gamma_2, A_1 \cup A_2 \vdash a = c}$$

$$\text{InstTM}\ \frac{\Gamma, A \vdash \phi}{\sigma(\Gamma), \sigma(A) \vdash \sigma(\phi)}\ \sigma \text{ is a well-formed map}$$

$$\text{FVCong}\ \frac{\Gamma_1, A_1 \vdash t_1 = t_1', \cdots, \Gamma_n, A_n \vdash t_n = t_n'}{\bigcup_{i=1}^n \Gamma_i, \bigcup_{i=1}^n A_i \vdash \mathcal{F}[\vec{s}](\vec{t}) \Leftrightarrow \mathcal{F}[\vec{s}](\vec{t'})}$$

Figure 1 Natural Deduction style presentation of our sorted FOL.

To apply generalisation ($\forall$-I) with a variable $a : s(t_1, \cdots, t_n)$, we require that (i) a does not occur in the assumption set; (ii) there is no term in the context depending on a; (iii) all the variables of sort s must also be in $\Gamma \setminus \{x\}$, and (iv) a does not appear in the sort list of any formula variable appearing in the conclusion. Once all these conditions are satisfied, we have

$$\forall\text{-I}\ \frac{\Gamma, A \vdash \phi(x)}{\Gamma \setminus \{x : s\}, A \vdash \forall x : s.\ \phi(x)}$$

We define $(\exists x.\phi) = \neg(\forall x.\neg\phi)$. The instantiation rule for formula variables is given as:

$$\text{Form-Inst}\ \frac{\Gamma, A(\mathcal{F}[\vec{s}]) \vdash \phi(\mathcal{F}[\vec{s}])}{\Gamma \cup \mathsf{Vars}(\psi), A[\mathcal{F}[\vec{s}] \mapsto \psi] \vdash \phi[\mathcal{F}[\vec{s}] \mapsto \psi]}$$

Instantiating a formula variable $\mathcal{F}[\vec{s}]$ is to replace each occurrence of $\mathcal{F}[\vec{s}]$ into a concrete formula on an argument list with sorts $\vec{s}$, and apply this predicate on $\vec{t}$. This is done by providing a map sending each such formula variable to a formula. This formula may or may not contain more formula variables, and is encoded by a pair consisting of a variable list $v_1, \cdots, v_n$ of sort $\vec{s}$ and a formula ϕ, such that $\forall v_1, \cdots, v_n.\phi$ is a well-formed formula. We rely on the term instantiation rule to make changes to the sort list, and then instantiate the formula variable when required.

When defining a new foundation we assume the existence of a signature recording that foundation's sorts, function symbols and predicate symbols. We extend the signature with new predicate symbols using the predicate specification rule.

$$\text{Pred-spec}\ \frac{}{\mathsf{Vars}(\vec{t}) \vdash \mathcal{P}(\vec{t}) \Leftrightarrow \phi(\vec{t})}\ \mathcal{P} \text{ does not occur in } \phi$$

Applying such a rule will define a new predicate with the name $\mathcal{P}$. The defined predicate will be polymorphic, where each tuple whose sort is matchable with the list $\vec{t}$ can be taken as the arguments. Here the argument of the new predicate symbol is not required to be all of $\mathsf{Vars}(\phi)$, we only require the whole set of free variables involved can be recovered from the arguments. For instance, if $\{a_1 : s_1, a_2 : s_2(a_1)\}$ exhausts the free variables involved, then the predicate can just take the single argument a_2 instead of both a_1 and a_2.

2.3.1.1 Function specification rule

The specification rule for new function symbols is the most complicated. Given a theorem $\Gamma, A \vdash \exists a_1 : s_1, \cdots a_n : s_n.Q(a_1, \cdots, a_n)$, if the existence of the tuple $(a_1, \cdots, a_n)$ is unique up to any sense which is accepted as suitable by the foundation, we define function symbols $f_1, \cdots, f_n$ such that their output tuple satisfies Q. To define a new function symbol, we provide a theorem stating the unique existence of some terms up to some relation, a theorem stating the relation is an equivalence relation, and a theorem guaranteeing non-emptiness of the relevant sorts.

In general, an equivalence must be captured by a predicate on two lists of variables, representing the two entities being related. As the built-in logic does not have a notion of tuples, we cannot define an equivalence relation to be a subset of the set consisting pairs of tuples of a certain form. Instead, we require theorems of the form:

$$\vdash R(\langle a_1 : s_1, ..., a_n : s_n\rangle, \langle a_1 : s_1, ..., a_n : s_n\rangle)$$

$$\vdash R(\langle a_1 : s_1, ..., a_n : s_n\rangle, \langle a'_1 : s'_1, ..., a'_n : s'_n\rangle) \implies R(\langle a'_1 : s'_1, ..., a'_n : s'_n\rangle, \langle a_1 : s_1, ..., a_n : s_n\rangle)$$

$$\vdash R(\langle a^1_1 : s^1_1, ..., a^1_n : s^1_n\rangle, \langle a^2_1 : s^2_1, ..., a^2_n : s^2_n\rangle) \wedge R(\langle a^2_1 : s^2_1, ..., a^2_n : s^2_n\rangle, \langle a^3_1 : s^3_1, ..., a^3_n : s^3_n\rangle) \implies R(\langle a^1_1 : s^1_1, ..., a^1_n : s^1_n\rangle, \langle a^3_1 : s^3_1, ..., a^3_n : s^3_n\rangle)$$

If the three theorems all hold for a concrete property R, then R is an equivalence relation (abbreviated as $\mathsf{eqth}(R)$ in the rest of the discussion). If R is used as the equivalence relation above, the unique existential theorem is required to be of the form:

$$\exists a_i : s_i.\ Q(\langle a_1 : s_1, ..., a_n : s_n\rangle) \wedge \forall a'_i : s'_i.\ Q(\langle a'_1 : s'_1, ..., a'_n : s'_n\rangle) \implies R(\langle a_1 : s_1, ..., a_n : s_n\rangle, \langle a'_1 : s'_1, ..., a'_n : s'_n\rangle)$$

We abbreviate the formula above as $\exists!_R a_i : s_i.\ Q(\langle a_1 : s_1, ..., a_n : s_n\rangle)$. The sorts of the two argument lists are not required to be equal, and they are generally not equal because the sorts of the latter variables often depend on the previous ones. The rule is expressed as:

$$\frac{\Gamma_0, \emptyset \vdash \exists a_i : s_i.\top \qquad \Gamma', A' \vdash \mathsf{eqth}(R) \qquad \Gamma, A \vdash \exists!_R a_i : s_i.Q(\langle a_1 : s_1, ..., a_n : s_n\rangle)}{\Gamma, A \vdash Q(\langle f_1(\Gamma'), ..., f_n(\Gamma')\rangle)}$$

where

- Q and R do not contain any formula variables; and
- $\Gamma_0 \subseteq \Gamma$, $\Gamma' \subseteq \Gamma$, and $A' \subseteq A$.

Our rule's leftmost premise requires the existence of terms of the required (output) sorts, given the existence of variables in the context corresponding to the sorts of the arguments. In this way, the rule guarantees that terms built using the new function symbol will always denote values in the output sort. For the equivalence relation, we can take R to be equality, meaning we are specifying new function symbols according to unique existence. If we take R to be the everywhere-true relation we have imported the Axiom of Choice into our system. The choice of which R's to allow is up to the designer of the object logic.

2.3.2 Semantics *via* Translation to Sorted FOL

In work that is not further described here, we have mechanised the proof that formula variables and their proof rules represent a conservative extension and can be eliminated. Subsequently, the term-instantiation rule (InstTM in Figure 1) can be derived from ∀-I and ∀-E and can also be removed from the list of primitive rules. As a result, our semantics

below ignores them (meaning that our formulas come in just four forms: $\bot$, implications, the universal quantifier and predicate symbols). Our logic can be translated into non-dependent sorted FOL, which is equivalent to FOL. Given a list of sorts $s_1, \cdots, s_n$, such that s_k only depends on terms with sorts occurring earlier in the list for each $1 \leq k \leq n$, we create non-dependent sorts $\overline{s}_1, \cdots, \overline{s}_n$. These sorts are thought of as the non-dependent versions of $s_1, \cdots s_n$. We can think of the set of terms of sort $\overline{s_i}$ as the union of all terms of sort $s_i(\vec{t})$ for all possible tuples $\vec{t}$ of terms.

$$\{a \mid a : \overline{s_i}\} = \bigcup_{\vec{t_k}} \{a \mid a : s_i(\vec{t_k})\}$$

For example, the ETCS terms $f : A \to B$ and $g : C \to D$ are of different arrow sorts, but their translation both have sort $\overline{ar}$. For a function symbol f taking a list of terms $[t_1 : s_1, \cdots, t_n : s_n]$, we create a non-dependent sorted function symbol $\overline{f}$, such that its argument term list has the corresponding sort list $\overline{s_1}, \cdots, \overline{s_n}$. We do the same for predicate symbols. Translation from terms of s_k into those of FOL sort $\overline{s}_k$ is done by forgetting sort dependency:

$$\begin{aligned} [\![\mathsf{Var}(x, s_k(t_1, \cdots t_m))]\!]_{\mathsf{t}} &= \mathsf{Var}(x, \overline{s}_k) \\ [\![\mathsf{Fun}(f, s_k(t_1, \cdots, t_m), (a_1, \cdots, a_n))]\!]_{\mathsf{t}} &= \mathsf{Fun}(f, \overline{s}_k, ([\![a_1]\!]_{\mathsf{t}}, \cdots, [\![a_n]\!])) \end{aligned}$$

For sorts s depending on terms $t_1 : s_1, \cdots, t_m : s_m$, we create function constants $d_{s,1}$, $\cdots$, $d_{s,m}$. For $1 \leq i \leq m$, $d_{s,i}$ takes an argument of sort $\overline{s}$ and outputs a term of sort $\overline{s_i}$. If a function symbol f takes arguments $(t_1 : s_1, \cdots, t_n : s_n)$, and outputs a non-ground sort s, where s depends on terms $r_1, \cdots, r_n$, and each s_k depends on terms $q_{k,i}$, then we add an axiom to regulate the dependency information of its sort when translated into non-dependent-sort FOL:

$$\bigwedge_k \bigwedge_i d_{s_k,i}([\![v_k]\!]_{\mathsf{t}}) = [\![q_{k,i}]\!]_{\mathsf{t}} \implies \bigwedge_j d_{s,j}([\![f(v_1, \cdots, v_n)]\!]_{\mathsf{t}}) = [\![r_j]\!]_{\mathsf{t}}$$

As an example, the composition function symbol in ETCS takes $g : B \to C$ and $f : A \to B$, and outputs $g \circ f : A \to C$. The corresponding axiom is:

$$\begin{aligned} &\forall (A : \overline{\mathsf{ob}})\ (B : \overline{\mathsf{ob}})\ (C : \overline{\mathsf{ob}})\ (f : \overline{\mathsf{ar}})\ (g : \overline{\mathsf{ar}}). \\ &\quad d_{\mathsf{ar},1}(f) = A \wedge d_{\mathsf{ar},2}(f) = B \wedge d_{\mathsf{ar},1}(g) = B \wedge d_{\mathsf{ar},2}(g) = C \implies \\ &\qquad d_{\mathsf{ar},1}(g \circ f) = A \wedge d_{\mathsf{ar},2}(g \circ f) = C \end{aligned}$$

For an arbitrary function symbol f, although its arguments can include ground terms, the axiom only needs to state information about the dependently sorted argument, where the functions d_k, as shown above, exist. If the output of a function symbol is a ground sort, we do not need such an axiom for it.

Translation of formulas only makes sense under the translation of some context that contains at least all of its free variables. Defining the translation of a context amounts to translating sort judgments of variables. We translate the sort judgment of any ground sort into $\top$. As for a variable $a : s_k(t_1, \cdots, t_n)$, we write

$$[\![a : s_k(t_1, \cdots, t_n)]\!]_{\mathsf{ts}} = \bigwedge_i d_{k,i}([\![a]\!]_{\mathsf{t}}) = [\![t_n]\!]_{\mathsf{t}}$$

to denote the translation of a context element ($[\![\cdots]\!]_{\mathsf{ts}}$ calculates the denotation of a term's sorting assertion). An entire context Γ is translated into the conjunction of the translation of its elements.

As we do for function symbols, we create for each dependent sorted predicate symbol $\mathcal{P}$ a corresponding non-dependent sorted one, written as $\overline{\mathcal{P}}$. We define the translation of formulas by induction as:

$$\begin{aligned}
[\![\mathcal{P}(t_1 : s_1, \cdots, t_n : s_n)]\!]_{\mathsf{f}} &= \overline{\mathcal{P}}([\![t_1]\!]_{\mathsf{t}}, \cdots, [\![t_n]\!]_{\mathsf{t}}) \\
[\![\phi \implies \psi]\!]_{\mathsf{f}} &= [\![\phi]\!]_{\mathsf{f}} \implies [\![\psi]\!]_{\mathsf{f}} \\
[\![\forall x : s.\ \psi]\!]_{\mathsf{f}} &= \forall x : \overline{s}.\ [\![x]\!]_{\mathsf{ts}} \implies [\![\psi]\!]_{\mathsf{f}}
\end{aligned}$$

Finally, a theorem $\Gamma, A \vdash \psi$ translates into

$$\forall v_1 \ldots v_n. \bigwedge_{(v_i:s_i)\in\Gamma} [\![v_i : s_i]\!]_{\mathsf{ts}} \wedge [\![A]\!]_{\mathsf{f}} \implies [\![\psi]\!]_{\mathsf{f}}$$

It is routine to check that the rules are valid under the translation and hence have the intended sense. As an example, consider ∀-I. Assume $\Gamma, \{a : s(t_1, \cdots, t_n)\}, A \vdash \phi(a)$ and the variable a appears in neither Γ nor A. The theorem translates into

$$[\![\Gamma]\!]_{\mathsf{ts}}, [\![a : s(t_1, \cdots, t_n)]\!]_{\mathsf{ts}}, \bigwedge [\![A]\!]_{\mathsf{f}} \vdash [\![\phi(a)]\!]_{\mathsf{f}}$$

(where we overload $[\![\cdots]\!]_{\mathsf{ts}}$ and $[\![\cdots]\!]_{\mathsf{f}}$ to include the versions mapping sets to conjunctions of translations). The fact that a does not appear in Γ translates into the corresponding variable $a : \overline{s}$ not appearing in $[\![A]\!]_{\mathsf{f}}$, and the requirement that no variable depends on a translates to the requirement that $[\![a : s(\ldots)]\!]_{\mathsf{ts}}$ does not appear in $[\![\Gamma]\!]_{\mathsf{ts}}$ either. Therefore, we can discharge $[\![a]\!]_{\mathsf{ts}}$ from the assumption and deduce from the FOL universal elimination rule that $[\![\Gamma]\!]_{\mathsf{ts}}, [\![A]\!]_{\mathsf{f}} \vdash \forall a : [\![s]\!]_{\mathsf{s}}.\ [\![a : s]\!]_{\mathsf{ts}} \implies [\![\phi(a)]\!]_{\mathsf{f}}$. This is the translation of $\Gamma, A \vdash \forall a : s(t_1, \cdots, t_n).\phi(a)$, as required.

Implementation

Our implementation is a proof-of-concept written in SML. It provides a simple REPL similar to those provided by HOL4 and HOL Light. The kernel (core syntax and proof rules) is implemented in 2443 lines of code; user-level parsing (including a simple type inference algorithm) and printing is a further 1633 lines of code. Additional core libraries (goal stack package, common tactics including the rewriting tactic) take 4386 lines.

The source code for this implementation is available from `https://github.com/u5943321/DiaToM`

3 ETCS and SEAR

ETCS [5] and SEAR [17] are both structural set theories. With each, we work within a well-pointed boolean topos. In particular, they both have products, coproducts, exponentials, an initial object 0 and a terminal object 1. Whereas the existence of all of these are given as primitive axioms in ETCS, we can construct them in SEAR.

ETCS has two sorts: objects (A, B, ...; a ground sort) and arrows (*e.g.,* $A \to B$), where an arrow sort depends on two object terms. Equality can only hold between arrows. An object is to be considered as a set in the usual sense: an arrow $1 \to X$ is regarded as an element of the set X. As *per* Shulman's original construction, SEAR has three sorts: sets (A, B, ...; a ground sort); members ($\_ \in A$, depending on a set term); and relations ($A \nrightarrow B$, depending on two set terms). SEAR also adds a primitive predicate $\mathsf{Holds}(R : A \nrightarrow B, a \in A, b \in B)$, declaring that the relation R relates a and b.Equality can hold between relations with the same domain and codomain, and elements of the same set.

In SEAR, a relation R is called a function if $\forall a.\exists!b.\ \mathsf{Holds}(R, a, b)$. In practice, we want to be able to write $f(a)$ as the result of applying a function to an argument, but we cannot do this if we are restricted to just the relation sort. A first thought might be to create a function symbol Eval, that takes a relation and a member of A, so the term $\mathsf{Eval}(R : A \nrightarrow B, a \in A)$ is a member of B. However, such a function symbol breaks soundness, as the term $\mathsf{Eval}(R, a)$ can be expressed for every a of the correct sort before checking the function condition on R. In particular, we can write a term $\mathsf{Eval}(R : 1 \nrightarrow 0, *)$, nominally producing an element of 0.

Rather, we introduce a function sort which is a "proper subsort" of the relation sort.[1] A function f from A to B is written $f : A \to B$, and we add the following axiom describing terms of function sorts:

$$\mathsf{isFunction}(R) \implies \\ \exists!f : A \to B.\ \forall(a \in A)\ (b \in B).\mathsf{Eval}(f, a) = b \Leftrightarrow \mathsf{Holds}(R, a, b)$$

The $\mathsf{isFunction}$ predicate embodies the definition above, and we also have a new Eval function symbol that takes a function term from A to B and an element term of A, and outputs an element term of B.

We will write $\mathsf{Eval}(f, a)$ simply as $f(a)$ in the rest of paper. The Eval symbol is typed so that only functions terms can be its first argument. It is clear that this is a conservative extension, as any theorems involving Eval can be expressed using just Holds and uses of the $\mathsf{isFunction}$ hypothesis if desired.

Subsets are handled differently in ETCS and SEAR. Using the SEAR axioms, it is straightforward to show that for each formula ϕ on members $x \in X$, we can form the subset $\{x \mid \phi(x)\}$. In what follows, $\mathcal{F}$ is an arbitrary formula variable, and we are defining a *comprehension schema*. Our subset is constructed *via* a member of the power set $\mathsf{Pow}(X)$,[2] and ultimately as a term of set sort with an injection to X. This construction is described by the following two theorems (following Shulman [17]). First, we prove the existence of the member of the power-set. Given that A is a set, then IN requires two arguments of sort $\_ \in A$ and $\_ \in \mathsf{Pow}(A)$. Then:

$$\exists!s \in \mathsf{Pow}(A).\ \forall a.\ \mathsf{IN}(a, s) \Leftrightarrow \mathcal{F}[\mathsf{mem}(A)](a)$$

We also have the existence of a set B, and an injection from it into A:

$$\exists B\ (i : B \to A).\ \mathsf{Inj}(i) \wedge \forall(a \in A).\ \mathcal{F}[\mathsf{mem}(A)](a) \Leftrightarrow \exists b \in B.\ a = i(b) \tag{1}$$

The combination of i and B can be seen as identifying the subset of A satisfying predicate P.

The following isset predicate, connecting a member (s) to a set (B, given implicitly in i's sort) is also occasionally useful:

$$\mathsf{isset}(i : B \to A, s \in \mathsf{Pow}(A)) \overset{\text{def}}{\Leftrightarrow} \mathsf{Inj}(i) \wedge \mathsf{image}(i, B) = s$$

The "subset story" in ETCS is more restrictive. There we can only form subsets from predicates on elements of X which can be captured by an arrow $p : X \to 2$, where 2 is defined to be the coproduct $1 + 1$. Such arrows are turned into elements of the power object 2^X by taking transposes. We regard 2 as the set of truth values, where $\top_I, \bot_I : 1 \to 2$ denote truth

[1] Shulman (personal communication) agrees that the resulting system is still effectively SEAR as he conceives it.

[2] The existence of the powerset function is easy to establish from the function specification rule: power set of each set is unique up to isomorphism that respects the membership relation.

and falsity respectively. Our arrow p gives rise to a separate object as characterised by the theorem:

$$\forall A\ (p : A \to 2).\ \exists B\ (i : B \to A).\ \mathsf{Inj}(i) \wedge \forall a : 1 \to A.\ p \circ a = \top_I \Leftrightarrow \exists b.\ a = i \circ b$$

The existence of i is witnessed by the pullback of the map $\top_I$ along p. Note that this method only shows the existence of subsets for arrows $p : X \to 2$. We do not achieve the generality of SEAR, where the construction starts with an arbitrary formula variable. ETCS *does* allow for the construction of subsets using something resembling set comprehension, but this requires a detour *via* its internal logic (see Section 4 below).

Another notable difference between the two logics is that ETCS comes with the axiom of choice in the form of the statement that any epimorphism has a section, whereas this is not given in SEAR. In fact, if we change SEAR by adding the axiom of choice, and also requiring that the input formula of our comprehension schema be bounded, then the resulting system has the same strength as ETCS.

For both ETCS and SEAR, the injection we construct from each predicate is unique up to respectful isomorphism. This allows us to use the specification rule to obtain new constants without the full form of choice. In SEAR, for example, we can prove if there are $i : B \to A$ and $i' : B' \to A$, which are both injections, and moreover, we have $\forall a.\ P(a) \Leftrightarrow \exists b \in B.\ a = i(b)$ and $\forall a.\ P(a) \Leftrightarrow \exists b \in B'.\ a = i'(b)$, then the relation between pairs $(B, i : B \to A)$ and $(B', i' : B' \to A)$ defined by

$$\begin{aligned}&\exists (f : B \to B')\ (g : B' \to B).\\ &\quad f \circ g = \mathsf{Id}(B) \wedge g \circ f = \mathsf{Id}(B') \wedge i' \circ f = i \wedge i \circ g = i'\end{aligned}$$

holds. This is clearly an equivalence relation. Moreover, for all sets A, the existence of a set B and a map $B \to A$ is witnessed by the identity isomorphism. Therefore, once we instantiate the P above into a concrete predicate without any formula variables, we have met all of the specification rule's antecedents, and we can use it to define two constants: the subset and its inclusion into the ambient set. In SEAR, the sets of natural numbers, integers, lists and co-lists are all constructed in this way. More generally, given any member $s \in \mathsf{Pow}(A)$, we use the specification rule to turn it into a "real set" via the constant $\mathsf{m2s}(s)$ of set sort. This set is injected into A by the map $\mathsf{minc}(s) : \mathsf{m2s}(s) \to A$.

4 Internal logic in ETCS

As discussed in the last section, an arrow $p : X \to 2$ corresponds to a predicate on X in the sense that if $x : 1 \to X$, then $p \circ x = \top_I$ means p is true for x. An ETCS formula is *bounded* precisely when all quantified variables are elements (*i.e.*, arrows with domain 1). Let us call the formulas of our logic (all formulas seen so far) *external formulas*. If an external formula is bounded with all free variables also elements, we can automatically construct a corresponding *internal formula* as a term of the logic. When the external formula is on variables with sorts $(1 \to X_1), (1 \to X_2), \ldots$, then the internal formula will be an arrow of sort $\Pi X_i \to 2$. For an external formula $\Phi[x_1 : 1 \to X_1, \ldots]$, then let $p : \Pi X_i \to 2$ be the corresponding formula. We require

$$\forall a : 1 \to \Pi X_i.\ p \circ a = \top_I \Leftrightarrow \Phi[(\pi_i \circ a)/x_i]$$

where $\Phi[t/x]$ is the substitution of term t for variable x. This could be regarded as an axiom, one rather like Separation in ZF. However, we can instead prove all results of this form automatically. This is simply by rewriting with all the theorems with relevant definitions and properties of the internal logic operators as explained below.

We have implemented an automatic translation (a "derived rule") that generates an internal logic formula given a list of variables, considered as the arguments, and the formula. The translation produces an internal logic predicate and proves that it gives the value $\top_I$ if and only if the formula is true when applied to the arguments. We illustrate our algorithm with an example over $\mathbb{N}$, the natural number object, the arrow $\mathsf{SUC} : \mathbb{N} \to \mathbb{N}$, and the function symbol $\_^+$, such that $n^+ \stackrel{\text{def}}{=} \mathsf{SUC} \circ n$. Then, the pair $([n], m^+ - n^+ = m - n)$ encodes a simple unary predicate on n. In this case, the output of our derived rule is an arrow term $p : \mathbb{N} \to 2$ satisfying:

$$\forall n : 1 \to \mathbb{N}.\ \ p \circ n = \top \Leftrightarrow m^+ - n^+ = m - n$$

If the list of arguments is $[m, n]$ instead, the produced arrow $p : \mathbb{N} \times \mathbb{N} \to 2$ will satisfy:

$$\forall m, n : 1 \to \mathbb{N}.\ p \circ \langle m, n \rangle = \top \Leftrightarrow m^+ - n^+ = m - n$$

Table 1 Operators of the Internal Logic.

Operator	Sort	Defining Property
$\wedge_I$	$2 \times 2 \to 2$	$\wedge_I \circ \langle p_1, p_2 \rangle \Leftrightarrow p_1 = \top_I \wedge p_2 = \top_I$
$\vee_I$	$2 \times 2 \to 2$	$\vee_I \circ \langle p_1, p_2 \rangle \Leftrightarrow p_1 = \top_I \vee p_2 = \top_I$
$\Rightarrow_I$	$2 \times 2 \to 2$	$\Rightarrow_I \circ \langle p_1, p_2 \rangle \Leftrightarrow p_1 = \top_I \implies p_2 = \top_I$
$\neg_I$	$2 \to 2$	$\neg_I \circ p = \top_I \Leftrightarrow p = \bot_I$
$\forall_X$	$2^X \to 2$	$\forall_X \circ \overline{p} \circ y = \top_I \Leftrightarrow \forall x.p \circ \langle x, y \rangle = \top_I$
$\exists_X$	$2^X \to 2$	$\exists_X \circ \overline{p} \circ y = \top_I \Leftrightarrow \exists x : 1 \to X.p \circ \langle x, y \rangle = \top_I$

To convert formulas into internal formulas, we need to first convert terms into "internal terms". In particular, function symbols will map into arrows of an appropriate sort. For example, if our "external formula" is on variables $[x : 1 \to \mathbb{N}, y : 1 \to \mathbb{N}]$, then any "internal term" built as part of this translation will be from $\mathbb{N} \times \mathbb{N}$. In our $\mathbb{N}$-example, the arrow corresponding to y^+ will be $\mathsf{SUC} \circ \pi_2(\mathbb{N}, \mathbb{N})$. In most circumstances, the connection between the function symbol and the arrow will simply be that symbol's definition. For generality's sake, our implementation stores the external-internal correspondence of function and predicate symbols in a simple dictionary data structure.

Our formula-converting function is recursive on the structure of formula, using the semantics of the various connectives and quantifiers given in Table 1. The only built-in predicate, equality, corresponds to the characteristic map of the diagonal monomorphism. For user-defined predicates, such as $<$ over natural numbers, users can store the correspondences manually. The induction steps for the connectives are straightforward. For quantifiers, for example, consider the formula $\forall a : 1 \to A.\ \ a = a_0$. Begin by converting the body $a = a_0$ into a predicate on $[a, a_0]$; and then transpose the output and post-compose with the internal logic operator $\forall_A$. The existential case is similar.

5 Quotients in ETCS and SEAR

In both ETCS and SEAR, we can make a number of definitions, and prove theorems about quotienting by equivalence relations. Here we present our approach in the terminology of SEAR. We only consider full equivalence relations, since partial equivalences become full by restricting their domains. Our approach does not require any form of the Axiom of Choice. Given a binary relation R on a set A, we say a map $i : Q \to \mathsf{Pow}(A)$ is a quotient with respect

to R if it injects Q into the set of relational images of R (which is the set of equivalence classes if R is an equivalence relation). That is,

$$\begin{aligned}&\mathsf{Quot}(R : A \nrightarrow A, i : Q \to \mathsf{Pow}(A)) \Leftrightarrow \\ &\quad \mathsf{Inj}(i) \ \wedge \ \forall s \in \mathsf{Pow}(A).\ (\exists q \in Q.\ s = i(q)) \Leftrightarrow \exists a \in A.\ s = \{b \mid \mathsf{Holds}(R, a, b)\}\end{aligned}$$

In contrast to HOL, where any injection has an inverse, constructing an inverse of an injection requires an element witnessing that the domain is non-empty. For an injection i from X to Y, and given an element $x \in X$, we define $\mathsf{LINV}(i, x)(y)$ to map $y \in Y$ to x_0 if $i(x_0) = y$, or to x otherwise. This is then a left inverse of i. If $i : Q \to \mathsf{Pow}(A)$ is a quotient of R, then given any $q_0 \in Q$, the composition of the map $a \mapsto \{b \mid \mathsf{Holds}(R, a, b)\}$ and $\mathsf{LINV}(i, q_0)$ is the quotient map $A \to Q$. We denote the output of this map applied to an element $a \in A$ as $\mathsf{abs}(R, i, q_0, a)$. We write $\mathsf{resp1}(f, R)$ if f agrees on elements related by R and $\mathsf{ER}(R)$ for R an equivalence relation. Then we prove:

$$\begin{aligned}&\mathsf{ER}(R) \wedge \mathsf{resp1}(f, R) \wedge \mathsf{Quot}(R, i) \implies \\ &\quad \forall q_0 \in Q.\ \exists! \overline{f} : Q \to B.\ \forall a \in A.\ \overline{f}(\mathsf{abs}(R, i, q_0, a)) = f(a)\end{aligned}$$

This does not only allow us to lift functions at the level of elements related by R, but also supports lifting predicates, which can be regarded as maps to 2. For instance, lifting the definition of evenness of a natural number to that of an integer amounts to lifting a map $\mathbb{N} \to 2$ into $\mathbb{Z} \to 2$.

A function into a quotient can be defined by composing with the inverse of the inclusion map and hence is easy to define. The interesting case is when we want to define a function from a product of quotients. In such cases, we realise the product of quotients as a quotient as well in the following way: Given two relations R_1 on A and R_2 on B, we define their product relation as:

$$\mathsf{Holds}(\mathsf{prrel}(R_1, R_2), (a_1, b_1), (a_2, b_2)) \Leftrightarrow \mathsf{Holds}(R_1, a_1, a_2) \wedge \mathsf{Holds}(R_2, b_1, b_2)$$

And given quotients $i_1 : Q_1 \to \mathsf{Pow}(A), i_2 : Q_2 \to \mathsf{Pow}(B)$ of R_1 and R_2, we define a map $\mathsf{ipow2}(i_1, i_2) : Q_1 \times Q_2 \to \mathsf{Pow}(A \times B)$ such that for every pair $(a, b) \in Q_1 \times Q_2$, we have:

$$\mathsf{IN}((a, b), \mathsf{ipow2}(i_1, i_2)(q_1, q_2)) \Leftrightarrow \mathsf{IN}(a, i_1(q_1)) \wedge \mathsf{IN}(b, i_2(q_2))$$

If R_1 and R_2 are both equivalence relations, we have $\mathsf{Quot}(\mathsf{prrel}(R_1, R_2), \mathsf{ipow2}(i_1, i_2))$ Application of this result allows us to define maps such as integer addition and multiplication, and more generally, the group operation in a quotient group.

6 Group Theory

Many mathematical results look neater in theorem-provers based on dependent type theory (DTT), since instead of assuming complicated predicates, we can internalise those predicates as types, thereby shortening the statement. By formalising some group theory, we demonstrate that we can prove similarly neat versions of statements in our simple logic.

We encode a group with underlying set G as an element of $\mathsf{Grp}(\mathsf{G})$. Such a set is constructed from the comprehension schema which injects to the subset of the product $G^{G \times G} \times G^G \times G$ satisfying the usual group axioms. For a group $g \in \mathsf{Grp}(G)$, also by comprehension, we construct the set of its subgroups $\mathsf{sgrp}(g)$ as injected into $\mathsf{Pow}(G)$, and set of its normal subgroups $\mathsf{nsgrp}(g)$ that injects to $\mathsf{sgrp}(g)$. As groups are encoded by members of sets, it is possible to compare if two groups are equal, *e.g.*, $g_1 = g_2$, with $g_1, g_2 \in \mathsf{Grp}(G)$. However,

if $h_1 \in \mathsf{sgrp}(g_1)$ and $h_2 \in \mathsf{sgrp}(g_2)$, we cannot write $h_1 = h_2$ because such an equality will not type check. We hold this to be appropriate because equality is not the correct way to compare abstract structures such as groups. Even if we wanted to work with equality on groups g_1, g_2, we should compare their representatives or define transferring functions like the ones of sort $\mathsf{sgrp}(g_1) \to \mathsf{sgrp}(g_2)$, which map a subgroup of g_1 to a subgroup of g_2.

For a normal subgroup $N \in \mathsf{nsgrp}(g)$, the underlying set of the quotient group $\mathsf{qgrp}(N)$ has as its underlying set the set of all right cosets of N. The function symbol qgrp only needs to take the group N as argument, since the group being quotiented is contained in the sort information of N. The quotient homomorphism $\mathsf{qhom}(N)$ is a member of $\mathsf{ghom}(g, \mathsf{qgrp}(g))$ of all homomorphisms between the original group and the quotient. Its underlying function $\mathsf{homfun}(\mathsf{qhom}(N))$ sends a group element to its coset.

By construction, each underlying function of a homomorphism respects the relation induced by its kernel. Then the first isomorphism theorem can be obtained by instantiating the quotient mapping theorem as in the last section, giving

$$\begin{array}{l}\forall G_1\ G_2\ \ g_1 \in \mathsf{Grp}(G_1)\ \ g_2 \in \mathsf{Grp}(G_2)\ \ f \in \mathsf{ghom}(g_1, g_2).\\ \quad \exists!\ \overline{f} \in \mathsf{ghom}(\mathsf{qgrp}(\mathsf{ker}(f)), g_2).\\ \qquad \mathsf{Inj}(\mathsf{homfun}(\overline{f}))\ \wedge\ \mathsf{homfun}(\overline{f}) \circ \mathsf{qmap}(\mathsf{ker}(f)) = \mathsf{homfun}(f)\end{array} \tag{2}$$

This is a nice illustration of the strengths of the "DTT style".

6.1 Discussion

Our approach to group theory is very different from its counterpart in HOL. Firstly, the HOL type α group is inhabited by values that must record their underlying carrier set. Secondly, the HOL quotient group function takes two α groups and outputs a term of $(\alpha \to \mathsf{bool})$ group, which is proved to satisfy the group axioms if the first term satisfies the group axioms and the second term is a normal subgroup. Further, as HOL types cannot depend on terms, we certainly cannot construct the type of all homomorphisms between two groups.

There is actually a trade-off between choosing the HOL style and the DTT style of stating theorems. Whereas the first isomorphism theorem is clearly better in DTT style (2), the second and third isomorphism theorems in DTT style can look complicated, with a great deal of coercions happening under the covers.[3] Since the HOL quotient group only takes two groups of the same type, we can use exactly the same term for the ambient group and its subgroup, and do not need to construct different terms to regard the same group as subgroups of an ambient group. In this case, the convenience of the HOL style (using assumptions) is evident. We can choose each style in our system, so users can try both approaches and compare them. To find the best form of a statement, we may try combining the two approaches: we do not always have to create a subset once we come up with a new property, but we may use them as assumptions as well.

7 Inductive and Coinductive definitions

We experiment with inductive definitions by mechanising induction on natural numbers, finite sets and lists, and with coinductive definitions by constructing co-lists.

[3] Of course, DTT *systems* offer the ability to write statements in HOL's predicate-heavy style as well.

7.1 Natural numbers, Finite sets and Lists

Our system implements a version of Harrison's [3] inductive relation definition package. To define an inductive subset, we just need to provide the inductive clauses.

For example, there is no primitive natural number object in SEAR. We are given only a set $\mathbb{N}_0$ with an element z_0 and an injection $\mathsf{s}_0 : \mathbb{N}_0 \to \mathbb{N}_0$, where z_0 is not in the range of s_0. To apply our induction tool to cut down $\mathbb{N}_0$ into a natural number object, we firstly define a subset, i.e., a member of the power set $\mathsf{Pow}(\mathbb{N}_0)$, of $\mathbb{N}_0$, by giving two clauses saying that z_0 is in N and if n_0 is in N, then $\mathsf{s}_0(n_0)$ is in N. Using theorem 1 in Section 3 together with the specification rule, we extract the subset of $\mathbb{N}_0$, which consists of elements in N, as a constant term $\mathbb{N}$ of set sort. We call the lifted zero element and successor map 0 and SUC respectively, with SUC obtained by specialising the following lemma with the inclusion from $\mathbb{N}_0$:

$$\begin{array}{l}\forall A\ A_0\ (i : A \to A_0)\ (f_0 : A_0 \to A_0).\\ \quad \mathsf{Inj}(i)\ \wedge\ (\forall a_1.\exists a_2.\ f_0(i(a_1)) = i(a_2))\ \Longrightarrow\\ \qquad \exists! f : A \to A.\ \forall a \in A.\ i(f(a)) = f_0(i(a))\end{array}$$

The constructed $\mathbb{N}$ then can be shown to satisfy the standard induction principle.

$$\begin{array}{l}\mathcal{F}[\mathsf{mem}(\mathbb{N})](0) \wedge (\forall n \in \mathbb{N}.\ \mathcal{F}[\mathsf{mem}(\mathbb{N})](n)\ \Longrightarrow\ \mathcal{F}[\mathsf{mem}(\mathbb{N})](\mathsf{SUC}(n)))\ \Longrightarrow\\ \quad \forall n \in \mathbb{N}.\ \mathcal{F}[\mathsf{mem}(\mathbb{N})](n)\end{array}$$

By instantiating the formula variable $\mathcal{F}$ with concrete properties, we apply the above to perform inductive proofs for ordering and natural number arithmetic. We later use such theorems together with quotient lemmas to construct the set of integers.

Also inductively, we define the predicate $\mathsf{isFinite}$ on members of some set X's power set. The empty subset $\mathsf{Empty}(X)$ is finite, and if $s \in \mathsf{Pow}(X)$ is finite, then the set $\mathsf{Ins}(x, s)$, which inserts x into s, is finite for any $x \in X$. Similar to the counterpart of natural numbers, the principle of induction on the finiteness of a set is proved as:

$$\begin{array}{l}\mathcal{F}[\mathsf{mem}(\mathsf{Pow}(X))](\mathsf{Empty}(X)) \wedge\\ (\forall x\ (xs_0 \in \mathsf{Pow}(X)).\ \mathcal{F}[\mathsf{mem}(\mathsf{Pow}(X))](xs_0)\ \Longrightarrow\ \mathcal{F}[\mathsf{mem}(\mathsf{Pow}(X))](\mathsf{Ins}(x, xs_0)))\ \Longrightarrow\\ \quad \forall xs \in \mathsf{Pow}(X).\ \mathsf{isFinite}(xs)\ \Longrightarrow\ \mathcal{F}[\mathsf{mem}(\mathsf{Pow}(X))](xs)\end{array}$$

We define a relation $\mathsf{Pow}(X) \nrightarrow \mathbb{N}$ relating a subset of X to its cardinality. By induction on finiteness, we prove each subset is related to a unique natural number, which gives us a function $\mathsf{Pow}(X) \to \mathbb{N}$ that sends a finite subset to its cardinality and sends any infinite subset to 0. The output of the function applied on $s \in \mathsf{Pow}(X)$ is denoted as $\mathsf{Card}(s)$. To build lists over a set X as an "inductive type", we firstly define the subset of $\mathsf{Pow}(\mathbb{N} \times X)$ which encodes a list, such sets are finite sets of the form $\{(0, x_1), \cdots, (n, x_n)\}$. The base case of the induction is the empty subset of $\mathsf{Pow}(\mathbb{N} \times X)$, and the step case inserts the set s started with by the pair $(\mathsf{Card}(s), x)$. Using the same approach we constructed $\mathbb{N}$, we form $\mathsf{List}(X)$. It is then straightforward to prove the list induction principle and define the usual list operations like taking the head, tail, n-th element of the list, and map, etc.

7.2 Co-lists

Following the HOL approach, we construct co-lists over sets X, by using maps $\mathbb{N} \to X + 1$ as representatives. The codomain is regarded as X option, whose members either have the form $\mathsf{SOME}(x)$ for $x \in X$, or $\mathsf{NONE}(X)$. First, by dualising the argument we used to define inductive predicates, we define a coinductive predicate on members ($f \in (X+1)^{\mathbb{N}}$) expressing that such a member captures a co-list, and collect the subset where this predicate holds,

defining $\mathsf{list_c}(X)$, just as we did for constructing inductive types. Every term of $\mathsf{list_c}(X)$ has a representative: it is either the constant function mapping to $\mathsf{NONE}(X)$, corresponding to the empty co-list $\mathsf{Nil_c}(X)$, or it is the function obtained by attaching an element $x \in X$ to an existing function encoding a co-list. Almost all the HOL4 definitions can be readily translated into SEAR. The only exception is we cannot write expressions such as $\mathsf{THE}(\mathsf{Hd_c}(l))$. Here $\mathsf{Hd_c}$ is the function that returns $\mathsf{SOME}(x)$ when l is a co-list with element x at its front. If l is the empty co-list, then $\mathsf{Hd_c}(l) = \mathsf{NONE}$. In HOL4, THE is the left-inverse of SOME; in SEAR, our (set) parameter X may be empty, and so there is no general value (even if unspecified) for the head of the co-list. So far, this has not been an obstacle in any of our proofs. The HOL proof of the key co-list principle, which states that two co-lists $l_1, l_2 \in \mathsf{list_c}(X)$ are equal if and only if they are connected by a bisimulation relation R, translates into SEAR, yielding:

$$
\begin{array}{l}
l_1 = l_2 \Leftrightarrow \\
\exists R : \mathsf{list_c}(X) \nrightarrow \mathsf{list_c}(X). \\
\quad \mathsf{Holds}(R, l_1, l_2) \wedge \\
\quad \forall l_3\ l_4 \in \mathsf{list_c}(X).\ \ \mathsf{Holds}(R, l_3, l_4) \implies \\
\qquad\qquad\qquad\qquad (l_3 = \mathsf{Nil_c}(X) \wedge l_4 = \mathsf{Nil_c}(X)) \vee \\
\qquad\qquad\qquad\qquad \exists (h \in X)\ t_1\ t_2.\ \mathsf{Holds}(R, t_1, t_2) \wedge l_3 = \mathsf{Cons_c}(h, t_1) \wedge l_4 = \mathsf{Cons_c}(h, t_2)
\end{array}
$$

where $\mathsf{Nil_c}(X)$ is the empty co-list over X, and $\mathsf{Cons_c}(h, t)$ is the co-list built by putting element $h \in X$ in front of co-list t. We can perform coinductive proofs on co-lists by the theorem above. For instance, the above helps to prove that $\mathsf{Map_c}$ function, with the usual definition, is functorial.

8 Modal Model Theory

In recent work, we developed a mechanisation of some basic modal logic theory [20]. While defining the notion of being preserved under simulation, we observed that if a property of a modal formula is defined in terms of the behaviour of the formula on all models, then such a property cannot be faithfully captured by HOL. Such an issue can be resolved by choosing a dependent sorted foundation and doing the proof in our logic. We demonstrate this here by mechanising the proof that characterises formulas preserved under simulation as those are equivalent to a positive existential formula in SEAR.

Using roughly the general method introduced at the end of Harrison [3], we first construct the "type" (actually a set in SEAR) of modal formulas over variables drawn from the set V. We then denote the set of modal formulas over V as $\mathsf{form}(V)$. A Kripke model on a set W of such formulas is an element of $\mathsf{Pow}(W \times W) \times \mathsf{Pow}(V)^W$ (written as $\mathsf{model}(W, V)$ in the following paragraphs). The first component encodes the model's reachability relation, while the second encodes the variable valuation. Satisfaction of modal formulas can then be defined in the standard way, and if ϕ is satisfied at w in the model M, we write $M, w \Vdash \phi$.

The two key definitions of this proof are that of simulation, and of being preserved under simulation (written as PUS below). The former is identical to its counterpart in HOL, and we write $\mathsf{Sim}(R, M_1, M_2)$ to indicate that R is a simulation from M_1 to M_2. The latter is more interesting. Unlike in HOL, where we can only express a formula being preserved under simulation between models of certain HOL types, forcing the definition to take an extra type

parameter, the definition in SEAR is purely a predicate on formulas:

$$\begin{array}{l}\forall V\ (\phi \in \mathsf{form}(V)).\\ \quad \mathsf{PUS}(\phi) \Leftrightarrow\\ \quad \forall W_1\ W_2\ (R : W_1 \nrightarrow W_2)\ (w_1 \in W_1)\ (w_2 \in W_2)\\ \quad\ (M_1 \in \mathsf{model}(W_1, A))\ \ (M_2 \in \mathsf{model}(W_2, A)).\\ \quad\quad \mathsf{Sim}(R, M_1, M_2) \wedge \mathsf{Holds}(R, w_1, w_2)\ \wedge\ M_1, w_1 \Vdash \phi\ \Longrightarrow\ M_2, w_2 \Vdash \phi\end{array}$$

This convenience is brought about by the fact that our logic allows for quantification over sets, whereas HOL does not allow for quantification over types. Thus, the notion of equivalence of modal formulas only takes two modal formulas as arguments and requires no extra "type parameter". Under these definitions, the proofs of both directions of theorem 2.78 in [1] can be faithfully translated, yielding the two formal statements:

$$\forall V\ (\phi \in \mathsf{form}(V))\ (\phi_0 \in \mathsf{form}(V)).\ \mathsf{PE}(\phi_0) \wedge \phi \sim \phi_0\ \Longrightarrow\ \mathsf{PUS}(\phi)$$

$$\forall V\ (\phi \in \mathsf{form}(V)).\ \mathsf{PUS}(\phi)\ \Longrightarrow\ \exists \phi_0 \in \mathsf{form}(V).\ \mathsf{PE}(\phi_0) \wedge \phi \sim \phi_0$$

Clearly, the two directions can be put together into an if-and-only-if, hence giving the full form of the characterisation theorem, which cannot even be stated in HOL.

$$\forall V\ (\phi \in \mathsf{form}(V)).\ \mathsf{PUS}(\phi) \Leftrightarrow \exists f_0 \in \mathsf{form}(V).\ \mathsf{PE}(\phi_0) \wedge \phi \sim \phi_0)$$

9 Existence of Large Sets

Whereas iterating the procedure of taking the power set by infinite times is impossible in HOL due to foundational issues, the collection axiom schema in SEAR makes it possible. The statement of the SEAR collection axiom is formalised as:

$$\begin{array}{l}\exists B\ Y\ (p : B \to A)\ (M : B \nrightarrow Y).\\ \quad (\forall S\ (i : S \to Y)\ (b \in B).\\ \quad\quad \mathsf{isset}(i, \{y \mid \mathsf{Holds}(M, b, y)\})\ \Longrightarrow\ \mathcal{F}[\mathsf{mem}(A), \mathsf{set}](p(b), S)) \wedge\\ \quad (\forall (a \in A)\ X.\ \mathcal{F}[\mathsf{mem}(A), \mathsf{set}](a, X)\ \Longrightarrow\ \exists b.\ p(b) = a)\end{array}$$

with $\mathcal{F}[\mathsf{mem}(A), \mathsf{set}]$ a formula variable, to be instantiated to be a predicate on an element of A and a set.

Using this axiom, we will prove:

$$\forall A.\ \exists P.\ \forall n \in \mathbb{N}.\ \exists i : \mathsf{Pow}^n(A) \to P.\ \mathsf{Inj}(i)$$

Here the $\mathsf{Pow}^n(A)$ is "the" n-th power set of A. Note that the induction principle on natural numbers does not allow us to take a set as an argument, and does not allow the output to be a set as well. To create this function symbol, we start by defining a predicate $\mathsf{nPow}(n, A, B)$, which means B is *an* n-th power set of A. We then prove such B is unique up to bijection, hence the specification rule applies. In the following, we write $\mathsf{P}(s) \in \mathsf{Pow}(\mathsf{Pow}(A))$ for the set of subsets of $s \in \mathsf{Pow}(A)$. For $s_1 \in \mathsf{Pow}(A)$ and $s_2 \in \mathsf{Pow}(B)$, we write $|s_1| = |s_2|$ for s_1 and s_2 have the same cardinality. We write $\mathsf{Whole}(A) \in \mathsf{Pow}(A)$ as the subset of A consisting of all members of A.

We define $\mathsf{nPow}(n, A, B)$ if there exists a set X and a function $f : X \to \mathbb{N}$ such that $|f^{-1}(0)| = |\mathsf{Whole}(A)|$, $|f^{-1}(n)| = |\mathsf{Whole}(B)|$, and for each $n_0 < n$, $|f^{-1}(n_0^+)| = |\mathsf{P}(f^{-1}(n_0))|$. Such a function f records a sequence of power set relation, in this case, we write $\mathsf{nPowf}(n, A, B, f)$. By induction on n_0, $\mathsf{nPow}(n, A, B, f)$, implies $\mathsf{nPow}(n_0, A, \mathsf{m2s}(f^{-1}(n_0)), f)$ for each $n_0 \leq n$.

If $\mathsf{nPow}(n, A, B_1)$ and $\mathsf{nPow}(n, A, B_2)$, we can infer B_1 and B_2 have the same cardinality by induction on n. The base case is trivial. Assume $f_1 : X_1 \to \mathbb{N}$ witnesses $\mathsf{nPow}(n^+, A, C_1)$ and $f_2 : X_2 \to \mathbb{N}$ witnesses $\mathsf{nPow}(n^+, A, C_2)$, as $n < n^+$, we have f_1, f_2 witness that their preimage at n is an n-th powerset of A, and hence by inductive hypothesis has the same cardinality. Therefore, the cardinality of C_1 and C_2 are equal as power sets of sets with the same cardinality.

Now we prove the existence of these iterated power sets. Suppose we have $\mathsf{nPowf}(n, A, B, f_0 : X \to \mathbb{N})$, we construct $f' : \mathsf{Pow}(X + 1) \to \mathbb{N}$ such that $\mathsf{nPowf}(n^+, A, \mathsf{Pow}(B), f')$. Define $f : X \to \mathbb{N}$ such that as if $f_0(x) \leq n$ then $f(x) = f_0(x)$, else $f(x) = n^{++}$, then we have $\mathsf{nPowf}(n, A, B, f : X \to \mathbb{N})$, and n^+ is not in the range of f. According to the definition of nPow, there exists an injection $B \to X$, and thus an injection $i : \mathsf{Pow}(B) \to \mathsf{Pow}(X)$. We define the function $f' : \mathsf{Pow}(X + 1) \to \mathbb{N}$ as:

$$f'(s) = \begin{cases} f(x) & \text{if } s = \{\mathsf{SOME}(x)\} \\ n^+ & \text{if } \exists xs \in \mathsf{Pow}(X).\ i(xs) = s_0 \wedge s = \{\mathsf{NONE(X)}\} \cup s_0 \\ n^{++} & \text{else} \end{cases}$$

It follows that $|f'^{-1}(n_0)| = |f^{-1}(n_0)|$ for $n_0 \leq n$, and the preimage of n^+ is a copy of $\mathsf{Pow}(B)$, so f' witnesses $\mathsf{Pow}(B)$ is the n^+-th power set of A.

To prove the existence of the large set. By specialising the axiom of collection, we obtain a set B, a function $p : B \to \mathbb{N}$, a set Y and a relation $M : B \nrightarrow Y$ satisfying:

$$(\forall S\ (i : S \to Y)\ (b \in B).\ \mathsf{isset}(i, \{y \mid \mathsf{Holds}(M, b, y)\}) \implies \mathsf{nPow}(p(b), A, S)) \wedge$$
$$(\forall n \in \mathbb{N}\ \ X.\ \mathsf{nPow}(n, A, X)) \implies \exists b \in B.\ p(b) = n)$$

The set Y is the large set we want to construct. For any $n \in \mathbb{N}$, we have $\mathsf{nPow}(n, A, \mathsf{Pow}^n(A))$, and thus there exists a $b \in B$ with $p(b) = n$. For this b, Let $H(b)$ denotes the set of elements y such that $\mathsf{Holds}(M, b, y)$, then $\mathsf{minc}(H(b))$ gives an injection $\mathsf{m2s}(H(b)) \to Y$. As $\mathsf{nPow}(n, A, \mathsf{m2s}(H(b)))$ and also $\mathsf{nPow}(n, A, \mathsf{Pow}^n(A))$, by uniqueness proved above, there exists a bijection $j : \mathsf{Pow}^n(A) \to \mathsf{m2s}(H(b))$. The composition $\mathsf{minc}(H(b)) \circ j$ is the desired injection.

10 Conclusion

Our work aims to enable the direct encoding of first-order mathematical foundations based on axioms, while keeping the underlying logic as simple as possible.

We have already seen that it is useful to explore various mathematical foundations: by experimenting with SEAR, we overcome two well-known shortcomings of HOL. Firstly, because it allows us to quantify over types, SEAR enables us to prove the full version of our previous theorem in modal model theory. Secondly, using the collection axiom of SEAR, we overcome the cardinality shortcoming of types in HOL. We are unaware of any other work addressing this issue.

10.1 Related Work

Quantification of types in HOL has been addressed in work by Melham [10] and Homeier [4]. Both pieces of work propose to extend the HOL logic, but neither goes so far as to introduce dependencies linking terms to types or sorts.

There is much existing work on logical systems with dependent sorts. All of them are designed with an aim different from ours. For instance, FOLDS (Makkai [7]) is designed to only be able to capture mathematical theories where truth is invariant under a certain notion

of isomorphism, and hence its expressive power is meant to be more restrictive. In particular, in its standard presentation, FOLDS works only with predicate symbols but not function symbols. DFOL (Rabe [15]) does not support expressing axiom schemata at the object level, and is constructed within LF's dependently typed environment. Compared with both, our work is customized for directly embedding axiomatic systems. Our system is simple, and can be easily implemented, not relying on an ambient implementation of dependent types.

When investigating a particular mathematical foundation, one approach is to implement the logic in a domain-specific manner. For instance, Cáccamo and Winskel [2], and New and Licata [12], both present logics addressing formalisation of proofs in category theory by designing particular type theories. In contrast, our system is a generic theorem-prover, making it easier to compare multiple systems, and to reuse proofs.

Isabelle (Paulson [13]) was famously designed as a generic theorem-proving system, and one of the sample object logics distributed with it is `MLTT` (Martin-Löf Type Theory). Nonetheless, as the ambient types of the Isabelle meta-level are those of simple type theory, working with dependent types in Isabelle requires the interesting type structures and typing judgements to appear at the level of terms. Once this compromise has been made, handling equalities, for example, becomes quite tedious; our system's restrictive handling of equality gains us a great deal of pragmatic power: simple rewriting, and a straightforward notion of matching.

10.2 Future Work

In future work, we will publish our formalisation of McLarty's CCAF [8] and our mechanisation of the proof theory of the system.

The existence of large sets is a consequence of the SEAR collection axiom, and is already stronger than what is possible in HOL, but there is still more that is possible in SEAR. In particular, from its collection axiom, we can follow Shulman [16] to derive the replacement schema, and get a minimal set from amongst these large sets. This would enable more transfinite constructions, such as that of Beth cardinals, which we plan to work on next. Moreover, we are interested in implementing a uniform approach of applying an axiomatic foundation as a metatheory, and hence developing the "two-layered" workspace discussed by McLarty and Rodin [9]. We are also looking forward to mechanising some of the theorems in the list "Formalising 100 Theorems" [19] in either SEAR or ETCS. Finally, it would be interesting to support the usage of different ambient logics, so people might, in particular, choose to do intuitionistic proofs as well.

References

1 Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001. doi:10.1017/CBO9781107050884.

2 Mario Cáccamo and Glynn Winskel. A higher-order calculus for categories. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics*, pages 136–153, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

3 John Harrison. Inductive definitions: automation and application. In Phillip J. Windley, Thomas Schubert, and Jim Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications: Proceedings of the 8th International Workshop*, volume 971 of *Lecture Notes in Computer Science*, pages 200–213, Aspen Grove, Utah, 1995. Springer-Verlag.

4 Peter V. Homeier. The HOL-Omega logic. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 244–259, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

5 F. William Lawvere. An elementary theory of the category of sets. *Proceedings of the National Academy of Sciences*, 52(6):1506–1511, 1964. doi:10.1073/pnas.52.6.1506.

6 F. William Lawvere. The category of categories as a foundation for mathematics. In S. Eilenberg, D. K. Harrison, S. MacLane, and H. Röhrl, editors, *Proceedings of the Conference on Categorical Algebra*, pages 1–20, Berlin, Heidelberg, 1966. Springer Berlin Heidelberg.

7 Michael Makkai. First order logic with dependent sorts, with applications to category theory. Available from https://www.math.mcgill.ca/makkai/folds/foldsinpdf/FOLDS.pdf, 1995.

8 Colin McLarty. Axiomatizing a category of categories. *The Journal of Symbolic Logic*, 56(4):1243–1260, 1991. URL: http://www.jstor.org/stable/2275472.

9 Colin McLarty and Andrei Rodin. A discussion between Colin McLarty and Andrei Rodin about structuralism and categorical foundations of mathematics, 2013. URL: http://philomatica.org/wp-content/uploads/2013/02/colin.pdf.

10 Thomas F. Melham. The HOL logic extended with quantification over type variables. In Luc J. M. Claesen and Michael J. C. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications*, IFIP Transactions A: Computer Science and Technology, pages 3–17. North-Holland, Amsterdam, 1993. doi:10.1016/B978-0-444-89880-7.50007-3.

11 Elliott Mendelson. *Introduction to Mathematical Logic*. Princeton: Van Nostrand, 1964.

12 Max S. New and Daniel R. Licata. A formal logic for formal category theory, 2022. doi:10.48550/ARXIV.2210.08663.

13 Lawrence Charles Paulson. Isabelle: The next 700 theorem provers. *ArXiv*, cs.LO/9301106, 2000.

14 W. V. Quine. New foundations for mathematical logic. *The American Mathematical Monthly*, 44(2):70–80, 1937. URL: http://www.jstor.org/stable/2300564.

15 Florian Rabe. First-order logic with dependent types. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning*, pages 377–391, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

16 Michael Shulman. Comparing material and structural set theories. *Annals of Pure and Applied Logic*, 170(4):465–504, 2019. doi:10.1016/j.apal.2018.11.002.

17 Michael Shulman. SEAR, 2022. URL: https://ncatlab.org/nlab/show/SEAR.

18 Andrzej Trybulec. Tarski-Grothendieck set theory, 1990. URL: http://mizar.uwb.edu.pl/JFM/Axiomatics/tarski.html.

19 Freek Wiedijk. Formalizing 100 theorems, 2013. URL: https://www.cs.ru.nl/~freek/100/.

20 Yiming Xu and Michael Norrish. Mechanised modal model theory. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *International Joint Conference on Automated Reasoning (IJCAR), Paris, France*, volume 12166 of *Lecture Notes in Computer Science*, pages 518–533. Springer, 2020. doi:10.1007/978-3-030-51074-9_30.

Formalizing Results on Directed Sets in Isabelle/HOL (Proof Pearl)

Akihisa Yamada
National Institute of Advanced Industrial Science and Technology, Tokyo, Japan

Jérémy Dubut
National Institute of Advanced Industrial Science and Technology, Tokyo, Japan

Abstract

Directed sets are of fundamental interest in domain theory and topology. In this paper, we formalize some results on directed sets in Isabelle/HOL, most notably: under the axiom of choice, a poset has a supremum for every directed set if and only if it does so for every chain; and a function between such posets preserves suprema of directed sets if and only if it preserves suprema of chains. The known pen-and-paper proofs of these results crucially use uncountable transfinite sequences, which are not directly implementable in Isabelle/HOL. We show how to emulate such proofs by utilizing Isabelle/HOL's ordinal and cardinal library. Thanks to the formalization, we relax some conditions for the above results.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Theory of computation → Denotational semantics

Keywords and phrases Directed Sets, Completeness, Scott Continuous Functions, Ordinals, Isabelle/HOL

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.34

Funding This work was supported by JST, CREST Grant Number JPMJCR22M1, Japan.

1 Introduction

A *directed set* is a set D equipped with a binary relation $\sqsubseteq$ such that any finite subset $X \subseteq D$ has an upper bound in D with respect to $\sqsubseteq$. The property is often equivalently stated that D is non-empty and any two elements $x, y \in D$ have a bound in D, assuming that $\sqsubseteq$ is transitive (as in posets).

Directed sets find uses in various fields of mathematics and computer science. In topology (see for example the textbook [8]), directed sets are used to generalize the set of natural numbers: sequences $\mathbb{N} \to A$ are generalized to *nets* $D \to A$, where D is an arbitrary directed set. For example, the usual result on metric spaces that continuous functions are precisely functions that preserve limits of sequences can be generalized in general topological spaces as: the continuous functions are precisely functions that preserve limits of nets. In domain theory [1], key ingredients are *directed-complete posets*, where every directed subset has a supremum in the poset, and *Scott-continuous functions* between posets, that is, functions that preserve suprema of directed sets. Thanks to their fixed-point properties (which we have formalized in Isabelle/HOL in a previous work [6]), directed-complete posets naturally appear in denotational semantics of languages with loops or fixed-point operators (see for example Scott domains [13, 15]). Directed sets also appear in reachability and coverability analyses of transition systems through the notion of ideals, that is, downward-closed directed sets. They allow effective representations of objects, making forward and backward analysis of well-structured transition systems – such as Petri nets – possible (see e.g., [7]).

Apparently milder generalizations of natural numbers are chains (totally ordered sets) or even well-ordered sets. In the mathematics literature, the following results are known (assuming the axiom of choice):


© Akihisa Yamada and Jérémy Dubut;
licensed under Creative Commons License CC-BY 4.0

▶ **Theorem 1** ([5]). *A poset is directed-complete if (and only if) it has a supremum for every non-empty well-ordered subset.*

▶ **Theorem 2** ([10]). *Let f be a function between posets, each of which has a supremum for every non-empty chain. If f preserves suprema of non-empty chains, then it is Scott-continuous.*

The pen-and-paper proofs of these results use induction on cardinality, where the finite case is merely the base case. The core of the proof is a technical result called Iwamura's Lemma [9], where the countable case is merely an easy case, and the main part heavily uses transfinite sequences indexed by uncountable ordinals.

In this paper, we formalize these results in the proof assistant Isabelle/HOL [11]. We extensively use the existing library for ordinals and cardinals in Isabelle/HOL [4], but we needed some delicate work in emulating the pen-and-paper proofs. In Isabelle/HOL, or any proof assistant based on higher-order logic (HOL), it is not possible to have a datatype for arbitrarily large ordinals; hence, it is not possible to directly formalize transfinite sequences. We show how to emulate transfinite sequences using the ordinal and cardinal library [4]. As far as the authors know, our work is the first to mechanize the proof of Theorems 1 and 2, as well as Iwamura's Lemma. We prove the two theorems for quasi-ordered sets, relaxing antisymmetry, and strengthen Theorem 2 so that chains are replaced by well-ordered sets and conditions on the codomain are completely dropped.

Related Work

Systems based on Zermelo-Fraenkel set theory, such as Mizar [2, 3] and Isabelle/ZF [12], have more direct support for ordinals and cardinals and should pose less challenge in mechanizing the above results. Nevertheless, a part of our contribution is in demonstrating that the power of (Isabelle/)HOL is strong enough to deal with uncountable transfinite sequences.

Except for the extra care for transfinite sequences, our proof of Iwamura's Lemma is largely based on the original proof from [9]. Markowsky presented a proof of Theorem 1 using Iwamura's Lemma [10, Corollary 1]. While he took a minimal-counterexample approach, we take a more constructive approach to build a well-ordered set of suprema. This construction was crucial to be reused in the proof of Theorem 2, which Markowsky claimed without a proof [10]. Another proof of Theorem 1 can be found in [5], without using Iwamura's Lemma, but still crucially using transfinite sequences.

Outline

The paper is organized as follows. In Section 2, we recall some basic concepts of order theory, ordinals, and cardinals, as well as their prior formalizations [4, 6]. In Section 3, we tackle the main formalization work of Iwamura's Lemma. The axiom of choice plays two crucial roles in the proof: first to obtain a well-ordering of a given set, and then to pick an upper bound for every finite subset. Finally, we use induction on directed sets – enabled by Iwamura's Lemma – to prove the equivalence between directed-completeness and well-completeness (Section 4), and the equivalence between Scott-continuity and preservation of suprema of chains (Section 5).

The formalization is available in the development version of the Archive of Formal Proofs as entry `Directed_Sets`, consisting of 726 lines of Isabelle code in total. The work also involves refactoring of our previous AFP entry `Complete_Non_Orders`[1] for reformulating continuity, completeness, well-foundedness and directed sets. The most changes are found in the new files `Continuity.thy` and `Directedness.thy` (427 lines).

2 Preliminaries

We assume some familiarity with Isabelle/HOL and use its notations also in mathematical formulas in the paper. We refer interested readers to the textbook [11] for more detail. Logical implication is denoted by $\Longrightarrow$ or $\longrightarrow$. We use *meta-equality* $\equiv$ to introduce definitions and abbreviations. By $X :: 'a\ set$ we denote a set X whose elements are of type $'a$, and $R :: 'a \Rightarrow 'a \Rightarrow bool$ is a binary predicate defined over $'a$. Type annotations "$:: \_$" are omitted unless necessary. The application of a function f to an element x is written $f\ x$, and the image of a set X under f is $f\ `\ X$. The power set of X is denoted by $Pow\ X$.

2.1 Binary Relations

In our previous Isabelle/HOL formalization on binary relations [6], some notations and properties of relations are defined as *locales*. Another approach is to use Isabelle's *type class* mechanism, which fixes a relation $\leq$ for each type so that one do not have to specify the relation of concern as a parameter. The drawback of the class-based approach is that one must use this relation $\leq$, which is too restrictive in the current development where we want to use *some* well-ordering of a given set.

To illustrate the use of locales, we revisit some definitions we need for the current paper. By *related set* we mean a set A with a binary relation (predicate) *less_eq* defined on A, denoted by infix symbol $\sqsubseteq$. In Isabelle:

```
locale related_set =
  fixes A :: 'a set and less_eq :: 'a ⇒ 'a ⇒ bool (infix ⊑ 50)
```

Then *reflexivity* and *transitivity* are defined as locales by making corresponding assumptions as follows:

```
locale reflexive = related_set + assumes x ∈ A ⟹ x ⊑ x
```

```
locale transitive = related_set +
  assumes x ⊑ y ⟹ y ⊑ z ⟹ x ∈ A ⟹ y ∈ A ⟹ z ∈ A ⟹ x ⊑ z
```

Then *quasi-ordered sets* are defined as the combination of reflexivity and transitivity:

```
locale quasi_ordered_set = reflexive + transitive
```

In this paper, we may use terminologies assuming that the right side of $\sqsubseteq$ is "greater", and use $\sqsupseteq$ to denote the dual of $\sqsubseteq$, though the notation is not always available in the actual Isabelle code. An (upper) *bound* of a set X is formalized by

```
definition bound X (⊑) b ≡ ∀x ∈ X. x ⊑ b for r (infix ⊑ 50)
```

[1] `https://www.isa-afp.org/entries/Complete_Non_Orders.html`

Dually, *bound* X $(\sqsupseteq)$ b specifies a lower bound. A *greatest (extreme)* element in X is a bound which is also in X:

definition *extreme* X $(\sqsubseteq)$ $e \equiv e \in X \wedge (\forall x \in X.\ x \sqsubseteq e)$ **for** r (**infix** $\sqsubseteq$ *50*)

Dually, *extreme* X $(\sqsupseteq)$ e specifies a least element. The following generalization of well-ordered sets frequently appears in this paper:

locale *well_related_set* = *related_set* +
  assumes $X \subseteq A \Longrightarrow X \neq \{\} \Longrightarrow \exists e.$ *extreme* X $(\sqsupseteq)$ e

that is, a set A together with a relation $\sqsubseteq$ such that every non-empty subset of A has a least element for $\sqsubseteq$. It can be also rephrased as the well-foundedness of the negation of $\sqsubseteq$. A well-related set is necessarily reflexive, which can be formalized by a sublocale statement:

sublocale *well_related_set* $\subseteq$ *reflexive*...

A *well-ordered* set is a well-related set where $\sqsubseteq$ is also antisymmetric (or equivalently a total order). A *pre-well-ordered* set is a well-related set which is also a quasi-order.

2.2 Ordinals and Cardinality Library

Here we briefly recap the ordinal and cardinality library [4] of Isabelle/HOL.

The library chooses the *set-oriented* formulation of relations: type *'a rel* is a shorthand for (*'a* × *'a*) *set*, and proposition $(x,y) \in R$ denotes that x and y are in relation $R ::$ *'a rel*.

An *order embedding* of a relation $(A, \sqsubseteq)$ into $(B, \trianglelefteq)$ is a function $f : A \to B$ such that $x \sqsubseteq y \iff f\ x \trianglelefteq f\ y$. The polymorphic relation $\leq o ::$ *'a rel* ⇒ *'b rel* ⇒ *bool* over binary relations is defined by $R \leq o\ S$ if and only if there is an order embedding from R to S. Two relations $R ::$ *'a rel* and $S ::$ *'b rel* are *order isomorphic*, $R =o\ S$, if $R \leq o\ S$ and $S \leq o\ R$.

One of the important results from the ordinal library is that $<o$, the asymmetric part of $\leq o$ (defined by $x <o\ y \equiv x \leq o\ y \wedge \neg\ y \leq o\ x$), seen as a relation over the same type, is well-founded. In fact, $\leq o$ forms a pre-well-order.

Conceptually, an ordinal can be seen as the equivalence class of well-orderings which are order isomorphic to each other. In Isabelle/HOL, or in any other HOL-based systems, it is not possible to have a set collecting well-orderings of different types. It is hence not possible to have a type for general ordinals in Isabelle/HOL. Instead, any well-ordering of any type is used to represent an ordinal in [4].

The *cardinality* of a set X is the least ordinal that is bijective with X. In Isabelle/HOL, $|X| ::$ *'a rel* is defined as *one of the* well-orderings on $X ::$ *'a set* which are least with respect to $\leq o$; there are well-orderings on X thanks to the well-order theorem (which is in turn due to the axiom of choice), and there are least ones since $\leq o$ is a pre-well-order.

3 Iwamura's Lemma

The main idea for proving Theorem 1 is, given a directed set D, to construct a well-ordered set whose supremum (which exists by assumption) is also a supremum for D. The difficulty is that the usual methods to construct a well-ordered set, such as Zorn's lemma, fail to achieve this goal. The crucial idea brought by Markowsky [10, Corollary 1] is that this well-ordered set can be obtained by a transfinite induction on the cardinality of the directed set, using Iwamura's Lemma [9]. Concretely, Iwamura's Lemma states the following:

▶ **Theorem 3.** *Let $(A, \sqsubseteq)$ be a reflexive directed set. If A is infinite, then there exists a transfinite sequence $\{I_\alpha\}_{\alpha<|A|}$ of subsets of A that satisfies the following four conditions:*

- *directedness: I_α is directed for all $\alpha < |A|$,*
- *cardinality: $|I_\alpha| < |A|$ for all $\alpha < |A|$,*
- *monotonicity: $I_\alpha \subseteq I_\beta$ whenever $\alpha \leq \beta < |A|$, and*
- *range: $\bigcup_{\alpha<|A|} I_\alpha = A$.*

Note that, if we drop directedness, then the statement is equivalent to the well-ordering theorem. The main point of Iwamura's Lemma is that one can extend any subset of a directed set into a directed one without changing the cardinality.

As in the original statement, $\sqsubseteq$ need not be transitive. Hence, directedness is formalized as follows:

```
definition directed_set A (⊑) ≡ ∀X ⊆ A. finite X ⟶ (∃b ∈ A. bound X (⊑) b)
  for less_eq (infix ⊑ 50)
```

As the proof involves a number of (inductive) definitions, we build a **locale** for collecting those definitions and lemmas.

```
locale Iwamura_proof = related_set +
  assumes dir: directed_set A (⊑)
begin
```

Inside this locale, a related set $(A, \sqsubseteq)$ is fixed and assumed to be directed. The proof starts with declaring, using the axiom of choice, a function f that chooses a bound $f\ X \in A$ for every finite subset $X \subseteq A$. This function can be formalized using the *SOME* construction:

```
definition f where f X ≡ SOME x. x ∈ A ∧ bound X (⊑) x
```

In Isabelle, *SOME* $x.\ \phi\ x$ takes *some* value x that satisfies the condition $\phi\ x$, if such a value exists; otherwise it takes an unspecified value. As we assume that any finite subset $X \subseteq A$ has an upper bound in A, we can prove that f satisfies the following specification:

```
lemma assumes X ⊆ A and finite X
  shows f X ∈ A and bound X (⊑) (f X) ...
```

After obtaining this f, the proof constructs $\{I_\alpha\}_{\alpha<|A|}$ depending crucially on whether A is countably or uncountably infinite.

3.1 Uncountable Case

We start with the core case, where A is uncountable. The original proof goes as follows: Thanks to the well-order theorem, one can have a sequence $\{A_\alpha\}_{\alpha<|A|}$ of subsets of A that satisfies the following three conditions:

- cardinality: $|A_\alpha| < |A|$ for every $\alpha < |A|$,
- monotonicity: $A_\alpha \subseteq A_\beta$ whenever $\alpha \leq \beta < |A|$, and
- range: $A = \bigcup_{\alpha<|A|} A_\alpha$.

Then it is shown that any subset of A, in particular A_α, can be monotonically extended to a directed one I_α, such that $|I_\alpha| \leq |A_\alpha| \cdot \aleph_0$. Since $|A_\alpha| < |A|$ and $|A|$ is uncountable, it follows that $|I_\alpha| < |A|$.

In order to formalize the above argument in Isabelle/HOL, one of the challenges is that we do not have a datatype for ordinals (that works for arbitrary types of A), and thus one cannot formalize transfinite sequences as functions from ordinals.

3.1.1 Formalizing Transfinite Sequences

As we cannot formalize transfinite sequences directly, we take the following approach: We just use A as the index set, and instead of the ordering on ordinals, we take the well-order $(\preceq_A)$ that is chosen by the cardinality library to denote $|A|$, as follows:

definition ... **where** $(\preceq_A)\ x\ y \equiv (x,y) \in |A|$

Recall that $|A|$ is defined as *one of the* well-orders on A which are least with respect to $\leq o$, in a set-oriented formulation of relations. We also introduce infix notations for $\preceq_A$ and its asymmetric part $\prec_A$ as follows:

abbreviation ... **where** $x \preceq_A y \equiv (\preceq_A)\ x\ y$
abbreviation ... **where** $x \prec_A y \equiv \mathit{asympartp}\ (\preceq_A)\ x\ y$

Now we show that $A_\prec : A \to \mathit{Pow}\ A$ serves the purpose of $\{A_\alpha\}_{\alpha<|A|}$ above, where

definition ... **where** $A_\prec\ a \equiv \{x \in A.\ x \prec_A a\}$

First, we prove the counterpart of the cardinality condition $|A_\alpha| < |A|$.

lemma *Pre_card*: **assumes** $a \in A$ **shows** $|A_\prec\ a| <o |A|$

Proof. On pen and paper, one would first well-order A as $\{a_\alpha\}_{\alpha<|A|}$ and chose $A_\alpha = \{a_\beta\}_{\beta<\alpha}$; then $|A_\alpha| < |A|$ would look obvious. Note that there is an implicit use of the fact that $|A|$ is least; otherwise $\alpha < |A|$ and $|\{a_\beta\}_{\beta<\alpha}| = |A|$ is possible.

In the formalization, we derive this fact by connecting to the cardinality library. In fact, $A_\prec\ a$ corresponds precisely to *underS* $|A|\ a$ in terms of the library. Then lemma *card_of_underS* from the library easily concludes the lemma. ◀

Second, the monotonicity condition, $A_\alpha \subseteq A_\beta$ whenever $\alpha \leq \beta$, is easy:

lemma *Pre_mono*: *monotone_on* $A\ (\preceq_A)\ (\subseteq)\ (A_\prec)$ ...

The final property we need is $\bigcup_{\alpha<|A|} A_\alpha = A$. This is not as easy as the previous two properties; note that it cannot hold for finite A. We first prove that if the well-ordering $(A, \preceq_A)$ has a greatest element, then A must be finite:

lemma *extreme_imp_finite*: **assumes** *extreme* $A\ (\preceq_A)\ e$ **shows** *finite* A

Proof. Since e is greatest in A, we have $A_\prec\ e = A - \{e\}$. On the other hand, $|A - \{e\}| =o |A|$ if A is infinite. This cannot happen due to Lemma *Pre_card*. ◀

This allows us to prove the desired property:

lemma *infinite_imp_Un_Pre*: **assumes** *infinite* A **shows** $\bigcup(A_\prec\ `\ A) = A$

Proof. The inclusion $A_\prec\ `\ A \subseteq A$ is obvious. For the other direction, consider $a \in A$. Due to Lemma *extreme_imp_finite*, $a$ cannot be the greatest in $A$ with respect to $\preceq_A$. So there exists some $b \in A$ such that $a \prec_A b$. Hence $a \in A_\prec\ b \subseteq \bigcup(A_\prec\ `\ A)$. ◀

3.1.2 Expanding Infinite Sets into Directed Sets

Actually, the main part of the proof of Iwamura's Lemma is about monotonically expanding an infinite subset (in particular A_α) of A into a directed one, without changing the cardinality. To this end, Iwamura's original proof introduces a function $F : Pow\ A \to Pow\ A$ that expands a set with upper bounds of *all finite subsets*. This approach is different from Markowsky's reproof (based on [14]) which uses nested transfinite induction to extend a set one element after another.

definition F **where** $F\ X \equiv X \cup f\ `\ Fpow\ X$

Here, $Fpow\ X$ is an Isabelle/HOL notation for the set of finite subsets of X. Hence, for any finite subset Y of X, there is an upper bound $f\ Y$ in $F\ X$. We take the ω-iteration of the monotone function F, namely:

definition *Flim* (F^ω) **where** $F^\omega\ X \equiv \bigcup i.\ F^i\ X$

We prove that $\{F^\omega\ (A_\prec\ a)\}_{a\in A}$ serves the purpose of $\{I_\alpha\}_{\alpha<|A|}$ when A is uncountable.

Directedness condition is satisfied regardless of uncountability. More generally, $F^\omega\ X$ is directed for every $X \subseteq A$.

lemma *Flim_directed*: **assumes** $X \subseteq A$ **shows** *directed_set* $(F^\omega\ X)\ (\sqsubseteq)$

Proof. Take an arbitrary finite subset $Y \subseteq F^\omega\ X$. Since Y is finite, we inductively obtain $i \in \mathbb{N}$ such that $Y \subseteq F^i\ X$, i.e., $Y \in Fpow\ (F^i\ X)$. Hence we find an upper bound $f\ Y \in F^{i+1}\ X \subseteq F^\omega\ X$. ◀

The cardinality condition holds when $|A|$ is uncountable. Using the cardinality library, (un)countability is stated using the term *natLeq*, which denotes the well-order $(\mathbb{N}, \leq)$, i.e., the ordinal ω or cardinality $\aleph_0$.

lemma *card_uncountable*:
assumes $a \in A$ **and** $natLeq <o |A|$ **shows** $|F^\omega\ (A_\prec\ a)| <o |A|$

Proof. Let $X = A_\prec\ a$. The proof proceeds by case distinction on whether X is finite or not. If X is finite, then every $F^i\ X$ is finite and thus $F^\omega\ X$ is at most countable. Note that $F^\omega\ X$ is not necessarily finite. Nevertheless, since A is assumed to be uncountable, we conclude $|F^\omega\ X| <o |A|$.

Now we show that if X is infinite, then $|F^\omega\ X| =o |X|$. This will conclude the claim as $|X| <o |A|$ due to Lemma *Pre_card*. First, we have $|F\ X| =o |X|$. This is easy using the library fact *card_of_Fpow_infinite*: $infinite\ X \Longrightarrow |Fpow\ X| =o |X|$. Then this property is carried over to $|F^i\ X| =o |X|$ for every $i \in \mathbb{N}$, proved by an easy induction.

Now, the following fact (*card_of_UNION_ordLeq_infinite*) is available in the library:

$$infinite\ B \Longrightarrow |I| \leq o\ |B| \Longrightarrow \forall i \in I.\ |A\ i| \leq o\ |B| \Longrightarrow |\bigcup\ (A\ `\ I)| \leq o\ |B|$$

Since X is infinite, we know $|\mathbb{N}| \leq o\ |X|$, and we have proved that $|F^i\ X| \leq o\ |X|$ for all $i \in \mathbb{N}$. Thus, by taking $I = \mathbb{N}$, $A\ i = F^i\ X$, and $B = X$, we conclude $|F^\omega\ X| \leq o\ |X| <o |A|$. Since $X \subseteq F^\omega\ X$, we also have $|F^\omega\ X| =o |X|$. ◀

Monotonicity is due to that of the building components:

lemma *mono_uncountable*: *monotone_on* $D\ (\preceq_A)\ (\subseteq)\ (F^\omega \circ A_\prec)$

Proof. As $A_\prec$ is monotone (Lemma *Pre_mono*) and monotonicity is preserved by composition, it suffices to show that F^ω is monotone. It is easy to see that F is monotone. Then so is F^i for every $i \in \mathbb{N}$, as i-th fold of a monotone function is still monotone. Finally, we conclude the monotonicity of F^ω by the following more general statement:

```
lemma Sup_funpow_mono:
  fixes f :: 'a :: complete_lattice ⇒ 'a
  assumes mono f shows mono (⨆i. f^i) ...
```

which is proved easily. ◀

Finally, for the range condition, the infiniteness of A is sufficient.

```
lemma range_uncountable: assumes infinite A shows ⋃((F^ω ∘ A_≺) ‘ A) = A
```

Proof. The $(\subseteq)$-direction is obvious. For the $(\supseteq)$-direction, take $a \in A$. As A is infinite, by **lemma** *extreme_imp_finite*, we obtain $b \in A$ such that $a \in A_\prec\, b$. By definition, $X \subseteq F\ X$. By induction, $X \subseteq F^\omega\ X$. We conclude $a \in A_\prec\ b \subseteq F^\omega\ (A_\prec\ b) \subseteq \bigcup((F^\omega \circ A_\prec) \text{ ‘ } A$. ◀

3.2 Countable Case

Next we consider the case where A is countably infinite. We make the assumption by making a subcontext within the locale *Iwamura_proof*:

```
context
  assumes countable: |A| =o natLeq
begin
```

The assumption above means that there exists an order-isomorphism between $(\mathbb{N}, \leq)$ and $(A, \preceq_A)$. In Isabelle/HOL, we can obtain the isomorphism as follows:

```
definition seq :: nat ⇒ 'a where seq ≡ SOME g. iso natLeq |A| g
```

```
lemma seq_iso: iso natLeq |A| seq ...
```

The definition of the predicate *iso* is given in the ordinal library. For our use, it suffices to know a few consequences of *seq_iso*. Most importantly, *seq* is bijective between $\mathbb{N}$ and A:

```
lemma seq_bij_betw: bij_betw seq UNIV A
```

This means that A has been indexed by $\mathbb{N}$: $A = \{seq\ 0,\ seq\ 1,\ seq\ 2,\ \ldots\}$. We turn the sequence into a sequence of directed subsets of A: $Seq\ 0 \subseteq Seq\ 1 \subseteq Seq\ 2 \subseteq \ldots \subseteq A$.

```
fun Seq :: nat ⇒ 'a set where
  Seq 0 = {f {}}
| Seq (Suc n) = Seq n ∪ {seq n, f (Seq n ∪ {seq n})}
```

As *Seq* is a plain inductive function, it is an easy exercise to formally prove that $\{Seq\ n\}_{n\in\mathbb{N}}$ satisfies the four requirements of Iwamura's Lemma. A more interesting formalization work is in combining with the uncountable case. In Section 3.1, we took $F^\omega \circ A_\prec$ as the candidate of I, which is of type *'a ⇒ 'a set*. On the other hand, *Seq* is of type *nat ⇒ 'a set*. To match the types, we use the inverse seq^{-1} :: *'a ⇒ nat* (*inv seq* in the standard Isabelle notation) of the isomorphism *seq*. We define the final I as follows:

definition I **where** $I \equiv$ ***if*** $|A| =o\ natLeq$ ***then*** $Seq \circ seq^{-1}$ ***else*** $F^{\omega} \circ A_{\prec}$

Now we close the locale *Iwamura_proof* and state the final result in the global scope.

theorem (**in** *reflexive*) *Iwamura*:
assumes *directed_set* A $(\sqsubseteq)$ **and** *infinite* A
shows $\exists I.\ (\forall a \in A.\ directed\_set\ (I\ a)\ (\sqsubseteq) \wedge |I\ a| <o |A|\ ) \wedge$
$monotone\_on\ A\ (\preceq_A)\ (\subseteq)\ I \wedge \bigcup(I`A) = A$

Proof. Inside the proof we reopen the proof locale:

interpret *Iwamura_proof*...

By this we obtain I defined above. We conclude by proving that I satisfies the requirements.

- *directed_set* $(I\ a)$ $(\sqsubseteq)$: The uncountable case is by *Flim_directed*. For the countable case, we show that $Seq\ n$ is directed for every $n \in \mathbb{N}$. Note that $Seq\ n$ can be written $X \cup \{f\ X\}$ for appropriate X. Then since $f\ X$ is an upper bound of X and $\sqsubseteq$ is reflexive, $f\ X$ serves as an upper bound of any (finite) subset of $X \cup \{f\ X\}$.
- $|I\ a| <o |A|$: The uncountable case is by *card_uncountable*. For countable case, we just prove that $Seq\ n$ is finite for any $n \in \mathbb{N}$, by easy induction.
- *monotone_on* A $(\preceq_A)$ $(\subseteq)$ I: The uncountable case is by *mono_uncountable*. For the countable case, we need another consequence of lemma *seq_iso*:

  lemma *inv_seq_mono*: $monotone\_on\ A\ (\preceq_A)\ (\leq)\ (seq^{-1})$ ...

  We then combine with the monotonicity of *Seq*, which is easily proved by induction.
- $\bigcup(I\ `\ A) = A$: The uncountable case is by *range_uncountable*. For the countable case, we need to prove $\bigcup((Seq \circ seq^{-1})\ `\ A) = A$. The $(\subseteq)$-direction is obvious. For the other direction, take an arbitrary $a \in A$. We know $a = seq\ (seq^{-1}\ a) \in Seq\ n$ with $n = Suc\ (seq^{-1}\ a)$. On the other hand, $seq\ n \in A$. Hence $a \in Seq\ n = Seq\ (seq^{-1}\ (seq\ n)) \subseteq \bigcup(Seq \circ seq^{-1})\ `\ A$. ◀

4 Directed Completeness

Now we formalize Theorem 1: A quasi-ordered set has a supremum for every directed subset, if and only if it does so for every non-empty well-related subset. The statement is slightly generalized, so that the underlying order need not be antisymmetric.

The property that certain class of subsets have suprema is called *completeness*. We formalize completeness as follows:

definition ... **where**
$\mathcal{C}$-*complete* A $(\sqsubseteq) \equiv \forall X \subseteq A.\ \mathcal{C}\ X\ (\sqsubseteq) \longrightarrow (\exists s.\ extreme\_bound\ A\ (\sqsubseteq)\ X\ s)$
for *less_eq* (**infix** $\sqsubseteq$ *50*)

Using this notation, we can formalize Theorem 1 concisely as follows:

theorem (**in** *quasi_ordered_set*) *well_complete_iff_directed_complete*:
$(nonempty \sqcap well\_related\_set)$-*complete* A $(\sqsubseteq) \longleftrightarrow$ *directed_set-complete* A $(\sqsubseteq)$

where *nonempty* $A \equiv$ ***if*** $A = \{\}$ ***then*** $\bot$ ***else*** $\top$. For the ($\longleftarrow$)-direction we must prove that non-empty well-related sets are actually directed. Well-related sets clearly are *connex*, i.e., every two elements are comparable. Under transitivity this is sufficient for directedness, but we can actually prove a stronger statement without transitivity: every non-empty finite subset X of a well-related set A has a greatest element.

lemma (**in** *well_related_set*) *finite_sets_extremed*:
  assumes *finite* X **and** $X \neq \{\}$ **and** $X \subseteq A$
  shows *extremed* X ($\sqsubseteq$)

Proof. By induction on the number[2] of elements in the finite set X. As X is nonempty, by well-relatedness, it has a least element l. If $X - \{l\}$ is empty, then l is the greatest in $X = \{l\}$ by reflexivity. Otherwise, by induction hypothesis, $X - \{l\}$ has a greatest element e. As l is least in X and in particular $l \sqsubseteq e$, e is also greatest in X. ◀

For the ($\longrightarrow$)-direction, we prove the following elaborated statement:

lemma (**in** *quasi_ordered_set*) *directed_completeness_lemma*:
  assumes (*nonempty* $\sqcap$ *well_related_set*)*-complete* A ($\sqsubseteq$)
    and *directed_set* D ($\sqsubseteq$) **and** $D \subseteq A$
  shows $\exists x.$ *extreme_bound* A ($\sqsubseteq$) D x

Proof. We apply induction on the cardinality $|D|$ with respect to $<o$. To be more precise, we are given fresh D for which we must prove ϕ D, where ϕ X denotes

$$\textit{directed\_set}\ X\ (\sqsubseteq) \Longrightarrow X \subseteq A \Longrightarrow \exists x.\ \textit{extreme\_bound}\ A\ (\sqsubseteq)\ X\ x$$

assuming ϕ D' for any D' with $|D'| <o |D|$.

If D is finite, then D has an upper bound of itself, i.e., a greatest element, which serves also as a supremum. So suppose that D is infinite. For this D, we apply Iwamura's Lemma and obtain I as follows.

  obtain I **where** *monotone_on* D ($\preceq_D$) ($\subseteq$) I
    and $\forall a \in D.\ |I\ a| <o |D|$
    and $\forall a \in D.$ *directed_set* (I a) ($\sqsubseteq$)
    and $\bigcup (I\ `\ D) = D$ ...

For every $d \in D$, since $|I\ d| <o |D|$, induction hypothesis ensures that $I\ d$ has a supremum in A. Thus, using the axiom of choice, we obtain a function s that picks a supremum for $I\ d$. Note that as we do not assume that $\sqsubseteq$ is antisymmetric, suprema are not unique so the axiom of unique choice cannot be used.

  obtain s **where** $d \in D \Longrightarrow$ *extreme_bound* A ($\sqsubseteq$) (I d) (s d) **for** d ...

Next we show that $(s\ `\ D, \sqsubseteq)$ is well-related. To this end, we formalized the following fact: monotone image of a well-related set is well-related.

  lemma (**in** *well_related_set*) *monotone_image_well_related*:
    fixes *leB* (**infix** $\trianglelefteq$ *50*)
    assumes *monotone_on* A ($\sqsubseteq$) ($\trianglelefteq$) f **shows** *well_related_set* (f `` A) ($\trianglelefteq$) ...

[2] In Isabelle, *card* X is used to denote the number of elements in X, assuming that X is finite. In contrast, $|X|$ is the cardinality in more general sense.

So now we need that s is monotone from $(D, \preceq_D)$ to $(A, \sqsubseteq)$. This follows as I is monotone from $(D, \preceq_D)$ to $(Pow\ D, \subseteq)$, and taking suprema is monotone from $(Pow\ D, \subseteq)$ to $(A, \sqsubseteq)$. This concludes that $(s\ `\ D, \sqsubseteq)$ is well-related. Since $D$ is infinite and thus non-empty, thanks to the completeness assumption we obtain a supremum $x$ of $s\ `\ D$. We conclude by showing that x is also a supremum of D.

To show that x is a bound of D, consider an arbitrary $d \in D$. Since $D = \bigcup (I\ `\ D)$, we obtain $d' \in D$ such that $d \in I\ d'$. As $s\ d'$ is a supremum of $I\ d'$, we know $d \sqsubseteq s\ d'$. Since $s\ d' \in s\ `\ D$ and x is a supremum of $s\ `\ D$, we have $s\ d' \sqsubseteq x$. By transitivity we conclude $d \sqsubseteq x$.

Finally, let b be another bound of D. For any $d \in D$, since $I\ d \subseteq D$, b is a bound of $I\ d$. Since $s\ d$ is least among the bounds of $I\ d$, we have $s\ d \sqsubseteq b$. This shows that b is a bound of $s\ `\ D$. Since $x$ is least among the bounds of $s\ `\ D$, we conclude $x \sqsubseteq b$. ◀

5 Scott-Continuity

The previous inductive proof can be strengthened to prove and generalize Theorem 2: A function that preserves suprema of well-related subsets also preserves suprema of directed subsets, if the domain has a supremum for every nonempty well-related sets. Markowsky claimed Theorem 2 [10, Corollary 3], saying briefly that it follows from Iwamura's Lemma and transfinite induction. We did not find it that obvious (at least for mechanization), and by completing the proof, we could slightly generalize Markowsky's claim. Now it works for quasi-ordered domain, relaxing antisymmetry; the codomain need not be complete in any class, or even transitivity or reflexivity are not necessary; and chains are refined to well-related sets.

Functions that preserve a particular class of suprema are called *continuous*. We formalize the notion in Isabelle as follows:

```
definition ... where
  𝒞-continuous A (⊑) B (≼) f ≡ f ` A ⊆ B ∧
   (∀X s. 𝒞 X (⊑) ⟶ X ≠ {} ⟶ X ⊆ A ⟶
     extreme_bound A (⊑) X s ⟶ extreme_bound B (≼) (f ` X) (f s))
  for leA (infix ⊑ 50) and leB (infix ≼ 50)
```

A useful fact about continuous functions, is that, under a mild condition on the class $\mathcal{C}$ – namely, all pairs of related elements are in the class – every $\mathcal{C}$-continuous function is monotone:

```
lemma (in reflexive) continuous_imp_monotone_on:
  assumes 𝒞-continuous A (⊑) B (≼) f and ∀i ∈ A. ∀ j ∈ A. i ⊑ j ⟶ 𝒞 {i,j} (⊑)
  shows monotone_on A (⊑) (≼) f ...
```

This is the case for *well_related_set*-continuous functions.

The Isabelle statement of Theorem 2 then becomes:

```
theorem (in quasi_ordered_set)
  assumes (nonempty ⊓ well_related_set)-complete A (⊑)
  shows well_related_set-continuous A (⊑) B (≼) f ⟷ directed_set-continuous A (⊑) B
(≼) f
```

As before, the ($\longleftarrow$)-direction is obvious. For the ($\longrightarrow$)-direction, our strategy is to prove that f preserves the suprema of every directed set, at the same time we construct the suprema

in the previous section. Precisely, into the statement of **lemma** *directed_completeness_lemma* we add the following claim:

and *well_related_set-continuous* A $(\sqsubseteq)$ B $(\trianglelefteq)$ $f \Longrightarrow$
$D \neq \{\} \Longrightarrow$ *extreme_bound* A $(\sqsubseteq)$ D $x \Longrightarrow$ *extreme_bound* B $(\trianglelefteq)$ $(f \mathbin{`} D)$ $(f\ x)$

Proof. The claim is proved simultaneously with the previous statement by induction on $|D|$. Our new goal is to show, given a supremum x of D in $(A, \sqsubseteq)$, that $f\ x$ is a supremum of $f \mathbin{`} D$ in $(B, \trianglelefteq)$.

By monotonicity, $f\ x$ is a bound of $f \mathbin{`} D$, so we show that it is least of such. Recall that, in the previous section, a supremum of $D$ is obtained as a supremum of a well-related set $C$, where $C$ is a singleton set in the finite case, and is $s \mathbin{`} D$ in the infinite case. Note that, as we do not assume antisymmetry, this supremum is not necessarily the supremum x we are given. Nevertheless, we know that x is also a supremum of C, thanks to the transitivity of $(A, \sqsubseteq)$. As f preserves suprema of well-related sets, we also know that $f\ x$ is a supremum of $f \mathbin{`} C$ in $(B, \trianglelefteq)$. Hence, by showing that any bound $b$ of $f \mathbin{`} D$ is also a bound of $f \mathbin{`} C$, we can show $f\ x \trianglelefteq b$ and conclude the proof.

The finite case is obvious as $C \subseteq D$. Consider the infinite case: $C = s \mathbin{`} D$. We know that $b$ is a bound of $f \mathbin{`} I\ d$ for every $d \in D$, as $D = \bigcup (I \mathbin{`} D)$. Recall that, in the previous section, $s\ d$ is an inductively obtained supremum of $I\ d$. With $|I\ d| <o |D|$, by induction hypothesis we know that $f\ (s\ d)$ is a supremum of $f \mathbin{`} I\ d$. In particular $f\ (s\ d) \trianglelefteq b$, concluding that b is a bound of $f \mathbin{`} s \mathbin{`} D = f \mathbin{`} C$. ◀

6 Conclusion

In this paper, we formalized some results for directed sets: Iwamura's Lemma to enable induction arguments on them; Cohn's theorem stating the equivalence between directed-completeness and well-completeness; and Markowski's corollary on Scott-continuity being equivalent to the preservation of suprema of well-related chains. The proofs involved some non-trivial formalization work on transfinite sequences that has been enabled by a careful management of locales and contexts, and Isabelle/HOL's libraries on cardinals and ordinals.

References

1 Samson Abramsky and Achim Jung. *Domain Theory*. Number III in Handbook of Logic in Computer Science. Oxford University Press, 1994.

2 Grzegorz Bancerek. The ordinal numbers. *Journal of Formalized Mathematics*, 1, 1989.

3 Grzegorz Bancerek and Piotr Rudnicki. A compendium of continuous lattices in MIZAR. *J. Autom. Reason.*, 29(3-4):189–224, 2002. doi:10.1023/A:1021966832558.

4 Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. Cardinals in Isabelle/HOL. 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 111–127. Springer, 2014. doi:10.1007/978-3-319-08970-6_8.

5 Paul M. Cohn. *Universal Algebra*. Harper & Row, 1965.

6 Jérémy Dubut and Akihisa Yamada. Fixed point theorems for non-transitive relations. *Log. Methods Comput. Sci.*, 18(1), 2022. doi:10.46298/lmcs-18(1:30)2022.

7 Alain Finkel and Jean Goubault-Larrecq. Forward Analysis for WSTS, Part I: Completions. In Susanne Albers and Jean-Yves Marion, editors, *26th International Symposium on Theoretical Aspects of Computer Science*, volume 3 of *Leibniz International Proceedings in Informatics*

(LIPIcs), pages 433–444, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.STACS.2009.1844.

8 Jean Goubault-Larrecq. *Non-Hausdorff Topology and Domain Theory: Selected Topics in Point-Set Topology*, volume 22 of *New Mathematical Monographs*. Cambridge University Press, 2013. doi:10.1017/CBO9781139524438.

9 Tsurane Iwamura. A lemma on directed sets. *Zenkoku Shijo Sugaku Danwakai*, 262:107–111, 1944. in Japanese.

10 George Markowsky. Chain-complete posets and directed sets with applications. *Algebra Universalis*, 6:53–68, 1976.

11 Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. doi:10.1007/3-540-45949-9.

12 Lawrence C. Paulson and Krzysztof Grabczewski. Mechanizing set theory. *J. Autom. Reason.*, 17(3):291–323, 1996. doi:10.1007/BF00283132.

13 Dana Scott. Outline of a Mathematical Theory of Computation. Technical Report PRG02, OUCL, 1970.

14 Lev Anatol'evich Skornyakov. *Complemented modular lattices and regular rings*. Oliver & Boyd, 1964.

15 Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing. The MIT Press, 1993.

Formalising the Proj Construction in Lean

Jujian Zhang
Department of Mathematics, Imperial College London, UK

Abstract

Many objects of interest in mathematics can be studied both analytically and algebraically, while at the same time, it is known that analytic geometry and algebraic geometry generally do not behave the same. However, the famous GAGA theorem asserts that for projective varieties, analytic and algebraic geometries are closely related; the proof of Fermat's last theorem, for example, uses this technique to transport between the two worlds [13]. A crucial step of proving GAGA is to calculate cohomology of projective space [12, 8], thus I formalise the Proj construction in the Lean theorem prover for any $\mathbb{N}$-graded R-algebra A and construct projective n-space as $\operatorname{Proj} A[X_0, \ldots, X_n]$. This is the first family of non-affine schemes formalised in any theorem prover.

2012 ACM Subject Classification Theory of computation → Logic and verification; Mathematics of computing → Topology

Keywords and phrases Lean, formalisation, algebraic geometry, scheme, Proj construction, projective geometry

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.35

Supplementary Material *Software (Source Code)*: `https://github.com/leanprover-community/mathlib/pull/18138/commits/00c4b0918a2c7a8b62291581b0e1eddf2357b5be`
archived at `swh:1:dir:0876b1af377d6c78a3d76073c0bbd7fe0176d9c6`

Funding *Jujian Zhang*: Schrödinger Scholarship Scheme.

Acknowledgements I want to thank Eric Wieser for his contribution and suggestion in formalising homogeneous ideals and homogeneous localisation; Andrew Yang and Junyan Xu for their review and comments on my code; Kevin Buzzard for suggesting this project and all the contributors to mathlib for otherwise this project would not have been possible.

1 Introduction

Algebraic geometry concerns polynomials and analytic geometry concerns holomorphic functions. Though all polynomials are holomorphic, the converse is not true; thus many analytic objects are not algebraic, for example, $\{x \in \mathbb{C} \mid \sin(x) = 0\}$ can not be defined as the zero locus of a polynomial in one variable, for polynomials always have only finite number of zeros. However, for projective varieties over $\mathbb{C}$, the categories of algebraic and analytic coherent sheaves are equivalent; a consequence of this statement is that all closed analytic subspace of projective n-space $\mathbb{P}_n$ is also algebraic [13, 4]. A crucial step in proving the above statement is to consider the cohomology of projective n-space $\mathbb{P}_n$ [12].

While one can define $\mathbb{P}_n$ over $\mathbb{C}$ without consideration of other projective varieties, it would be more fruitful to formalise the Proj construction as a **scheme** and recover $\mathbb{P}_n$ as $\operatorname{Proj} \mathbb{C}[X_0, \ldots, X_n]$, since, among other reasons, by considering different base rings, one may obtain different projective varieties, for example, for any homogeneous polynomials $f_1, \ldots, f_k$, $\operatorname{Proj}\left(\frac{\mathbb{C}[X_0,\ldots,X_n]}{(f_1,\ldots,f_k)}\right)$ defines a projective variety over $\mathbb{C}$.

In this paper I describe a formal construction of $\operatorname{Proj} A$ in the Lean3 theorem prover [7] by closely following [9, Chapter II]. The formal construction uses various results from the Lean mathematical library mathlib, most notably the graded algebra and Spec construction; this project has been partly accepted into mathlib already while the remaining part is still

© Jujian Zhang;
licensed under Creative Commons License CC-BY 4.0

undergoing a review process. The code discussed in this paper can be found on GitHub[1]. I have freely used the axiom of choice and the law of excluded middle throughout this project since the rest of mathlib freely uses classical reasoning as well; consequently, the final construction is not computable. This will not matter for the applications in mind, for example calculating sheaf cohomology and the GAGA theorem.

As previously mentioned, Proj construction heavily depends on graded algebras and the Spec construction. A detailed description of graded algebra in Lean and mathlib, as well as a comparison of graded algebras with that in other theorem provers, can be found in [17]; for my purpose, I have chosen to use an internal grading for any graded ring $A \cong \bigoplus \mathcal{A}_i$ so that the result of the construction is about homogeneous prime ideals of A directly instead of $\bigoplus_i \mathcal{A}_i$. The earliest complete Spec construction in Lean can be found in [2] where the construction followed a "sheaf-on-a-basis" approach from [14, Section 01HR], however, it differs significantly from the Spec construction currently found in mathlib where the construction follows [9, Chapter II]; for this reason, I have also chosen to follow the definition in [9, Chapter II]. Some other theorem provers also have or partially have the Spec construction: in Isabelle/HOL, Spec is formalised by using locales and rewriting topology and ring theory part of the existing library in [1], however, the category of schemes is yet to be formalized; an early formalisation of Spec in Coq can be found in [3] and a definition of schemes in general can be found in its `UniMath` library [16]; due to homotopy type theory of `Agda`, only a partial formalisation of Spec construction can be found in [11]. Though some theorem provers have defined a general scheme, I could not find any concrete construction of a scheme other than Spec of a ring[2]. Thus this paper exhibits the first concrete formalised example of non-affine scheme.

After explaining the mathematical details involved in the Proj construction in Section 2, Lean code will be provided and explained in Section 3. For typographical reasons, some code of formalisation will be omitted and marked as `omitted` or `_` and some code presented in this paper is presented with shortened notations for presentability and readability.

2 Mathematical details

In this section, certain familiarity with basic ring theory, topology and category theory will be assumed. In Sections 2.1 and 2.2, definition of a scheme is explained in detail; Spec construction will also be briefly explained to fix the mathematical approach used in mathlib. Then by following the definition of a scheme step by step, the Proj construction will be explained in Section 2.3.

2.1 Sheaves and Locally Ringed Spaces

Let X be a topological space and $\mathfrak{Opens}(X)$ be the category of open subsets of X.

▶ **Definition 1** (Presheaves [10]). *Let C be a category. A C-valued presheaf $\mathcal{F}$ on X is a functor $\mathfrak{Opens}(X)^{\mathsf{op}} \Longrightarrow C$. Morphisms between C-valued presheaves $\mathcal{F}, \mathcal{G}$ are natural transformations. The category thus formed is denoted as $\mathfrak{PSh}(X, C)$.*

In this paper, the category of interest is the category of presheaves of rings $\mathfrak{Psh}(X, \mathfrak{Ring})$. More explicitly, a presheaf of rings $\mathcal{F}$ assigns to each open subset $U \subseteq X$ a ring $\mathcal{F}(U)$ whose elements are called sections on U and for any open subsets $U \subseteq V \subseteq X$, $\mathcal{F}$ assigns

[1] url: `https://github.com/leanprover-community/mathlib/pull/18138/`
[2] In this paper, all rings are assumed to be unital and commutative.

a ring homomorphism $\mathcal{F}(V) \to \mathcal{F}(U)$ often denoted as res^V_U or simply with a vertical bar $s\,|_U$ (a section s on V restricted to U). Examples of presheaves of rings are abundant: considering open subsets of $\mathbb{C}$, $U \mapsto \{\text{(continuous, holomorphic) functions on } U\}$ with the natural restriction map defines a presheaf of rings. In these examples, compatible sections on different open subsets can be glued together to form bigger sections on the union of the said open subsets; this property can be generalized to arbitrary categories:

▶ **Definition 2** (Sheaves [10, 14]). *A presheaf $\mathcal{F} \in \mathfrak{Psh}(X, C)$ is said to be a* sheaf *if for any open covering of an open set $U = \bigcup_i U_i \subseteq X$, the following diagram is an equalizer*

$$\mathcal{F}(U) \xrightarrow{\left(\mathrm{res}^U_{U_i}\right)} \prod_i \mathcal{F}(U_i) \underset{\left(\mathrm{res}^{U_j}_{U_i \cap U_j}\right)}{\overset{\left(\mathrm{res}^{U_i}_{U_i \cap U_j}\right)}{\rightrightarrows}} \prod_{i,j} F(U_i \cap U_j).$$

The category of sheaves $\mathfrak{Sh}(X, C)$ *is the full subcategory of the category of presheaves satisfying the sheaf condition.*

▶ **Definition 3** (Locally Ringed Space [14, 9]). *If $\mathcal{O}_X$ is a sheaf of rings on X, then the pair $(X, \mathcal{O}_X)$ is called a* ringed space*; a* morphism *between two ringed space $(X, \mathcal{O}_X)$ and $(Y, \mathcal{O}_Y)$ is a pair (f, ϕ) such that $f : X \to Y$ is continuous and $\phi : \mathcal{O}_Y \to f_*\mathcal{O}_X$ is a morphism of sheaves where $f_*\mathcal{O}_X \in \mathfrak{Sh}(Y)$ assigns $V \subseteq Y$ to $\mathcal{O}_X(f^{-1}(V))$. A* locally ringed space *$(X, \mathcal{O}_X)$ is a ringed space such that for any $x \in X$, its stalk $\mathcal{O}_{X,x}$ is a local ring where $\mathcal{O}_{X,x} = \mathrm{colim}_{x \in U \in \mathfrak{Opens} X} \mathcal{O}_X(U)$; a* morphism *between two locally ringed spaces $(X, \mathcal{O}_X)$ and $(Y, \mathcal{O}_Y)$ is a morphism (f, ϕ) of ringed space such that for any $x \in X$ the ring homomorphism induced on stalk $\phi_x : \mathcal{O}_{Y,f(x)} \to \mathcal{O}_{X,x}$ is local.*

From the previous definitions, if $\mathcal{O}_X$ is a presheaf and $U \subseteq X$ is an open subset, then there is a presheaf $\mathcal{O}_X\,|_U$ on U by assigning every open subset V of U to $\mathcal{O}_X(V)$. This is called restricting a presheaf; sheaves, ringed spaces and locally ringed spaces can also be similarly restricted.

2.2 Definition of Affine Scheme and Scheme

The Spec construction

Let R be a ring and let $\mathrm{Spec}\, R$ denote the set of prime ideals of R. Then for any subset $s \subseteq R$, its zero locus is defined as $\{\mathfrak{p} \mid s \subseteq \mathfrak{p}\}$. These zero loci can be considered as closed subsets of $\mathrm{Spec}\, R$; the topology thus formed is called the Zariski topology. Then a sheaf of rings on $\mathrm{Spec}\, R$ can be defined by assigning $U \subseteq \mathrm{Spec}\, R$ to the ring

$$\left\{ s : \prod_{x \in U} R_x \mid s \text{ is locally a fraction} \right\},$$

where s is *locally a fraction* if and only if for any prime ideal $x \in U$, there is always an open subset $x \in V \subseteq U$ and $a, b \in R$ such that for any prime ideal $y \in V$, $b \notin y$ and $s(y) = \frac{a}{b}$. This sheaf $\mathcal{O}$ is called the *structure sheaf* of $\mathrm{Spec}\, R$. $(\mathrm{Spec}\, R, \mathcal{O})$ is a locally ringed space because for any prime ideal $x \subseteq R$, $\mathcal{O}_x \cong A_x$ [9, Chapter 2, Proposition 2.2].

Scheme

▶ **Definition 4** (Scheme). *A locally ringed space $(X, \mathcal{O}_X)$ is said to be a* scheme *if for any $x \in X$, there is always some ring R and some open subset $x \in U \subseteq X$ such that $(U, \mathcal{O}_X\,|_U) \cong (\mathrm{Spec}\, R, \mathcal{O}_{\mathrm{Spec}\, R})$ as locally ringed spaces. The* category of schemes *is the full subcategory of locally ringed spaces where objects are schemes.*

Thus to construct a scheme, one needs the following:

- a topological space X;
- a presheaf $\mathcal{O}$;
- a proof that $\mathcal{O}$ satisfies the sheaf condition;
- a proof that all stalks are local;
- an open covering $\{U_i\}$ of X;
- a collection of rings $\{R_i\}$ and isomorphism $(U_i, \mathcal{O}_X|_{U_i}) \cong (\operatorname{Spec} R_i, \mathcal{O}_{\operatorname{Spec} R})$.

In Section 2.3, the Proj construction will be described following the steps above. Hence, the Proj construction though appears to be a definition, is in fact a mixture of defining a ringed space and a proof that the constructed ringed space is locally affine.

2.3 The Proj Construction

Throughout this section, R will denote a ring and A an $\mathbb{N}$-graded R-algebra, in order to keep notations the same as Section 3, the grading of A will be written as $\mathcal{A}$, i.e. $A \cong \bigoplus_{i\in\mathbb{N}} \mathcal{A}_i$ as R-algebras.

Topology

▶ **Definition 5** ($\operatorname{Proj}\mathcal{A}$ as a set)**.** $\operatorname{Proj}\mathcal{A}$ *is defined to be* $\{\mathfrak{p} \in \operatorname{Spec} A \mid \mathfrak{p}$ *is homogeneous and relevant*$\}$*, where*

- *an ideal* $\mathfrak{p} \subseteq A$ *is said to be* homogeneous *if for any* $a \in \mathfrak{p}$ *and* $i \in \mathbb{N}$*,* a_i *is in* $\mathfrak{p}$ *as well where* $a_i \in \mathcal{A}_i$ *is the* i*-th projection of* a *with respect to grading* $\mathcal{A}$*;*
- *an ideal* $\mathfrak{p} \subseteq A$ *is said to be* relevant *if* $\bigoplus_{i=1}^{\infty} \mathcal{A}_i \not\subseteq \mathfrak{p}$*.*

Similar to Spec construction in Section 2.2, there is a topology on $\operatorname{Proj}\mathcal{A}$ whose close sets are exactly the zero loci where for any $s \subseteq A$, zero locus of s is $\{\mathfrak{p} \in \operatorname{Proj}\mathcal{A} \mid s \subseteq \mathfrak{p}\}$; this topology is also called the Zariski topology. For any $a \in A$, $D(a)$ denotes the set $\{x \in \operatorname{Proj}\mathcal{A} \mid a \notin x\}$.

▶ **Theorem 6.** *For any* $a \in A$*,* $D(a)$ *is open in Zariski topology and* $\{D(a) \mid a \in A\}$ *forms a basis of the Zariski topology.*

Proof. Proofs can be found in [14, 00JM] and [9, Chapter 2, proposition 2.5] ◀

Structure sheaf

Let $U \subseteq \operatorname{Proj}\mathcal{A}$ be an open subset. The *sections* on U are defined to be

$$\mathcal{O}(U) = \left\{ s \in \prod_{x\in U} A^0_x \mid s \text{ is locally a homogeneous fraction} \right\},$$

where $A^0_{\mathfrak{p}}$ denotes the homogeneous localization of A at a homogeneous prime ideal $\mathfrak{p}$, i.e. the subring of $A_{\mathfrak{p}}$ of elements of degree zero, and s is said to be *locally a homogeneous fraction* if for any $x \in U$, there is some open subset $x \in V \subseteq U$, $i \in \mathbb{N}$ and $a, b \in \mathcal{A}_i$ such that for all $y \in V$, $s(y) = \frac{a}{b}$. Equipped with the natural restriction maps, $\mathcal{O}$ defined in this way forms a presheaf; the sheaf condition of $\mathcal{O}$ is checked in the category of sets where it follows from the definition of locally homogeneous fractions. This sheaf is called the *s*tructure sheaf of $\operatorname{Proj}\mathcal{A}$, also written as $\mathcal{O}_{\operatorname{Proj} A}$

Locally ringed spaces

▶ **Theorem 7.** *The stalk of* $(\operatorname{Proj}\mathcal{A}, \mathcal{O})$ *at a homogeneous prime relevant ideal* $\mathfrak{p}$ *is isomorphic to* $A^0_{\mathfrak{p}}$.

Proof. It can be checked that the function $A^0_{\mathfrak{p}} \to \mathcal{O}_{\operatorname{Proj}\mathcal{A},\mathfrak{p}}$ defined by $\frac{a}{b} \mapsto \langle D(b), x \mapsto \frac{a}{b}\rangle$ is a ring isomorphism. Details can be found in [14, 01M4] ◀

Since $A^0_{\mathfrak{p}}$ is a local ring for any homogeneous prime ideal $\mathfrak{p}$, it can be concluded that $(\operatorname{Proj}\mathcal{A}, \mathcal{O}_{\operatorname{Proj}\mathcal{A}})$ is a locally ringed space.

Affine cover

▶ **Lemma 8.** *For any* $x \in \operatorname{Proj}\mathcal{A}$*, there is some* $0 < m \in \mathbb{N}$ *and* $f \in \mathcal{A}_m$*, such that* $x \in D(f)$*, i.e.* $f \notin x$.

Proof. Let $x \in \operatorname{Proj}\mathcal{A}$, by construction, $\bigoplus_{i=1}^{\infty}\mathcal{A}_i \not\subseteq x$. Thus there is some $f = f_1 + f_2 + \cdots \notin x$, then at least one $f_i \notin x$ for otherwise $f \in x$. ◀

Thus, to construct an affine cover, it is sufficient to prove that for all $0 < m \in \mathbb{N}$ and homogeneous element $f \in \mathcal{A}_m$, $(D(f), \mathcal{O}_{\operatorname{Proj}\mathcal{A}}|_{D(f)})$ is isomorphic to $(\operatorname{Spec}A^0_f, \mathcal{O}_{\operatorname{Spec}A^0_f})$ where A^0_f is the subring of the localised ring A_f consisting of elements of degree zero. By fixing the previous notations, an isomorphism between locally ringed space is a pair (ϕ, α) where ϕ is a homeomorphism between the topological spaces $D(f)$ and $\operatorname{Spec}A^0_f$ and α an isomorphism between $\phi_*(\mathcal{O}_{\operatorname{Proj}\mathcal{A}}|_{D(f)})$ and $\mathcal{O}_{\operatorname{Spec}A^0_f}$.

▶ **Theorem 9.** $D(f) \cong \operatorname{Spec}A^0_f$ *are homeomorphic as topological spaces.*

The following proofs are an expansion of [9, II.2.5] while drawing ideas from [15, II.4.5].

Proof. Define $\phi : D(f) \to \operatorname{Spec}A^0_f$ by $\mathfrak{p} \mapsto \operatorname{span}\left\{\frac{g}{1} \mid g \in \mathfrak{p}\right\} \cap A^0_f$; by clearing denominators, one can show that $\phi(\mathfrak{p}) = \operatorname{span}\left\{\frac{g}{f^i} | g \in \mathfrak{p} \cap A_{mi}\right\}$. One can check that $\phi(\mathfrak{p})$ is indeed a prime ideal. ϕ is continuous by checking on the topological basis consisting of basic open sets of $\operatorname{Spec}A^0_f$. The fact that basic open sets form a basis is already recorded in mathlib. Take $\frac{a}{f^n} \in A^0_f$, then $\phi^{-1}\left(D\left(a/f^n\right)\right) = D(f) \cap D(a)$.

- $D(f) \cap D(a)$ is a subset of $\phi^{-1}\left(D\left(a/f^n\right)\right)$ because if $y \in D(f) \cap D(a)$ and $a/f^n \in \phi(y)$, i.e. $a/f^n = \sum_i (c_i/f^{n_i})(g_i/1)$, then by multiplying suitable powers of f, $af^N/1 = (\sum_i c_i g_i f^{m_i})/1$ for some N, so by definition of localisation, $af^N f^M = \sum_i c_i g_i f^{m_i}$ for some M implying that $a \in y$. Contradiction.
- On the other hand, if $\phi(y) \in D\left(a/f^n\right)$ and $a \in y$, then $a/1 \in h(y)$, contradiction because $a/f^n = a/1 1/f^n \in \phi(y)$.

For the other direction, define $\psi : \operatorname{Spec}A^0_f \to D(f)$ to be $x \mapsto \left\{a \mid \text{for all } i \in \mathbb{N}, \frac{a_i^m}{f^i} \in x\right\}$. For ψ to be well-defined, one needs to check that $\psi(x)$ is a homogeneous prime ideal that is relevant. Continuity of ψ depends on that ϕ and ψ are inverse to each other. $D(f)$ with the subspace topology has a basis of the form $D(f) \cap D(a)$, thus it is sufficient to prove that preimages of these sets are open. By considering $\phi(D(f) \cap D(a)) = \bigcup_i \phi(D(f) \cap D(a_i))$, each $\phi(D(f) \cap D(a_i))$ is open because $\phi(D(f) \cap D(a_i)) = D\left(a_i^m/f^i\right)$ in $\operatorname{Spec}A^0_f$. To prove $\phi(D(f) \cap D(a_i)) = D\left(a_i^m/f^i\right)$, it is sufficient to prove $\phi^{-1}(D\left(a_i^m/f^i\right)) = D(f) \cap D(a)$ and this is true by continuity of ϕ. Since ϕ and ψ are inverses to each other, preimage of $D(f) \cap D(a)$ is indeed $\phi(D(f) \cap D(a))$. ◀

Let ϕ and ψ be the continuous functions defined in the previous proof, U be an open subset of $\operatorname{Spec} A_f^0$, s be a section on $\phi^{-1}(U)$ and $x \in U$, then $\psi(x) \in \phi^{-1}(U)$, hence $s(\psi(x)) = \frac{n}{d} \in A^0_{\psi(x)}$ for some $i \in \mathbb{N}$ and $n, d \in \mathcal{A}_i$. Keeping the same notation, a ring homomorphism $\alpha_U : \phi_*(\mathcal{O}_{\operatorname{Proj}}|_{D(f)})(U) \to \mathcal{O}_{\operatorname{Spec} A_f^0}(U)$ can be defined as $s \mapsto \left(x \mapsto \frac{n d^{m-1}/f^i}{d^m/f^i}\right)$ where $n, d \in \mathcal{A}_i$. Assuming α_U is well-defined, it is easy to check that $U \mapsto \alpha_U$ is natural in U, hence α defines a morphism of sheaves.

▶ **Lemma 10.** *For any open subset $U \subseteq \operatorname{Spec} A_f^0$, α_U is well-defined; hence α defines a morphism of sheaves.*

Proof. It is clear that both the numerator and denominator have degree zero. Now $d^m/f^i \notin x$ follows from $d \notin \psi(x)$. Next $\alpha_U(s)$ is locally a fraction: since s is locally a quotient, for any $x \in U$, there is some open set $V \subseteq \operatorname{Proj}\mathcal{A}$ such that $\psi(x) \in V \subseteq \phi^{-1}(U)$ such that $s(y) = \frac{a}{b}$ for all $y \in V$ where $a, b \in A_n$ and $b \notin y$, then to check $\alpha_U(s)$ is locally quotient, use the open subset $\phi(V)$ and check that for all $z \in \phi(V)$, $\alpha_U(s)(z) = \frac{a b^{m-1}}{b^m}$. The proof of α_U being a ring homomorphism involves manipulations of fractions in localised rings, for more details, see Section 3. ◀

In the other direction, if $s \in \mathcal{O}_{\operatorname{Spec} A_f^0}(U)$ and $y \in \phi^{-1}(U)$, then $\phi(y) \in U$, so $s(\phi(y))$ can be written as $\frac{a}{b}$ where $a, b \in A_f^0$; then a can be written as $\frac{n_a}{f^{i_a}}$ for some $n_a \in A_{m i_a}$ and b as $\frac{n_b}{f^{i_b}}$ for some $n_b \in A_{m i_b}$. Hence, a ring homomorphism $\beta_U : \mathcal{O}_{\operatorname{Spec} A_f^0}(U) \to \mathcal{O}_{\operatorname{Proj}}|_{D(f)}\ (\phi^{-1}(U))$ can be defined as $s \mapsto \left(y \mapsto \frac{n_a f_b^i}{n_b f^{i_a}}\right)$. Assuming β is well defined, it is easy to check that the assignment $U \mapsto \beta_U$ is natural so that β is a natural transformation.

▶ **Lemma 11.** *For any open subset $U \subseteq \operatorname{Spec} A_f^0$, β_U is well-defined; hence β defines a morphism of sheaves.*

Proof. $n_a f_b^{i_b}$ and $n_b f^{i_a}$ have the same degree. $n_b f^{i_a} \notin y$ follows from $b \notin \phi(y)$. Since s locally is a fraction, there are open sets $\phi(y) \in V \subseteq U$, such that for all $z \in V$, $s(z)$ is $\frac{a/f^{l_1}}{b/f^{l_2}}$. Then on $\phi^{-1}(V) \subseteq \phi^{-1}(U)$, $\psi_U(s)(y)$ is always $\frac{a f^{l_2}}{b f^{l_1}}$. Checking that β_U is a ring homomorphism involves manipulating fractions of fractions. ◀

▶ **Theorem 12.** *$\phi_*(\mathcal{O}_{\operatorname{Proj}\mathcal{A}}|_{D(f)})$ and $\mathcal{O}_{\operatorname{Spec} A_f^0}$ are isomorphic as sheaves.*

Proof. By combining Lemma 10 and Lemma 11, it is sufficient to check $\alpha \circ \beta$ and $\beta \circ \alpha$ are both identities.

- $\beta \circ \alpha = 1$: let $s \in \mathcal{O}_{\operatorname{Proj}}|_{D(f)}\ (\phi^{-1}(U))$, then for $x \in \phi^{-1}(U)$

  $$\alpha_U(s) = x \mapsto \frac{n d^{m-1}/f^i}{d^m/f^i},$$

  where $s(x) = \frac{n}{d}$. Thus, by definition

  $$\beta_U(\alpha_U(s))(x) = \frac{n d^{m-1} f^i}{d^m f^i} = \frac{n}{d} = s(x).$$

- $\alpha \circ \beta = 1$: let $s \in \mathcal{O}_{\operatorname{Spec} A_f^0}(U)$, then for $x \in U$

  $$\beta_U(s) = x \mapsto \frac{n_a f^{i_b}}{n_b f^{i_a}}$$

  where $s(x) = \frac{n_a/f^{i_a}}{n_b/f^{i_b}}$. Thus

  $$\phi_U(\psi_U(s))(x) = \frac{n_a f^{i_b} (n_b f^{i_a})^{m-1}/f^j}{(n_b f^{i_a})^m/f^j} = \frac{n_a/f^{i_a}}{n_b/f^{i_b}} = s(x).$$ ◀

▶ **Corollary 13.** *$(\operatorname{Proj}\mathcal{A}, \mathcal{O}_{\operatorname{Proj}\mathcal{A}})$ is a scheme.*

3 Formalisation details

3.1 Homogeneous Ideal

Let A be an R-algebra and an ι-grading $\mathcal{A} : \iota \to R$-submodules of A, `ideal.is_homogeneous` is the proposition of an ideal being homogeneous and `homogeneous_ideal` is the type of all homogeneous ideals of A [17]. Note that, by this implementation, homogeneous ideals are not literally ideals, for this reason, $\operatorname{Proj} \mathcal{A}$ cannot be implemented as a subset of $\operatorname{Spec} A$.

```
def ideal.is_homogeneous : Prop :=
∀ (i : ι) {r : A}, r ∈ I → (direct_sum.decompose 𝒜 r i : A) ∈ I

structure homogeneous_ideal extends submodule A A :=
(is_homogeneous' : ideal.is_homogeneous 𝒜 to_submodule)

def homogeneous_ideal.to_ideal (I : homogeneous_ideal 𝒜) : ideal A :=
    I.to_submodule

lemma homogeneous_ideal.is_homogeneous (I : homogeneous_ideal 𝒜) :
  I.to_ideal.is_homogeneous 𝒜 := I.is_homogeneous'

def homogeneous_ideal.irrelevant : homogeneous_ideal 𝒜 :=
⟨(graded_ring.proj_zero_ring_hom 𝒜).ker, omitted⟩
```

3.2 Homogeneous Localisation

If x is a multiplicatively closed subset of ring A, then the homogeneous localisation of A at x is defined to be the subring of localised ring A_x consisting of elements of degree zero. This ring is implemented as triples $\{(i, a, b) : \iota \times \mathcal{A}_i \times \mathcal{A}_i \mid b \notin x\}$ under the equivalence relation that $(i_1, a_1, b_1) \approx (i_2, a_2, b_2) \overset{\text{def}}{\iff} \frac{a_1}{b_1} = \frac{a_2}{b_2}$ in A_x. The quotient approach gives an induction principle via quotients, though the construction still uses classical reasoning, many lemmas will be automatic because of the rich API in mathlib about quotient spaces already; compared to the subring approach, one would need to write corresponding lemmas manually by excessively invoking `classical.some` and `classical.some_spec` which are APIs in Lean to extract the data and the corresponding proof from an existentially quantified proposition. One potential benefit of the subring approach is that different propositions can be specified for different multiplicative subsets to customize what properties and attributes are to be made explicit; for example for localisation away from a single element, it is useful to make powers of denominators explicit. But this would sacrifice a universal approach to homogeneous localisation for different multiplicative subsets so that auxiliary lemmas would have to be duplicated. To maintain consistency and prevent duplication, this paper will adopt the approach via quotient space. Before writing this paper, the subring approach has also been tested. Comparing the two approaches proves that there is no significant difference in the smoothness of two formalisations but the quotient approach has a smaller code size.

```
variables {ι R A: Type*} [add_comm_monoid ι] [decidable_eq ι]
variables [comm_ring R] [comm_ring A] [algebra R A]
variables (𝒜 : ι → submodule R A) [graded_algebra 𝒜]
variables (x : submonoid A)

structure num_denom_same_deg :=
(deg : ι) (num denom : 𝒜 deg) (denom_mem : (denom : A) ∈ x)
```

```
8
9  def embedding (p : num_denom_same_deg 𝒜 x) : localization x :=
10 localization.mk p.num ⟨p.denom, p.denom_mem⟩
11
12 def homogeneous_localization : Type* := quotient (setoid.ker $ embedding 𝒜 x)
```

Then if `(y : homogeneous_localization 𝒜 x)`, its value, degree, numerator and denominator can all be defined by using induction/recursion principles for quotient spaces:

```
1  variable (y : homogeneous_localization 𝒜 x)
2
3  def val : localization x :=
4    quotient.lift_on' y (num_denom_same_deg.embedding 𝒜 x) $ λ _ _, id
5
6  def num : A := (quotient.out' y).num
7  def denom : A := (quotient.out' y).denom
8  def deg : ι := (quotient.out' y).deg
9
10 lemma denom_mem : y.denom ∈ x := (quotient.out' y).denom_mem
11 lemma num_mem_deg : y.num ∈ 𝒜 f.deg := (quotient.out' y).num.2
12 lemma denom_mem_deg : y.denom ∈ 𝒜 y.deg := (quotient.out' y).denom.2
13 lemma eq_num_div_denom : y.val = localization.mk y.num ⟨y.denom, y.denom_mem⟩ :=
14 omitted
```

3.3 The Zariski Topology

In this section A will be graded by $\mathbb{N}$ and the grading denoted by $\mathcal{A}$. Proj $\mathcal{A}$ is formalised a structure:

```
1  structure projective_spectrum :=
2  (as_homogeneous_ideal : homogeneous_ideal 𝒜)
3  (is_prime : as_homogeneous_ideal.to_ideal.is_prime)
4  (not_irrelevant_le : ¬(homogeneous_ideal.irrelevant 𝒜 ≤ as_homogeneous_ideal))
```

After building more API around `projective_spectrum`, the Zariski topology with a basis of basic open sets can be formalised as:

```
1  def zero_locus (s : set A) : set (projective_spectrum 𝒜) :=
2  {x | s ⊆ x.as_homogeneous_ideal}
3
4  instance zariski_topology : topological_space (projective_spectrum 𝒜) :=
5  topological_space.of_closed (set.range (zero_locus 𝒜)) omitted omitted omitted
6
7  def basic_open (r : A) : topological_space.opens (projective_spectrum 𝒜) :=
8  { val := { x | r ∉ x.as_homogeneous_ideal },
9    property := ⟨{r}, set.ext $ λ x, set.singleton_subset_iff.trans $ not_not.symm⟩ }
10
11 lemma is_topological_basis_basic_opens : topological_space.is_topological_basis
12   (set.range (λ (r : A), (basic_open 𝒜 r : set (projective_spectrum 𝒜)))) :=
13 omitted
```

3.4 Locally Ringed Spaces

mathlib provides `Top.presheaf.is_sheaf_iff_is_sheaf_comp` to check the sheaf condition by composing a forgetful functor and `Top.subsheaf_to_Types` to construct subsheaf of types

satisfying a local predicate [6]; $\mathcal{O}_{\mathrm{Spec}}$ in mathlib adopted this approach [5], and structure sheaf of Proj will also be constructed in this way. `is_locally_fraction` is a local predicate expressing "being locally a homogeneous fraction" in Section 2.3:

```
def is_fraction {U : opens (Proj 𝒜)} (f : Π x : U,  A⁰_x ) : Prop :=
∃ (i : ℕ) (r s : 𝒜 i), ∀ x : U, ∃ (s_nin : s.1 ∉ x.1.as_homogeneous_ideal),
  f x = quotient.mk' ⟨i, r, s, s_nin⟩

def is_fraction_prelocal : prelocal_predicate (λ (x : Proj 𝒜),  A⁰_x ) :=
{ pred := λ U f, is_fraction f,
  res := by rintros V U i f ⟨j, r, s, w⟩; exact ⟨j, r, s, λ y, w (i y)⟩ }

def is_locally_fraction : local_predicate (λ (x : Proj 𝒜),  A⁰_x ) :=
(is_fraction_prelocal 𝒜).sheafify

def structure_sheaf_in_Type : sheaf Type* (Proj 𝒜):=
subsheaf_to_Types (is_locally_fraction 𝒜)
```

The presheaf of rings is also defined as `structure_presheaf_in_CommRing` and it is checked that composition with the forgetful functor is naturally isomorphic to the underlying presheaf of `structure_sheaf_in_Type` which implies that `structure_presheaf_in_CommRing` satisfies the sheaf condition as well by using `Top.presheaf.is_sheaf_iff_is_sheaf_comp`.

```
def structure_presheaf_in_CommRing : presheaf CommRing (Proj 𝒜) :=
{ obj := λ U, CommRing.of ((structure_sheaf_in_Type 𝒜).1.obj U), ..omitted }

def structure_presheaf_comp_forget :
  structure_presheaf_in_CommRing 𝒜 ⋙ (forget CommRing) ≅
  (structure_sheaf_in_Type 𝒜).1 :=
omitted

def Proj.structure_sheaf : sheaf CommRing (Proj 𝒜) :=
⟨structure_presheaf_in_CommRing 𝒜, (is_sheaf_iff_is_sheaf_comp _ _).mpr
  (is_sheaf_of_iso (structure_presheaf_comp_forget 𝒜).symm
    (structure_sheaf_in_Type 𝒜).cond)⟩
```

Then following Theorem 7, `stalk_to_fiber_ring_hom` is a family of ring homomorphism $\prod_x \mathcal{O}_{\mathrm{Proj}\,\mathcal{A},x} \to A^0_x$ obtained by universal property of colimit with its right inverse as a family of function `homogeneous_localization_to_stalk`:

```
def stalk_to_fiber_ring_hom (x : Proj 𝒜) :
  (Proj.structure_sheaf 𝒜).presheaf.stalk x ⟶ CommRing.of A⁰_x :=
limits.colimit.desc (((open_nhds.inclusion x).op) ⋙ (Proj.structure_sheaf 𝒜).1)
  omitted

def section_in_basic_open (x : Proj 𝒜) :
  Π (f : A⁰_x), (Proj.structure_sheaf 𝒜).1.obj (op (Proj.basic_open 𝒜 f.denom)) :=
λ f, ⟨λ y, quotient.mk' ⟨_, ⟨f.num, _⟩, ⟨f.denom,_⟩, _⟩, _⟩

def homogeneous_localization_to_stalk (x : Proj 𝒜) :
  A⁰_x → (Proj.structure_sheaf 𝒜).presheaf.stalk x :=
λ f, (Proj.structure_sheaf 𝒜).presheaf.germ
  (⟨x, homogeneous_localization.mem_basic_open _ x f⟩ : Proj.basic_open _ f.denom)
  (section_in_basic_open _ x f)

```

```
def Proj.stalk_iso' (x : Proj 𝒜) :
  (Proj.structure_sheaf 𝒜).presheaf.stalk x ≃+* CommRing.of A⁰_x :=
ring_equiv.of_bijective (stalk_to_fiber_ring_hom _ x)
  ⟨omitted, function.surjective_iff_has_right_inverse.mpr
    ⟨homogeneous_localization_to_stalk 𝒜 x, omitted⟩⟩
```

Hence establishing that $\operatorname{Proj}\mathcal{A}$ is a locally ringed space:

```
def Proj.to_LocallyRingedSpace : LocallyRingedSpace :=
{ local_ring := λ x, @@ring_equiv.local_ring _
    (show local_ring  A⁰_x , from infer_instance) _
    (Proj.stalk_iso' 𝒜 x).symm,
    ..(Proj.to_SheafedSpace 𝒜) }
```

3.5 Affine cover

```
variables {f : A} {m : ℕ} (f_deg : f ∈ 𝒜 m) (x : Proj| D(f))
```

`Spec.T` and `Proj.T` denote the topological space associated with each locally ringed space. Let $0 < m \in \mathbb{N}$ and $f \in \mathcal{A}_m$ and $x \in D(f)$, by following Theorem 9, the continuous function ϕ is formalised as `Proj_iso_Spec_Top_component.to_Spec` where continuity is checked on basic open sets:

```
namespace Proj_iso_Spec_Top_component
namespace to_Spec

def carrier : ideal A⁰_f :=
ideal.comap (algebra_map A⁰_f A_f)
  (ideal.span $ algebra_map A (away f) '' x.val.as_homogeneous_ideal)

def to_fun : Proj.T| D(f) → Spec.T A⁰_f :=
λ x, ⟨carrier 𝒜 x, omitted /-a proof for primeness-/⟩

end to_Spec

def to_Spec (f : A) : Proj.T| D(f) ⟶ Spec.T A_f :=
{ to_fun := to_Spec.to_fun 𝒜 f,
  continuous_to_fun := omitted }
```

Similarly, ψ is defined as a function first, then the fact that ϕ and ψ are inverses to each other is formalised next as `to_Spec_from_Spec` and `from_Spec_to_Spec` respectively. The continuity of ψ hence follows.

```
namespace from_Spec

def carrier (q : Spec.T A⁰_f) : set A :=
{a | ∀ i, (quotient.mk' ⟨_, ⟨proj 𝒜 i a ^ m, _⟩, ⟨f^i, _⟩, _⟩ : A⁰_f) ∈ q.1}

def carrier.as_ideal : ideal A := { carrier := carrier f_deg q, ..omitted }
def carrier.as_homogeneous_ideal : homogeneous_ideal 𝒜 :=
⟨carrier.as_ideal f_deg hm q, omitted⟩

def to_fun : Spec.T A⁰_f → Proj.T| D(f) :=
λ q, ⟨⟨carrier.as_homogeneous_ideal f_deg hm q, omitted, omitted⟩, omitted⟩
```

```
12
13 end from_Spec
14
15 lemma to_Spec_from_Spec : to_Spec.to_fun 𝒜 f (from_Spec.to_fun f_deg hm x) = x :=
16 omitted
17 lemma from_Spec_to_Spec : from_Spec.to_fun f_deg hm (to_Spec.to_fun 𝒜 f x) = x :=
18 omitted
19
20 def from_Spec : Spec.T A⁰_f ⟶ Proj.T| D(f) :=
21 { to_fun := from_Spec.to_fun f_deg hm,
22   continuous_to_fun := omitted }
23
24 end Proj_iso_Spec_Top_component
```

The homeomorphism between $D(f)$ and Spec A_f^0 is achieved by combining ϕ and ψ together.

```
1 def Proj_iso_Spec_Top_component:
2   Proj.T| D(f) ≅ Spec.T (A⁰_f) :=
3 { hom := Proj_iso_Spec_Top_component.to_Spec 𝒜 f,
4   inv := Proj_iso_Spec_Top_component.from_Spec hm f_deg,
5   ..omitted /-composition being identity-/ }
```

Then by following Lemma 11, β is formalised as `Proj_iso_Spec_Sheaf_component.from_Spec`.

```
1 namespace Proj_iso_Spec_Sheaf_component
2 namespace from_Spec
```

Let V be an open set in Spec A_f^0 and s be a section on V, then let y be an element of $\phi^{-1}(V)$,

```
1 variables (V : (opens (Spec A⁰_f))ᵒᵖ) (s : (Spec A⁰_f).presheaf.obj V)
2 variables (y : ((@opens.open_embedding Proj.T D(f)).is_open_map.functor.op.obj
3   ((opens.map (Proj_iso_Spec_Top_component hm f_deg).hom).op.obj V)).unop)
4 -- This is but a verbose way of spelling y is in ϕ⁻¹(V) for type checking reasons.
```

one can evaluate $s(\phi(y))$ and represent the result as a fraction $\frac{a}{b}$ where $a = \frac{n_a}{f^{i_a}}$ and $b = \frac{n_b}{f^{i_b}}$.

```
1 -- Corresponding to evaluating a section in Lemma 11.s(ϕ(y))
2 def data : structure_sheaf.localizations A⁰_f
3   ((Proj_iso_Spec_Top_component hm f_deg).hom ⟨y.1, _⟩) :=
4 s.1 ⟨_, _⟩
5
6 -- s(ϕ(y)) = a/b, this is a, see Lemma 11.
7 def data.num : A⁰_f := omitted
8
9 -- s(ϕ(y)) = a/b, this is b, see Lemma 11
10 def data.denom : A⁰_f := omitted
```

Then $\frac{n_a f^{i_b}}{n_b f^{i_a}}$ is a homogeneous fraction in A_y^0. The function thus defined is indeed a ring homomorphism and locally a fraction. This sheaf morphism is recorded as `from_Spec` where its naturality is checked automatically by Lean's simplifier.

```
1 -- s ↦ (y ↦ n_a f^{i_b} / n_b f^{i_a}), this is n_a f^{i_b}, see Lemma 11.
2 def num : A :=
3   (data.num _ hm f_deg s y).num * (data.denom _ hm f_deg s y).denom
4
5 -- s ↦ (y ↦ n_a f^{i_b} / n_b f^{i_a}), this is n_b f^{i_a}, see Lemma 11.
```

```
def denom : A :=
  (data.denom _ hm f_deg s y).num * (data.num _ hm f_deg s y).denom

def bmk : A⁰_y :=
quotient.mk'
{ deg := (data.num _ hm f_deg s y).deg + (data.denom _ hm f_deg s y).deg,
  num := ⟨num hm f_deg s y, _⟩,
  denom := ⟨denom hm f_deg s y, _⟩,
  denom_mem := omitted }

def to_fun.aux : ((Proj_iso_Spec_Top_component hm f_deg).hom _* (Proj|
    D(f)).presheaf).obj V :=
⟨bmk hm f_deg V s, omitted /-being locally a homogeneous fraction-/⟩

def to_fun : (Spec A⁰_f).presheaf.obj V ⟶
  ((Proj_iso_Spec_Top_component hm f_deg).hom _* (Proj| D(f)).presheaf).obj V :=
{ to_fun := λ s, to_fun.aux 𝒜 hm f_deg V s, ..omitted /-ring homomorphism
    proofs-/ }

end from_Spec

def from_Spec : (Spec A⁰_f).presheaf ⟶
  (Proj_iso_Spec_Top_component hm f_deg).hom _* (Proj| D(f)).presheaf :=
{ app := λ V, from_Spec.to_fun 𝒜 hm f_deg V,
  naturality' := λ _ _ _, by { ext1, simp } }

end Proj_iso_Spec_Sheaf_component
```

By following Lemma 10, α is formalised as `Proj_iso_Spec_Sheaf_component.to_Spec`: let U be an open set in $\operatorname{Spec} A_f^0$ and s a section in $\phi_*(\mathcal{O}_{\operatorname{Proj}}|_{D(f)})(U)$, then let y be any point in U,

```
namespace Proj_iso_Spec_Sheaf_component
namespace to_Spec
variable (U : (opens (Spec.T A⁰_f))ᵒᵖ)
variable (s : ((Proj_iso_Spec_Top_component hm f_deg).hom _*
    (Proj| D(f))).presheaf.obj U) -- (φ_*(𝒪_Proj|_D(f)))(U)
```

After evaluating $s(\psi(y))$, the result can be represented as $\frac{n}{d}$ where n, d both have degree i. Then $\frac{nd^{m-1}}{f^i}$ and $\frac{d^m}{f^i}$ are both homogeneous fractions of the same degree and hence ${}^{nd^{m-1}/f^i}/_{(d^m/f^i)}$ is an element of the twice localised ring $\left(A_f^0\right)_y$. The function thus defined is a ring homomorphism and locally a fraction. This sheaf morphism is recorded as `to_Spec` where its naturality is checked automatically by Lean's simplifier.

```
-- evaluating a section, this is s(ψ(y))
def hl (y : unop U) : homogeneous_localization 𝒜 _ :=
s.1 ⟨((Proj_iso_Spec_Top_component hm f_deg).inv y.1).1, _⟩

-- s ↦ (x ↦ n d^(m-1)/f^i / d^m/f^i) where n,d ∈ 𝒜_i, this is n d^(m-1)/f^i, see Lemma 10.
def num (y : unop U) : A⁰_f :=
quotient.mk'
{ deg := m * (hl hm f_deg s y).deg,
  num := ⟨(hl hm f_deg s y).num * (hl hm f_deg s y).denom ^ m.pred, _⟩,
  denom := ⟨f^(hl hm f_deg s y).deg, _⟩,
```

```
  denom_mem := _ }

def denom (y : unop U) : A⁰_f :=
quotient.mk'
{ deg := m * (hl hm f_deg s y).deg,
  num := ⟨(hl hm f_deg s y).denom ^ m, _⟩,
  denom := ⟨f ^ (hl hm f_deg s y).deg,_⟩,
  denom_mem := _ }

def fmk (y : unop U) : (A⁰_f)_y :=
mk (num hm f_deg s y) ⟨denom hm f_deg s y, _⟩

def to_fun :
  ((Proj_iso_Spec_Top_component hm f_deg).hom _* (Proj| D(f))).obj U ⟶
  (Spec A⁰_f).presheaf.obj U :=
{ to_fun := λ s, ⟨λ y, fmk hm f_deg s y, omitted /-proof of being locally a
    fraction-/⟩, ..omitted /-proof of being a ring homomorphism-/},
end to_Spec

def to_Spec :
  (Proj_iso_Spec_Top_component hm f_deg).hom _* (Proj| D(f)).presheaf ⟶
  (Spec A⁰_f).presheaf :=
{ app := λ U, to_Spec.to_fun hm f_deg U,
  naturality' := λ U V subset1, by { ext1, simp } }
end Proj_iso_Spec_Sheaf_component
```

After checking `from_Spec` (β) and `to_Spec` (α) compose to identity, one establishes that $(D(f), \mathcal{O}_{\operatorname{Proj}\mathcal{A}})$ is isomorphic $(\operatorname{Spec} A_f^0, \mathcal{O}_{\operatorname{Spec} A_f^0})$ as locally ringed spaces. Hence $\operatorname{Proj}\mathcal{A}$ with structure sheaf , $\mathcal{O}_{\operatorname{Proj}\mathcal{A}}$ is a scheme.

```
def Sheaf_component:
  (Proj_iso_Spec_Top_component hm f_deg).hom _* (Proj| D(f)).presheaf ≅
  (Spec A⁰_f).presheaf :=
{ hom := Proj_iso_Spec_Sheaf_component.to_Spec 𝒜 hm f_deg,
  inv := Proj_iso_Spec_Sheaf_component.from_Spec 𝒜 hm f_deg,
  ..omitted /-composition is identity-/ }

def iso:
  (Proj| D(f)) ≅ Spec A⁰_f :=
let H : (Proj| D(f)).to_PresheafedSpace ≅ (Spec A⁰_f).to_PresheafedSpace :=
  PresheafedSpace.iso_of_components
    (Proj_iso_Spec_Top_component hm f_deg) (Sheaf_component 𝒜 f_deg hm) in
LocallyRingedSpace.iso_of_SheafedSpace_iso
{ hom := H.1, inv := H.2, hom_inv_id' := H.3, inv_hom_id' := H.4 }

def Proj.to_Scheme : Scheme :=
{ local_affine := omitted,..Proj }
```

This concludes the formalisation of the Proj construction for any $\mathbb{N}$-graded rings. In [17], $R[X_0, \ldots, X_n]$ is endowed with a grading by its R-submodule of homogeneous polynomials of fixed degrees so that projective n-space over R can be formalised as `Proj.to_Scheme (λ i, mv_polynomial.homogeneous_submodule (fin (n + 1)) R i)`; similarly, once the fact that quotient operation induces a grading on the quotiented object is formalised, projective varieties can also be defined using `Proj.to_Scheme`.

3.6 Reflections on the formalisation

An example of a calculation

Most calculations in proofs of Theorem 9 and Lemmas 10 and 11 are omitted. I present the details of verifying β_U preserves multiplication to showcase the flavour of calculations involved. Verifying that β_U preserving zero and one is similar but slightly simpler while preservation of addition is more cumbersome. Since α only involves one layer of fractions, calculations are not as long.

Let x, y be two sections, the aim is to show $\beta_U(xy) = \beta_U(x)\beta_U(y)$, i.e. for all $z \in \phi^{-1}(U)$, $\beta_U(xy)(z) = \beta_U(x)(z)\beta_U(y)(z)$.

```
lemma bmk_mul (x y : (Spec A⁰_f).presheaf.obj V) :
  bmk hm f_deg V (x * y) = bmk hm f_deg V x * bmk hm f_deg V y :=
begin
  ext1 z,
```

by writing $x(\phi(z))$ as $\frac{a_x/f^{i_x}}{b_x/f^{j_x}}$, $y(\phi(z))$ as $\frac{a_y/f^{i_y}}{b_y/f^{j_y}}$ and $(xy)(\phi(z)) = \frac{a_{xy}/f^{i_{xy}}}{b_{xy}/f^{j_{xy}}}$, one deduces that $\frac{a_x a_y/f^{i_x+i_y}}{b_x b_y/f^{j_x+j_y}} = \frac{a_{xy}/f^{i_{xy}}}{b_{xy}/f^{j_{xy}}}$, by definition of equality in localised ring, it implies that, there is some $\frac{c}{f^l}$ such that

$$\frac{a_x a_y b_{xy} c}{f^{i_x+i_y+j_{xy}+l}} = \frac{a_{xy} b_x b_y c}{f^{i_{xy}+j_x+j_y+l}}.$$

```
  have mul_eq := data.eq_num_div_denom hm f_deg (x * y) z,
  ... -- simplification
  erw is_localization.eq at mul_eq,
  obtain ⟨⟨C, hC⟩, mul_eq⟩ := mul_eq, -- C is the c/f^l above.
  ...

  -- c ∉ z
  have C_not_mem : C.num ∉ z.1.as_homogeneous_ideal := omitted,

  -- setting up notations.
  set a_xy := _, set i_xy := _, set b_xy := _, set j_xy := _,
  set a_x := _, set i_x := _, set b_x := _, set j_x := _,
  set a_y := _, set i_y := _, set b_y := _, set j_y := _,
  set l := _,
  ...
```

By definition of equality in localisation again, there exists some $n_1 \in \mathbb{N}$ such that

$$a_x a_y b_{xy} c f^{i_x y + j_x + j_y + l + n_1} = a_{xy} b_x b_y c f^{i_x + i_y + j_{xy} + l + n_1} \tag{1}$$

```
  obtain ⟨⟨_, ⟨n1, rfl⟩⟩, mul_eq⟩ := mul_eq,
```

The aim is to show

$$\frac{a_{xy} f^{j_{xy}}}{b^{xy} f^{i_x}} = \frac{a_x f^{j_x}}{b^x f^{i_x}} \frac{a_y f^{j_y}}{b_y f^{i_y}},$$

by Equation (1) and definition of equality in localised ring, cf^{l+n_1} verifies this equality.

```
suffices : (mk (a_xy * f ^ j_xy) ⟨b_xy * f ^ i_xy, _⟩ : localization.at_prime _)
= mk (a_x * f ^ j_x) ⟨b_x * f ^ i_x, _⟩ * mk (a_y * f ^ j_y) ⟨b_y * f ^ i_y, _⟩
  := omitted,
  ...
refine ⟨⟨C.num * f^(l + n1), _⟩, _⟩,
  ...
end
```

In totality, this is about ~100 lines of code by following essentially three lines of calculation when done with pen-and-paper. Admittedly, the above code is not the most optimal, but the magnitude is not greatly exaggerated. Strictly speaking, setting 13 variable names takes a lot of code and is not necessary, but with readable variable names, rewriting is made much simpler in the latter stage of this calculation. I think the following factors contribute to the differences between formalisation and a pen-and-paper-proof:

- Every element of a localised ring can be written as a fraction of a numerator and a denominator is a corollary of the construction but does not follow straightly from its definition. When written on a paper, it is often read "let $\frac{a}{b} \in A_{\mathfrak{p}}$" while in Lean it is becomes `intro x, set x_denom := ..., set x_num := ..., have eq1 : x.val = x_num / x_denom := ...`. This problem is more noticeable when `rewrite [eq1]` is unsound. Thus, many extra steps are required to set up the proof.
- Elements of a (homogeneously) localised ring contain not only data, but proofs as well. For example, the denominator of an element is a term `⟨d, some_proof⟩` of a subtype. This makes `rewrite` less smooth to use, for equalities are often of the form `h : d = d'`, thus `rewrite [h]` is type theoretically unsound.
- Terms of `localization x` or `homogeneous_localization` $\mathcal{A}$ `x` have to contain proofs to make the definitions correct, thus constructing any term of these types requires many proofs or disproofs of membership. Thus, a formalised calculation cannot be as liberal as a pen-and-paper-proof when come to whether the terms are well-defined. The situation can be partially mitigated by writing a simple tactic to try lemmas involving degrees of an element in a graded object, for example automatically splitting `a * b` $\in \mathcal{A}$ `(m + n)` to `a` $\in \mathcal{A}$ `(m + n)` to `a` $\in \mathcal{A}$ `m` and `b` $\in \mathcal{A}$ `n` and try recursively try to solve both. However, if non-definitional equalities is involved, tactics would be less helpful, when the subterms are in the wrong order, one needs to manually re-organise the subterms into its correct order to use the customary tactic.
- Not many high powered tactics are available for localised ring, for example `ring` will be able to solve `x * y = y * x` and much more complicated goal in a commutative ring, but `ring` cannot (and should not be able to) solve `(a / b * c / d : localization _) = c / b * a / d`.

The first three bullet points are essentially all because formalisation requires more rigour than that of pen-and-paper proofs; whether the requirement of extra rigour is beneficial is another question and not in the scope of this paper. For the fourth bullet point, it is definitely helpful to have a tactic automating many proofs, the catch is that equality in localised ring is existentially quantified – $\frac{a}{b} = \frac{a'}{b'}$ if and only if $ab'c = a'bc$ for some c in a multiplicative subset, while proving $ab'c = a'bc$ is easily mechanized by the `ring` tactic, providing c to Lean is certainly hard to be made trivial by any tactic soon. Thus, a tactic can only do so much without human input for now.

On propositional equality

Originally, I expected propositional equalities that are not equal by definition such as $\phi(\psi(y)) = y$ in Theorem 9 would pose a challenge, but the difficulty is less severe: indeed, I only need to prove some redundant lemma like $\phi(\psi(y))$ is in some open sets that clearly contains y; the reason is that in this project I did not compare algebraic structures depending on propositional equality, i.e. $\mathcal{O}_y$ and $\mathcal{O}_{\phi(\psi(y))}$; but foreseeably, this difficulty will come back when one starts to develop the theory of projective variety furtherer.

4 Conclusion

Since a large part of modern algebraic geometry depends on the Proj construction, much potential future research is possible: calculating cohomology of projective spaces; defining projective morphisms; Serre's twisting sheaves to name a few. Other approaches to the Proj construction also exist, for example, by gluing a family of schemes together; however, since there is no other formalisation of the Proj construction, I could not compare different approaches or compare capabilities of formalising modern algebraic geometry of different theorem provers. Thus I would like to conclude this paper with an invitation/challenge – state and formalise something involving more than affine schemes in your preferred theorem prover; for the only way to know which, if any, theorem provers handle modern mathematics satisfactorily is to actually formalise more modern mathematics.

References

1 Anthony Bordg, Lawrence Paulson, and Wenda Li. Simple type theory is not too simple: Grothendieck's schemes without dependent types. *Experimental Mathematics*, 31(2):364–382, 2022. doi:10.1080/10586458.2022.2062073.

2 Kevin Buzzard, Chris Hughes, Kenny Lau, Amelia Livingston, Ramon Fernández Mir, and Scott Morrison. Schemes in Lean. *Experimental Mathematics*, 31(2):355–363, 2022.

3 Laurent Chicli. *Une formalisation des faisceaux et des schémas affines en théorie des types avec Coq*. PhD thesis, INRIA, 2001.

4 Wei-Liang Chow. On compact complex analytic varieties. *American Journal of Mathematics*, 71(4):893–914, 1949. URL: http://www.jstor.org/stable/2372375.

5 Mathlib Contributors. Lean mathlib. https://github.com/leanprover-community/mathlib, 2022.

6 Mathlib Contributors. Mathlib documentation. https://leanprover-community.github.io/mathlib_docs, 2023.

7 Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

8 Roger Godement. Topologie algébrique et théorie des faisceaux. *Publications de*, 1, 1958.

9 Robin Hartshorne. Graduate texts in mathematics. *Algebraic Geometry*, 52, 1977.

10 Saunders MacLane and Ieke Moerdijk. *Sheaves in geometry and logic: A first introduction to topos theory*. Springer Science & Business Media, 2012.

11 Anders Mörtberg and Max Zeuner. Towards a formalization of affine schemes in cubical agda.

12 Amnon Neeman. *Algebraic and analytic geometry*. Cambridge University Press, 2007.

13 Jean-Pierre Serre. Géométrie analytique et géométrie algébrique. *Ann. Inst. Fourier, VI (1955–56)*, pages 1–42, 1955.

14 The Stacks Project Authors. *Stacks Project*. https://stacks.math.columbia.edu, 2018.

15 Ravi Vakil. The rising sea: Foundations of algebraic geometry. 2017. *URL http://virtualmath1. stanford. edu/˜ vakil/216blog*, 24:29, 2017.

16 Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. UniMath — a computer-checked library of univalent mathematics. available at `http://unimath.org`. URL: `https://github.com/UniMath/UniMath`.

17 Eric Wieser and Jujian Zhang. Graded rings in Lean's dependent type theory. In *International Conference on Intelligent Computer Mathematics*, pages 122–137. Springer, 2022.

Fermat's Last Theorem for Regular Primes

Alex J. Best
King's College London, UK

Christopher Birkbeck
University of East Anglia, Norwich, UK

Riccardo Brasca
Université Paris Cité, France

Eric Rodriguez Boidi
King's College London, UK

Abstract

We formalise the proof of the first case of Fermat's Last Theorem for regular primes using the *Lean* theorem prover and its mathematical library *mathlib*. This is an important 19th century result that motivated the development of modern algebraic number theory. Besides explaining the mathematics behind this result, we analyze in this paper the difficulties we faced in the formalisation process and how we solved them. For example, we had to deal with a diamond about characteristic zero fields and problems arising from multiple nested coercions related to number fields. We also explain how we integrated our work to *mathlib*.

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

Keywords and phrases Fermat's Last Theorem, Cyclotomic fields, Interactive theorem proving, Lean

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.36

Category Short Paper

Supplementary Material *Software*: `https://github.com/leanprover-community/flt-regular`
archived at `swh:1:dir:c19341f16f36abca8203dcb2b5a8dbbbc1864b06`

Funding *Alex J. Best*: AB. was supported by NWO Vidi grant 639.032.613.

Acknowledgements We thank the *mathlib* community for a lot of useful discussions around our project. We especially thank Ruben Van de Velde for having formalised in Lean a proof of Fermat's Last Theorem in the case $n = 3$.

1 Introduction

Fermat's Last Theorem states that for $n \geq 3$, the equation $x^n + y^n = z^n$ has no solutions with $x, y, z \in \mathbb{Z}$ and $xyz \neq 0$. This question remained unsolved for 300 years until the eventual proof of this was completed by Andrew Wiles and Richard Taylor [12, 10] in 1994. This proof requires a great deal of mathematical machinery in order to study deep connections between number theory, algebra, geometry and analysis and its formalisation is currently out of reach. However, certain special cases of this theorem were already known long before Wiles' work. First of all, it's easy to prove that we can restrict to the case where the exponent n is an odd prime p. Moreover, Kummer proved in 1847 that

▶ **Theorem 1** (Kummer). *Let p be a regular (odd) prime. Then $x^p + y^p = z^p$ has no solutions with $x, y, z \in \mathbb{Z}$ and $xyz \neq 0$.*

Here *regular* means that p does not divide the class number of the cyclotomic field $\mathbb{Q}(\zeta_p)$, where ζ_p is a primitive p-th root of unity (i.e. $\zeta_p^p = 1$ and $\zeta_p^k \neq 1$ for any $0 < k < p$). For


© Alex J. Best, Christopher Birkbeck, Riccardo Brasca, and Eric Rodriguez Boidi;
licensed under Creative Commons License CC-BY 4.0

example, the only irregular primes less that 100 are 37, 59 and 67. It is worth noting that currently there is no proof that there are infinitely many regular primes, but it is expected that roughly 60% of primes are regular.

In this short paper we report on ongoing work, the `flt_regular` project, to formalise Kummer's proof using the Lean theorem prover [6]. We build upon the *mathlib* library [7] is a large library of formalized mathematics for Lean. This library contains, for example, the definition of a number field and that of its ring of integers. More specifically for this work, the definitions and basic lemmas about class groups and class numbers have already been formalised [3], including that the class number of a number field is finite, a nontrivial fact that is required to even define regular primes properly.

Kummer's proof can be split into two cases, depending on if p is allowed to divide xyz in the theorem, or not. These are known in the field as the *first case* and the *second case*:

▶ **Theorem 2** (Case I). *Let p be a regular (odd) prime. Then $x^p + y^p = z^p$ has no solutions with $x, y, z \in \mathbb{Z}$ and $\gcd(xyz, p) = 1$.*

Case II then changes $\gcd(xyz, p) = 1$ to $\gcd(xyz, p) = p$, and together the two cases imply Kummer's theorem. While the proofs of the first and second case are broadly similar, and use many of the same techniques, results, and ideas, the proof of the second case uses more delicate results about units in cyclotomic fields (including one known as Kummer's lemma), and is therefore more difficult to formalise, even though many underlying results are the same. The formalisation of the second case is work in progress, so we will focus on the first case from now on.

These results are by now viewed as classical results in algebraic number theory and are covered in a number of works, we have followed the standard reference [11] for the most part. We also made use of a *blueprint*, an informal document included with the project covering the formalization targets in sufficient detail that the formalization progress could be tracked against it.

For large parts of this project the formalisation process and the process of adding the results to *mathlib* were done almost in parallel, see Section 4 below for more details.

2 Cyclotomic fields

The formalisation begins with the definition of cyclotomic fields and of more general of cyclotomic extensions. This is necessary to even define regular primes, and also appear in the first step of the high level overview of Kummer's proof. The basic mathematical idea is to work in the field $\mathbb{Q}(\zeta_p)$, that is the field obtained by adding to $\mathbb{Q}$ a primitive p-th root of unity (in $\mathbb{C}$ say). In such a field the left-hand-side of Fermat's equation can be written as

$$x^p + y^p = (x + y)(x + \zeta_p y) \cdots (x + y\zeta_p^{p-1}).$$

Using this, one can deduce information about the rational, and integral, solutions of Fermat's equation. The arithmetic properties of $\mathbb{Q}(\zeta_p)$ (such as how integral elements decompose as products of prime elements) are much more complicated than those of $\mathbb{Q}$. Studying such questions is the main subject of algebraic number theory. Fortunately for us, *mathlib* already contains many of the basic definitions the we will need, such as algebras, number fields, rings of integers and class groups. Unfortunately, *mathlib* did not contain examples of non-trivial number fields, so these definitions also served as a good test of the existing API.

Let A and B be commutative rings. For an A-algebra B and a set S of positive natural numbers, we say that B is a S-cyclotomic extension, if for every $n \in S$ there exists a primitive n-th root of unity in B and moreover that B is generated over A by the n-th roots of unity (for $n \in S$). We hence define a class `is_cyclotomic_extension`, now part of *mathlib*.

```
@[mk_iff] class is_cyclotomic_extension : Prop :=
(exists_prim_root {n : ℕ+} (ha : n ∈ S) : ∃ r : B, is_primitive_root r n)
(adjoin_roots : ∀ (x : B), x ∈ adjoin A { b : B | ∃ n : ℕ+, n ∈ S ∧
    b ^ (n : ℕ) = 1 })
```

The choice of working with n in $\mathbb{N}+$ rather than in $\mathbb{N}$ is motivated by the fact that, even if it requires us to insert certain coercions (for example to say that n is prime), the 0-th cyclotomic extension is not well behaved and theorems have a neater statement when that possibility is excluded.

We then define `cyclotomic_field n K`, where n is as above and K a field, and we prove that the corresponding field extension is an instance of the `is_cyclotomic_extension` class. To be precise, we define `cyclotomic_field n K` as the splitting field over K of the n-cyclotomic polynomial.

```
@[derive [field, algebra K, inhabited]]
def cyclotomic_field : Type w := (cyclotomic n K).splitting_field
```

Here, the `derive` attribute makes the `field`, `algebra K` and `inhabited` instances from `splitting_field` apply to `cyclotomic_field`. The `is_cyclotomic_extension` instance must then be proved manually.

Mathematically, using a predicate in the way we use the class `is_cyclotomic_extension` is uncommon, as one usually only works with the specific example `cyclotomic_field n K` (and indeed all n-th cyclotomic extensions of a field K are isomorphic if $n \neq 0$ in K), but having a characteristic predicate is essential in the formalisation process, for example to state that subextensions of a given cyclotomic extension generated by roots of unity are still cyclotomic, and to be able to apply lemmas to them. We prove several results about cyclotomic extensions and importantly, we prove that if S is finite and K is a number field, then any S-cyclotomic extension of K is again a number field. This allows us to define the usual cyclotomic number fields, such as $\mathbb{Q}(\zeta_n)$.

After the setup defining such fields is done, the main task is then to prove that the ring of integers of $\mathbb{Q}(\zeta_n)$ is $\mathbb{Z}[\zeta_n]$. In general, for a field K, its ring of integers $\mathcal{O}_K$ is the set of elements of K that are roots of a monic polynomial with coefficients in $\mathbb{Z}$. In particular, the inclusion $\mathbb{Z}[\zeta_n] \subseteq \mathcal{O}_{\mathbb{Q}[\zeta_n]}$ is clear, but equality is a nontrivial result that is specific to cyclotomic extensions, and we proved it only when $n = p^k$ is a power of a prime. Proving only this case is sufficient for us, indeed to prove Kummer's theorem only the case $n = p$ is needed, and in addition the general case makes use of this result for the prime-power case. This lemma is therefore a natural candidate for inclusion in a library that aims to include results in as much generality as possible, i.e. *mathlib* in the case of Lean code. To prove that $\mathcal{O}_{\mathbb{Q}(\zeta_{p^k})} = \mathbb{Z}[\zeta_{p^k}]$ we had to add a considerable amount of mathematics to *mathlib*, and this was the first significant milestone of the project. This required expanding the existing API for the norm and trace of elements of a number field, defining the discriminant of a number field and proving results relating discriminants to bases of rings of integers, etc., all material that would appear in a first course on algebraic number theory. Amongst the required results, we proved that, for $k > 0$ and $p > 2$ a prime, the discriminant of $\mathbb{Q}(\zeta_{p^k})$ is $(-1)^{\varphi(p^k)/2} p^{p^{k-1}((p-1)k-1)}$, with φ the Euler's totient function. Moreover, for $p = 2$ and $k > 1$ the same formula holds. In *mathlib* we can use the fact that as *natural numbers*, we have by convention, that $1/2 = 0$ and $0 - 1 = 0$. This allows us to give a simple formula for the discriminant that applies in all cases:

```
lemma discr_prime_pow {p : ℕ+} {k : ℕ} {K L : Type*} {ζ : L} [field K]
  [field L] [algebra K L] [hcycl : is_cyclotomic_extension {p ^ k} K L]
  [hp : fact (p : ℕ).prime] (hζ : is_primitive_root ζ ↑(p ^ k))
  (hirr : irreducible (cyclotomic (↑(p ^ k) : ℕ) K)) :
  discr K (hζ.power_basis K).basis = (-1) ^ (((p ^ k : ℕ).totient) / 2) *
    p ^ ((p : ℕ) ^ (k - 1) * ((p - 1) * k - 1))
```

Note that here K and L are only assumed to be fields (so, for example, they could have characteristic p), which is why we need the additional assumptions, that hold if $K = \mathbb{Q}$ and $L = \mathbb{Q}(\zeta_{p^k})$.

3 About the proof of case I

One issue that we often encountered came from a typeclass diamond resulting from multiple inheritance paths when working with a field of characteristic zero (see [1] for more on how this sort of issue arises and is resolved). Our issue arises as `cyclotomic_field n K` is endowed with the instance `algebra K (cyclotomic_field n K)`, but if $K = \mathbb{Q}$, then there is another instance `algebra ℚ (cyclotomic_field n ℚ)`, coming from the fact that `cyclotomic_field n ℚ` is a characteristic zero field, and hence a $\mathbb{Q}$-algebra. These two $\mathbb{Q}$-algebra structures were propositionally, but not definitionally, equal. This caused some friction when using results stated via the more general instance but Lean finds the one resulting from characteristic zero. However, we were able to resolve this issue by changing the way that `splitting_field` is defined.

Previously, these instances were lifted from a base field to the splitting field by direct induction, and this gave us no definitional control of the field of this structure (specifically, the `qsmul` and `rat_cast` fields in `field (splitting_field f)`). The fix for this was to lift every field individually and put them together later, so that we can control these crucial definitional equalities. As we are lifting to a quotient, we need to take care that these operations are well defined, and this led to the introduction of `distrib_smul`: a typeclass carrying the action of one type on another weak enough that the "obvious" map $\mathbb{Q} \times K \to K$ satisfies it, but strong enough to guarantee that lifting this to the map $\mathbb{Q} \times K[X]/(p(X)) \to K[X]/(p(X))$ is well defined.

Having developed the cyclotomic field framework we then moved to proving the technical number theoretic lemmas which are required in the proof of case I. These involve the careful study of units in the rings of integers of cyclotomic fields as well as certain ideals in these rings. Before describing the necessary lemmas, let us highlight a recurring issue when dealing with units.

Consider the following situation. Let R be an integral domain and K its field of fractions. Given a unit $r \in R^\times$, we may want to think of r as an element of $R^\times$, R or K. In *mathlib* these are all different types so we need coercions maps between them. Now for r as an element of $R^\times$ and K, we can easily work with its inverse, i.e, we can consider r^{-1}, but this is not possible when considered as an element R, since R is only a ring, so in general elements don't have a multiplicative inverse, but when coerced one step further to elements of the field K we are once again able to define a well defined inverse function. These issues arise often when working with ideals, which are submodules of R but when the proofs require one to use units in several places. Note that it is not clear how to set up `simp` lemmas that normalise elements, since sometimes we want to move from R to K and sometimes from R to $R^\times$. The solution to this is to have simple lemmas relating the images of r^{-1} in R and K.

```
lemma coe_coe_inv (u : Rˣ) : ((u : R) : K)⁻¹ = ((u⁻¹ : Rˣ) : R)
```

As an example of where this is used we have the following lemma, where K will denote $\mathbb{Q}(\zeta_p)$ and $R = \mathbb{Z}[\zeta_p]$. We will also denote ζ_p by ζ; note that `hζ.unit'` is the same as ζ, but considered in the units $R^\times$.

▶ **Lemma 3.** *Let $p \neq 2$ be a prime. Then every unit $u \in \mathbb{Z}[\zeta_p]^\times$ can be written as $u = x\zeta_p^n$ for some $n \in \mathbb{Z}$ and $x \in \mathbb{Z}[\zeta_p]^\times$ such that $x \in \mathbb{R}$.*

```
lemma unit_lemma_gal_conj (h : p ≠ 2) (hp : (p : ℕ).prime) (u : Rˣ)
  (hζ : is_primitive_root ζ p) :
  ∃ (x : Rˣ) (n : ℤ), is_gal_conj_real p (x : K) ∧
    (u : R) = x * (hζ.unit' ^ n : Rˣ)
```

Here the integer n cannot be supposed to be in $\mathbb{N}$, so x must be an element of a group (namely $R^\times$), to allow integer-valued powers. On the other hand, the existence of $x \in K$ is not enough, so we both need R and $R^\times$. Finally, the Galois group acts on K, so we really need the three different types to state the lemma cleanly. Note also that we state the informal condition that $x \in \mathbb{R}$ to `is_gal_conj_real p x`, which says that x is fixed under complex conjugation, where complex conjugation is thought of as an element of $\mathrm{Gal}(\mathbb{Q}(\zeta_p)/\mathbb{Q})$. In particular, we can avoid the non-canonical coercion into the real numbers used implicitly in standard proofs by reformulating what it means to be "real" in this setting.

This situation is perhaps not yet fully satisfactory, as manually rewriting to convert between the same element coerced into different types forces us to work at a lower level than we would when discussing the material informally. Another solution such as more automation may be better in the long term.

For brevity, we will not list all of the lemmas required to prove case I, but full details can be found on our project blueprint here: `https://leanprover-community.github.io/flt-regular/`. The final result we prove is

```
theorem caseI {a b c : ℤ} {p : ℕ} [fact p.prime] (h : is_regular_prime p)
  (caseI : ¬ ↑p ∣ a * b * c) :
  a ^ p + b ^ p ≠ c ^ p
```

and a full sorry-free proof can be found in `https://github.com/leanprover-community/flt-regular/`. We note that the case of $p = 3$ can be done without the tools we have formalised here. In fact, in this case, the result was formalised by Ruben van de Velde at `https://github.com/Ruben-VandeVelde/flt` using elementary methods.

We end this section with the definition of `is_regular_prime`. Here `is_regular_number` says that a positive integer n is regular if n is coprime to the size of the class group of $\mathbb{Q}(\zeta_n)$ from which we define `is_regular_prime` as the condition that a prime number is regular. We currently have a proof that $p = 2$ is a regular prime, but in general proving that a certain prime is regular (with our current definition) requires us to compute the relevant class number, which in general is difficult to do in *mathlib*, this is something that will be addressed in the proof of case II, which will relate being regular to a more easily checkable condition (factorizations of certain Bernoulli numbers). Explicit calculations of class numbers of quadratic fields have been formalized [2], and while the cyclotomic fields we use here are in general not quadratic, some of the same techniques may be of use when calculating class numbers of cyclotomic fields directly.

```
def is_regular_number [hn : fact (0 < n)] : Prop :=
n.coprime
  (card (class_group (ring_of_integers (cyclotomic_field ⟨n, hn.out⟩ ℚ))))
```

4 Integration to *mathlib*

While working on a mathematical formalization such as this one, newly introduced material is often not stable. For instance, new definitions are often changed as working with them reveals deficiencies, and proof strategies are factored out into common lemmas or abstractions when they are recognized after being seen several times. This means that when working on such a project new material is initially not ready for use outside of the project as it may change radically to suit the needs of the project. Nevertheless contributing material to a large library is a way to ensure continued maintenance of the code, especially when the upstream library changes. Thus contribution to a library may be desirable when the code is sufficiently mature, despite the fact that adding such material to a large library requires external review and may take time.

One slightly unusual aspect of our work is that we tried to include our results in *mathlib* almost in real time, keeping the two projects closely in sync. This is in contrast to many other similar projects where first the main theorem is formalised in its entirety and then one begins the process of adding the results to *mathlib*, which usually results in a great deal of modifications to the original code. For example the *Perfectoid Project* [4] and the *Liquid Tensor Experiment* [5] both have huge `for_mathlib` folders with a lot of formalised mathematics that in principle is supposed to be integrated into *mathlib*, but the code does not yet have the required standards of quality and generality. This state is often reached as authors do not have the time required to polish the code to the standard required and open PRs to contribute it. Developing against a library with little to no backwards compatibility maintained such as *mathlib* then means that maintaining the code of a large project can be a painful job. For example, the *Perfectoid Project* is essentially stuck to a very old version of *mathlib*, and updating it to the latest version is a nearly impossible task.

Our approach is to have a folder `ready_for_mathlib` where we put as much as results as possible, opening PRs immediately. Even if this means sometimes proving certain results in unneeded generality, we think this is the best strategy for a medium-sized project as this one. Moreover, this also implies that our code is up to the standard of *mathlib*, that are usually very high. This kept the size of the `flt_regular` project relatively small, but for example the whole folder `number_theory/cyclotomic` (that is around 2000 lines of code) in *mathlib* was written as a byproduct of our work. One other side effect is that updating *mathlib* is a rather easy process: we are indeed using the latest version and we plan to keep doing so. In practice we opened (and had accepted) more than 110 PRs, in various areas of mathematics, ranging from linear algebra to number theory. A partial history of the PRs opened can be seen at `https://github.com/leanprover-community/flt-regular/wiki`.

In order for large ecosystems of formal proofs, such as Lean's mathlib and surrounding libraries, to continue to scale and cover a significant portion of graduate level mathematics it seems more automation will be necessary to ease the contribution and organisational burden. The fields of program repair and automated refactoring (and more specifically the burgeoning field of proof repair [9], with more emphasis on mathematical proofs) provide a model for what should be possible and useful. For instance when a large library that a project such as `flt_regular` depends on is updated, an automated summarisation of changes and required

modifications (or even automated patch creation and application) would reduce the manual effort keeping up to date with a rapidly changing upstream. Automated style fixers and code improvers (to simplify the process of golfing and generalizing working proofs or to avoid anti-patterns) would reduce the effort required to contribute functional but less mature proofs to a standard library. Automation to help more specifically with moving code between libraries, by situating it correctly and updating local import paths would also improve the workflow.

5 Future work

The next step in our work will be to give a full proof of Fermat's Last Theorem in the regular case. The main obstacle here is to prove Kummer's lemma:

▶ **Theorem 4.** *Let p be a regular prime and let $u \in \mathbb{Z}[\zeta_p]^\times$. If $u \equiv a \mod p$ for some $a \in \mathbb{Z}$, then there exists $v \in \mathbb{Z}[\zeta_p]^\times$ such that $u = v^p$.*

There are several ways to prove this lemma, with modern approaches using class field theory. For our purposes this approach would take is too far afield from our final goal. The first step will be to use an alternative definition of regular prime, which instead of asking that p does not divide the class number of $\mathbb{Q}(\zeta_p)$ asks that p does not divide the numerator of certain Bernoulli numbers. This definition also has the added benefit that it is easy to check that a prime is regular, since Bernoulli numbers are easy to compute (and this is already in *mathlib*). This then leaves the task of checking that these definitions are equivalent, which can be done without using class field theory, but will still require significant work. Following classical proofs the main obstacle in proving this equivalence of definitions (and Kummer's lemma) will be the need to understand the image in the p-adic completion of K the logarithm of certain units. Amongst other things the final proof of case II will require the formalisation of p-adic completions of number fields and their extensions. Furthermore we will require analytic results for p-adic logarithms and their links to Bernoulli numbers.

Several results relating Bernoulli numbers to values of p-adic L-functions have been formalized by Narayanan [8], these results may form a nontrivial part of the formalization of case II, depending on the approach taken.

References

1 Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, and Kazuhiko Sakaguchi. Competing inheritance paths in dependent type theory: A case study in functional analysis. In *Automated Reasoning: 10th International Joint Conference, IJCAR 2020, Paris, France, July 1–4, 2020, Proceedings, Part II*, pages 3–20, Berlin, Heidelberg, 2020. Springer-Verlag. doi:10.1007/978-3-030-51054-1_1.

2 Anne Baanen, Alex J. Best, Nirvana Coppola, and Sander R. Dahmen. Formalized class group computations and integral points on mordell elliptic curves. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2023, pages 47–62, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3573105.3575682.

3 Anne Baanen, Sander R. Dahmen, Ashvni Narayanan, and Filippo A. E. Nuccio Mortarino Majno di Capriglio. A Formalization of Dedekind Domains and Class Groups of Global Fields. *J. Autom. Reason.*, 66(4):611–637, 2022. doi:10.1007/s10817-022-09644-0.

4 Kevin Buzzard, Johan Commelin, and Patrick Massot. Formalising Perfectoid Spaces. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and*

Proofs, CPP 2020, pages 299–312, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372885.3373830.

5 Johan Commelin, Adam Topaz, et al. The Liquid Tensor Experiment. Github, 2022. URL: https://github.com/leanprover-community/lean-liquid.

6 Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A Metaprogramming Framework for Formal Verification. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017. doi:10.1145/3110278.

7 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.

8 Ashvni Narayanan. Formalization of p-adic L-functions in Lean 3, 2023. arXiv:2302.14491.

9 Talia Ringer. *Proof Repair*. PhD thesis, U. Washington, 2021. URL: https://www.proquest.com/dissertations-theses/proof-repair/docview/2568297410/se-2.

10 Richard Taylor and Andrew Wiles. Ring-theoretic properties of certain Hecke algebras. *Ann. of Math. (2)*, 141(3):553–572, 1995. doi:10.2307/2118560.

11 Lawrence C. Washington. *Introduction to cyclotomic fields*, volume 83 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, second edition, 1997.

12 Andrew Wiles. Modular elliptic curves and Fermat's last theorem. *Ann. of Math. (2)*, 141(3):443–551, 1995. doi:10.2307/2118559.

Implementing More Explicit Definitional Expansions in Mizar

Adam Grabowski
Institute of Computer Science, University of Białystok, Poland

Artur Korniłowicz
Institute of Computer Science, University of Białystok, Poland

Abstract

The Mizar language and its corresponding proof-checker offers the tactic of definitional expansions in proof skeletons. This apparatus is rather fragile in the case of intensive overloading of notions (which is widely observed e.g. in the field of algebra, but it is also present in the more fundamental set-theory contexts). We propose the extension of this mechanism: the change should offer users the more precise control over expansions via choosing the right definitional variant for the proof under consideration, still letting the authors to retain the more conservative approach. As a rule, the change will affect new Mizar texts, but obviously, it allows also for solving some context conflicts caused by the original approach in the Mizar repository. The usefulness of our approach is shown by a number of experiments carried out within MML, which is also affected by the change.

2012 ACM Subject Classification Theory of computation → Interactive proof systems

Keywords and phrases Mizar, definitions, proof assistants, mechanization of proof

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.37

Category Short Paper

Supplementary Material *Dataset (test files)*: `https://github.com/arturkornilowicz/unfolding` archived at `swh:1:dir:9d907e2d89509cc0fd1c73dd0d9adb9ced9af11b`

Acknowledgements The authors are grateful to the reviewers for their constructive comments. The Mizar processing has been performed using the infrastructure of the University of Białystok High Performance Computing Center.

1 Introduction

For almost fifty years the Mizar proof checking system together with its repository of mathematical texts – the Mizar Mathematical Library (MML) was developed to follow ordinary mathematicians' writing style. This included a formalization style mimicking human papers, the rigour in formal proofs acceptable by computer, and in the same time relative flexibility, allowing to choose author's favourite way of encoding. As systems evolved, computerized proof assistants are closer to real-life use, with growing popularity of many hammering techniques, but together with the more efficient proof automation we can loose human verbosity of the proof. This could affect an interactive proof assistant input script to be seen more like the collection of proof obligations for a theorem prover.

In mathematics, many notions can be defined in different ways, e.g., lattices can be understood as abstract algebras with two binary operations over the same carrier, or as posets with binary suprema and binary infima. The correspondence between such alternative definitions is usually expressed in the form of theorems or corollaries. Moreover, the same relations between certain types of mathematical objects can be expressed using various symbolisms specific for the types of these objects. If we take for example the notion of the equality for sets, relations, functions, finite sequences, etc., it can be expressed using either


© Adam Grabowski and Artur Korniłowicz;
licensed under Creative Commons License CC-BY 4.0

Table 1 Definitional variants of the equality predicate.

$\forall_{A,B} : (\forall_x : x \in A \Leftrightarrow x \in B) \Leftrightarrow A = B$	(1)
$\forall_{R_1,R_2} : (\forall_{x,y} : (x,y) \in R_1 \Leftrightarrow (x,y) \in R_2) \Leftrightarrow R_1 = R_2$	(2)
$\forall_{f_1,f_2} : (\mathrm{dom}(f_1) = \mathrm{dom}(f_2) \wedge \forall_x : x \in \mathrm{dom}(f_1) \Rightarrow f_1(x) = f_2(x)) \Leftrightarrow f_1 = f_2$	(3)
$\forall_{f_1:A\to B,f_2:A\to B} : (\forall_a : a \in A \Rightarrow f_1(a) = f_2(a)) \Leftrightarrow f_1 = f_2$	(4)
$\forall_{f_1:A\times B\to C,f_2:A\times B\to C} : (\forall_{a,b} : a \in A \wedge b \in B \Rightarrow f_1(a,b) = f_2(a,b)) \Leftrightarrow f_1 = f_2$	(5)
$\forall_{F_1,F_2} : (\mathrm{len}(F_1) = \mathrm{len}(F_2) \wedge \forall_{n\in\mathbb{N}} : 1 \leq n \leq \mathrm{len}(F_1) \Rightarrow F_1(n) = F_2(n)) \Leftrightarrow F_1 = F_2$	(6)

arbitrary sets, or ordered pairs, or domains and function applications, or lengths of sequences, respectively. The corresponding statements could be represented in Table 1, where A, B, C are sets, R_1, R_2 are binary relations, f_1, f_2 are functions, and F_1, F_2 are finite sequences. Due to the Mizar subtyping, the choice of the variant strictly depends on the type arguments. For mathematicians, such differences are negligible and so equivalent domains and symbolisms can be used interchangeably and mixed fluently. In the case of the above listed variants of the equality predicate [5], it is usually not needed to express directly which form of the statement should be used to justify a certain goal. The situation is different, however, when one does computer-aided mathematics under the control of proof-assistants. No matter which system is being used, authors are always supposed to indicate, in a more or less precise way, which version of the equality the program should apply to prove a statement in a given context.

In the Mizar system [1, 4], there are two methods to indicate which statements should be used in the proof of a given inference: either explicit references to theorems or schemes (theorems with free second-order variables, e.g., the scheme of separation), or using so-called definitional expansions, when the verifier automatically tries to find out and utilize appropriate definitions among all accessible definition statements. The latter are used in two different contexts: to control the logical structure of proofs and to enhance the computational power of the verifier [8]. In this paper we describe an extension of the language and the Mizar proof checker that allows authors to explicitly declare which definitions (of the predicate, with a keyword `pred`) should be expanded and according to which definitions proofs must be verified. This new feature will give authors more control over the structures of proofs which they send for verification.

The implementation of this extension requires changes in almost all basic modules of the Mizar verifier: Parser controlling the lexical structure of a given article and generating the parse tree of the text; MSM Processor – which identifies labels, variables, and constants; Analyzer – performing type checking, resolving ambiguities and identifying objects and operations; and Reasoner, controlling structures of proofs. The communication between these modules is realized by several `XML` files; from our viewpoint, the most important files are those created by Parser and MSM Processor: Parser generates `.wsx` files which represent parse trees of processed articles. Then, MSM Processor generates `.msx` files enriching `.wsx` files by the information about used labels, variables, and constants. MSM, WSM, ESM, which stand for Weakly Strict, More Strict, and Even more Strict Mizar respectively, are various XML data representation formats of the Mizar article, where ESM combines both syntactic and semantic data and can be used independently by the dedicated Mizar proof checking software [3]. Because our work extends the syntax of the Mizar language and semantics of the system, we modified the grammar of these files adding `XML` elements to store information about explicit references to expanded definitions.

Related work. In Coq, the unfold tactic replaces a defined term in the goal with its definition using syntax `unfold <term>`. Unfolding of definitions (transparent, not opaque, constants) is realized as δ-reductions (automatically), which replace variables defined in local contexts

or constants defined in the global environment with their values. Unfolding on a hypothesis in the context can be done with the syntax `unfold <term> in <hypothesis>`. In Isabelle, `unfold` expands the given definitions throughout all goals; any chained facts provided are inserted into the goal and subject to further rewriting. First, the given equations are used directly for rewriting; then, the equations are passed through the attribute `abs_def` – to ensure that definitions are fully expanded, regardless of the actual parameters that are provided. Exactly the rules specified as arguments are considered in order to optional fine-tuned decomposition of a proof problem [13]. In Lean, there are several ways to utilize unfolding definitions and none of them is affected by imports (unlike in the case of Mizar). Definitions can be unfolded explicitly on demand of users or can also be unfolded implicitly, using the definitional equality relation that is built in to dependent type theory. Users can use e.g., `unfold dname` or `simp [dname]`, replacing all occurrences of `dname` with its definition according to the *definitional equations* – theorems proven automatically by Lean with each definition.[1] Implicit unfolding of definitions can be used through an `intro` rule, which is valid if the goal is made to become a general quantified formula.

2 Definitional expansions in proof skeletons

A well-known idea standing behind the process of creating proof interactively is a *proof sketch*, finding all building blocks (hints) needed to follow the solution. This should facilitate creating the proper structure of the proof (in Mizar we call this structure of proofs *a proof skeleton*, in some other declarative proof languages this is referred to as *a proof outline*). As the next step, authors should decide which available lemmas could be used to justify intermediate proof steps (some of them can be reconstructed automatically). The choice of these auxiliary facts has a big influence on the length and complexity of proofs. For example, to prove the equality of two functions with the same domain, it is better to use the theorem (4) from Table 1 than (3), because the reasoning about domains can be omitted. It is also much better than (2), as the representation of those functions as sets of pairs can be avoided. Moreover, proofs based on (3) and (4) require implications while a proof based on (2) requires a bi-implication. In practice, mathematical papers often do not mention explicitly a theorem used in a proof, relying on the context knowledge of the readers. The Mizar system tries to mimic mathematicians' style and in some contexts (when current goals to be proven are atomic formulae) allows users not to refer explicitly to theorems which are needed to be used in order to justify given statements; instead, it tries to find out, use and accept appropriate definitions among all accessible definitions.

Definitional expansions allow to construct proofs (proof skeletons) according to the formula which defines the notion or property being proven in a given goal. It can be seen as an implicit reference to so-called *definitional theorems* – automatically generated for any new definitions. As an example we can compare two proofs of the monotonicity of the (relation) restriction, using two versions of the inclusion: first, using the general form valid for arbitrary sets – see Listing 1 [11] and its variant for binary relations (Listing 2) [14]. Listing 3 presents alternative, rather clumsy unfolding available: although valid and potentially useful, seems not to be widely used in the MML (only six explicit references overall). The MML item `RELSET_1` itself is frequently imported, because its second definition – the predicate of the equality for relations on sets – is quite handy, but then there is no mechanism preventing unneeded implicit expansions. In total, this file is imported in `*.def` definientia files by 331 files; it is one fourth of 1275 in the case of `RELAT_1` (but the ratio of definitions is 2 vs. 17).

[1] As one of the reviewers pointed out, from our perspective the related mechanism is the same as in the case of Coq, relying on δ-reduction.

Listing 1 Definition of inclusion of sets

```
definition let A,B be set;
 pred A c= B means :: TARSKI:def 3
 for x being object st
   x in A holds x in B;
end;
```

Listing 2 Definition of inclusion of relations

```
definition let P,R be Relation;
 redefine pred P c= R means:: RELAT_1:def 3
 for a,b being object st
   [a,b] in P holds [a,b] in R;
end;
```

Listing 3 Definition of inclusion of typed relation

```
definition let X,Y be set; let R for Relation of X,Y; let Z be set;
  redefine pred R c= Z means                    :: RELSET_1:def 1
  for x being Element of X, y being Element of Y holds
    [x,y] in R implies [x,y] in Z;
end;
```

Listing 4 contains some skeletal statements: `let x be object;` and `assume x in R|X;` which start the expansion of the inclusion according to the version related to arbitrary sets. Listing 5 via `let a,b be object;` utilizes the version specific for binary relations. An advantage of the latter approach is that we start the expansion with introducing two constants `a` and `b`, which are necessary to reason about the restriction at once. In the former proof, the related constants `a` and `b` had to be introduced by decomposing the constant `x`. The latter proof is evidently more compact.

Listing 4 The proof with `TARSKI` expansion

```
for R being Relation
  for X,Y being set st X c= Y holds
    R | X c= R | Y
  proof
    let R be Relation;
    let X,Y be set such that
A1: X c= Y;
    let x be object;
:: expansion of inclusion predicate starts
    assume
A2: x in R | X; then
    consider a,b being object such that
A3: x = [a,b] by RELAT_1:def 1;
A4: [a,b] in R by A2,A3,RELAT_1:def 11;
    a in X by A2,A3,RELAT_1:def 11;
    then a in Y by A1,TARSKI:def 3;
    hence x in R|Y by A3,A4,RELAT_1:def 11;
  end;
```

Listing 5 The proof with `RELAT_1` expansion

```
for R being Relation
  for X,Y being set st X c= Y holds
    R | X c= R | Y
  proof
    let R be Relation;
    let X,Y be set such that
A1: X c= Y;
    let a,b be object;
:: expansion of inclusion predicate starts
    assume
A2: [a,b] in R | X;


A3: [a,b] in R by A2,RELAT_1:def 11;
    a in X by A2,RELAT_1:def 11;
    then a in Y by A1,TARSKI:def 3;
    hence [a,b] in R|Y by A3,RELAT_1:def 11;
  end;
```

Each Mizar article is a plain text file consisting of two parts: *environment* (import section), and *text-proper*, with definitions, theorems, lemmas and other items supported by the Mizar language [4]. We focus on the `definitions` directive – a comma separated list of MML identifiers determining proof skeletons. The order of entities in this directive matters: when the statement written by the user does not match the current goal resulting from the logical structure of the statement being proved, the verifier goes through the list of file names listed in this directive, reads definitions from these articles, and tries to find those matching the current goal with the definiens. If such a definition is found, the current goal is replaced and the proof can be continued. In the example about the inclusion of restrictions of a relation to sets presented above, the first proof (related to the inclusion of sets) is allowed when the definition of the inclusion of sets labeled as `TARSKI:def 3` is matched before the definition of the inclusion of relations labeled as `RELAT_1:def 3`, that is, when `TARSKI` appears after `RELAT_1` on the `definitions` list; the second proof is correct in the opposite case.

3 Unfolding wrt chosen definitions

In this section we propose an extension of the Mizar language and the Mizar checker which allows authors to explicitly declare which definitions should be expanded and according to which definitions proofs must be verified. This new feature will give authors more control over the structures of proofs to be accepted by the verifier. Mizar texts are internally represented using a recursive *Blocks* and *Items* structure, where blocks are supposed to store lists of items, and items contain essential information and, if necessary, may contain one block. Among the items which may be tracked in the syntax of the language, the most important group are those representing proof skeletal elements, that is, *Generalization* (`let` for fixing variables), *Assumption* (with self-explanatory `assume`), *ExistentialAssumption* (similar, with `given`), *Exemplification* (presenting examples – `take`), and *Conclusion* (keywords `thus` or `hence`). All these items change the state of thesis and can be used adequately to the logical structure of proven statements.

As our goal is developing a way to explicitly indicate which definition should be automatically expanded, we propose to extend the Mizar language with the keyword `unfolding` and to extend the set of items related to proof skeletons with a new item *Unfolding*, which will be generated when Mizar keyword `unfolding` occurs within proofs. Then, we propose to extend the Mizar syntax with new grammar rules correspondingly (see Supplementary material for full version):

```
Skeleton-Item = Generalization | Assumption | Conclusion |
                        Exemplification | Unfolding .
Unfolding = ‘‘unfolding’’ References ‘‘;’’ .
```

Following these rules, a pseudo code of an exemplary proof text could look like `some reasoning; unfolding references; some reasoning;` where `references` used after the `unfolding` construction may contain references to either local or global definitions (imported from MML). When global definitions are referred to, filenames of articles where the definitions were introduced must be listed in both `theorems` and `definitions` declarations, where the earlier describes just the name space for file identifiers, and the latter forces the system to load the definitions and make them accessible and expandable within proofs. In the case when the list of used references has more than one reference to a definition, the references are processed one by one from left to right, and corresponding definitions are expanded.

As we mentioned before, the Mizar verifier generates a couple of `XML` files. Naturally, we proposed also corresponding extensions to the grammar of `.wsx` and `.msx` files. In `.wsx` file, each occurrence of the `unfolding` construction generates the `<Unfolding>` element within which we put the description of the list of used references.

```
<Item kind="Unfolding" position="20\11" endposition="20\33">
 <Unfolding>
  <Definition-Reference position="20\28" nr="2" spelling="RELAT_1"
                        number="3"/>
  <Local-Reference position="20\33" idnr="5" spelling="Lm1"/>
 </Unfolding>
</Item>
```

The attribute `position` stores the position of the `unfolding` keyword; the attribute `endposition` stores the position of the last reference used. The element `Unfolding` contains the list of all references used in the `unfolding` item. The elements `Definition-Reference` and `Local-Reference` keep information about a reference to a definition and a local statement, respectively. This information is later passed to `.msx` files, where references to local

Table 2 Top 17 of used expansions of definitions (only over 100 hits each).

Rk.	Definition	Symbol	Usage	Rk.	Definition	Symbol	Usage
1.	TARSKI:def 3	c=	13293	10.	RELAT_1:def 3	c=	139
2.	XBOOLE_0:def 10	=	2792	11.	LATTICE3:def 8	>=_than	132
3.	FUNCT_2:def 8	=	361	12.	ALGSTR_0:def 16	r-compl	129
4.	FUNCT_1:def 4	one-to-one	332	13.	STRUCT_0:def 1	empty	127
5.	XBOOLE_0:def 7	misses	209	14.	BVFUNC_1:def 12	'<'	125
6.	PBOOLE:def 2	c=	183	15.	RELAT_1:def 2	=	119
7.	LATTICE3:def 9	<=_than	183	16.	MEMBERED:def 14	=	115
8.	FUNCT_1:def 11	=	179	17.	MEMBERED:def 8	c=	106
9.	ALGSTR_0:def 11	r-compl	150		Total		33475

statements are disambiguated and then propagated to files representing deeper layers of information processed by the verifier. These changes do not directly interfere with the work of ordinary Mizar users as they can just avoid unfolding; however, for those using intermediate representation files [2, 6, 7, 10, 12], they are critical.

4 Experiments

The usefulness of our extension of the Mizar system was tested on the whole MML. To facilitate experiments we implemented generated reports on all definitional expansions performed during the verification of processed articles; the results are printed out into an XML file `filename.den`, where `filename` is the name of a given article. Each entry (within the root element `Positions`) in `.den` files is of the form: `<Position name="" nr="" line="" col=""/>` matching name and number of unfolded definition together with the starting position. This convinced us how often definitional expansions were effectively used and how many times each definition was expanded while processing each article. The total number of expansions is 33475 and the top 17 expanded definitions are presented in Table 2. The most often expanded definition is `TARSKI:def 3`, which is the definition of the inclusion of sets, with `XBOOLE_0:def 10` (the definition of the equality of sets expressed as the conjunction of two appropriate inclusions) as the second. These 17 primary positions sum up to total 18674, which shows that over half of all the expansions is represented in the table, i.e. the rest of dependencies make rather flat hierarchy. It should be compared with aditional 1997 explicit references for `TARSKI:def 3` and 1707 – for `TARSKI:2` (which is just extensionality of sets). Similarly, `XBOOLE_0:def 10` was called 1516 times. Most of them can be also changed to unfolding in this new implementation, making roughly ca. 5200 new uses of expansions.

As mentioned in Section 2, the Mizar verifier tries to expand definitions according to the order of filenames in the environment directive `definitions`, on last-come-first-serve basis (still, randomness could be accepted if notions overloading is only exceptional), so as our first experiment we sorted items in this environment directive according to the list `mml.lar` reflecting processing order of MML [9]. As a result we got 65 errors within 30 articles: even this most obvious, genetic ordering, causes serious problems: such clashes should be resolved in order to obtain a kind of canonical precedence. Further tests show that this is much too fragile – but as the frequent technique of obtaining your own environment declaration is just to copy it from some similar ones, the randomization is not the question of pure accident. To test the aforementioned system fragility, as another experiment we shuffled filenames in the `definitions` environment directive and put them in a random order. The experiment

was repeated: after each shuffling, all articles were verified[2]. Based on these results, we can observe that the order in which files are listed in the `definitions` environment directive definitely plays an important role. Previously, authors had to quite carefully construct the list of imports of definitions; with this new extension, they are allowed to indicate which definition should be expanded at given stages of proofs without paying special attention to the order in which definitions were imported.

5 Conclusions

The current version of the Mizar system (8.1.12) naturally supports building proofs according to logical structures of statements being proven. It also gives opportunity to modify them a bit using so-called *definitional expansions* using appropriate definitions among all accessible ones. However, there is a limitation caused by the order in which the definitions need to be imported. Our experiments show that the ordering of environment directives essentially matters; e.g. there are two substantial clashes well-known for authors: interchanging `RELAT_1` and `FUNCT_1` or placing `STRUCT_0` before fundamental classical (i.e., those based on pure set theory) part of the MML. Decades ago, the Mizar accomodator allowed to specify which single theorems should be imported (this made accomodator files just smaller and memory efficient), similar solution could be also feasible in the case of definitional expansions. In this work we developed a more practical workaround of that limitation: an extension of the Mizar checker which allow authors to precisely indicate according to which definitions proofs are being constructed. It seems to work well particularly in the case of notions with several redefined definienses. In some sense, the proposed implementation enriches the paradigm of declarative proof language, offering procedural elements. This is rather unusual for the system, but as the Mizar project turns fifty in 2023, maybe this justifies the need for some further experiments in this new direction.

References

1 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 Proceedings*, volume 9150 of *LNCS*, pages 261–279. Springer, 2015. doi:10.1007/978-3-319-20615-8_17.

2 Grzegorz Bancerek, Adam Naumowicz, and Josef Urban. System description: XSL-based translator of Mizar to LaTeX. In Florian Rabe, William M. Farmer, Grant O. Passmore, and Abdou Youssef, editors, *Intelligent Computer Mathematics – 11th International Conference, CICM 2018, Hagenberg, Austria, August 13–17, 2018, Proceedings*, volume 11006 of *Lecture Notes in Computer Science*, pages 1–6. Springer, 2018. doi:10.1007/978-3-319-96812-4_1.

3 Czesław Byliński, Artur Korniłowicz, and Adam Naumowicz. Syntactic-semantic form of Mizar articles. 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 11:1–11:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ITP.2021.11.

4 Adam Grabowski, Artur Korniłowicz, and Adam Naumowicz. Mizar in a nutshell. *Journal of Formalized Reasoning, Special Issue: User Tutorials I*, 3(2):153–245, 2010. doi:10.6092/issn.1972-5787/1980.

[2] Numbers of errors and files with errors obtained in each turn of the test are presented in suplementary materials; the number of errors varies between 516 and 672 in ca. 120 articles.

5 Adam Grabowski, Artur Korniłowicz, and Christoph Schwarzweller. Equality in computer proof-assistants. In Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki, editors, *Proceedings of the 2015 Federated Conference on Computer Science and Information Systems*, volume 5 of *Annals of Computer Science and Information Systems*, pages 45–54. IEEE, 2015. doi:10.15439/2015F229.

6 Cezary Kaliszyk and Karol Pąk. Isabelle import infrastructure for the Mizar Mathematical Library. In Florian Rabe, William M. Farmer, Grant O. Passmore, and Abdou Youssef, editors, *Intelligent Computer Mathematics – 11th International Conference, CICM 2018 Proceedings*, volume 11006 of *LNCS*, pages 131–146. Springer, 2018. doi:10.1007/978-3-319-96812-4_13.

7 Cezary Kaliszyk and Karol Pąk. Declarative proof translation (short paper). In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9–12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*, pages 35:1–35:7. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ITP.2019.35.

8 Artur Korniłowicz. Definitional expansions in Mizar. *Journal of Automated Reasoning*, 55(3):257–268, October 2015. doi:10.1007/s10817-015-9331-7.

9 Adam Naumowicz. Tools for MML environment analysis. In Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge, editors, *Intelligent Computer Mathematics – International Conference, CICM 2015 Proceedings*, volume 9150 of *LNCS*, pages 348–352. Springer, 2015. doi:10.1007/978-3-319-20615-8_26.

10 Karol Pąk. Combining the syntactic and semantic representations of Mizar proofs. In Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki, editors, *Proceedings of the 2018 Federated Conference on Computer Science and Information Systems, FedCSIS 2018*, volume 15 of *Annals of Computer Science and Information Systems*, pages 145–153. IEEE, 2018. doi:10.15439/2018F248.

11 Andrzej Trybulec. Tarski Grothendieck set theory. *Formalized Mathematics*, 1(**1**):9–11, 1990. URL: http://fm.mizar.org/1990-1/pdf1-1/tarski.pdf.

12 Qingxiang Wang, Chad E. Brown, Cezary Kaliszyk, and Josef Urban. Exploration of neural machine translation in autoformalization of mathematics in Mizar. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020*, pages 85–98, 2020. doi:10.1145/3372885.3373827.

13 Makarius Wenzel. *The Isabelle/Isar Reference Manual*, 2021. URL: https://isabelle.in.tum.de/doc/isar-ref.pdf.

14 Edmund Woronowicz. Relations and their basic properties. *Formalized Mathematics*, 1(**1**):73–83, 1990. URL: http://fm.mizar.org/1990-1/pdf1-1/relat_1.pdf.

Formalizing Almost Development Closed Critical Pairs

Christina Kohl
Universität Innsbruck, Austria

Aart Middeldorp
Universität Innsbruck, Austria

Abstract

We report on the first formalization of the almost-development closedness criterion for confluence of left-linear term rewrite systems and illustrate a problem we encountered while trying to formalize the original paper proof by van Oostrom.

2012 ACM Subject Classification Theory of computation → Rewrite systems; Theory of computation → Logic and verification

Keywords and phrases Term rewriting, confluence, certification

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.38

Category Short Paper

Supplementary Material *InteractiveResource (Isabelle Code)*: `http://informatik-protem.uibk.ac.at/ITP2023/`

Funding *Aart Middeldorp*: Austrian Science Fund (FWF) project I5943.

1 Introduction

Recently we formalized the well-known confluence criterion by van Oostrom based on development closed critical pairs of left-linear term rewrite systems in the proof assistant Isabelle/HOL [6]. Here we present an extension of this result which goes back to an observation by Toyama, namely that the condition on critical pairs can be weakened in case of overlays. This so-called *almost* development-closed criterion and its commutation version have now been integrated into the library IsaFoR[1] which enables the tool CeTA [5] to certify confluence and commutation proofs based on this criterion.

In Section 2 we recap some important definitions and basic results about term rewriting and proof terms. The latter are used to represent multi-steps as first-order terms in the formalized proof. In Section 3 we present a slightly adapted version of our recent formalization of the development-closed criterion presented in [3]. In Section 4 we first illustrate why we could not simply follow van Oostrom's paper proof for almost development-closedness in [9, 10]. Then we show how the proof in Section 3 can easily be extended to the more general version. Finally, we briefly describe the adaptations necessary for the commutation version of almost development-closed critical pairs.

HTML versions of the relevant Isabelle theory files can be found at

`http://informatik-protem.uibk.ac.at/ITP2023/`

and the main results in this paper are annotated by a ☑-symbol which directly links to the HTML presentation.

[1] `http://cl-informatik.uibk.ac.at/isafor`

© Christina Kohl and Aart Middeldorp;
licensed under Creative Commons License CC-BY 4.0

2 Preliminaries

We assume familiarity with the basics of term rewriting, as can be found in [1], and only recap some important definitions here. The *multi-step* relation $\multimap_{\mathcal{R}}$ is inductively defined on terms as follows:

- $x \multimap_{\mathcal{R}} x$ for all variables x,
- $f(s_1, \ldots, s_n) \multimap_{\mathcal{R}} f(t_1, \ldots, t_n)$ if $s_i \multimap_{\mathcal{R}} t_i$ for all $1 \leqslant i \leqslant n$, and
- $\ell\sigma \multimap_{\mathcal{R}} r\tau$ if $\ell \to r \in \mathcal{R}$ and $\sigma(x) \multimap_{\mathcal{R}} \tau(x)$ for all $x \in \mathcal{V}\mathsf{ar}(\ell)$.

A critical *overlap* $(\ell_1 \to r_1, p, \ell_2 \to r_2)_\sigma$ of two TRSs $\mathcal{R}_1$ and $\mathcal{R}_2$ consists of variants $\ell_1 \to r_1$ and $\ell_2 \to r_2$ of rewrite rules in $\mathcal{R}_1$ and $\mathcal{R}_2$ without common variables, a position $p \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(\ell_2)$, and a most general unifier σ of ℓ_1 and $\ell_2|_p$. From a critical overlap $(\ell_1 \to r_1, p, \ell_2 \to r_2)_\sigma$ we obtain a critical peak $\ell_2\sigma[r_1\sigma]_p \ {}_{\mathcal{R}_1}\!\leftarrow \ell_2\sigma[\ell_1\sigma]_p = \ell_2\sigma \to_{\mathcal{R}_2} r_2\sigma$ and the corresponding *critical pair* $\ell_2\sigma[r_1\sigma]_p \ {}_{\mathcal{R}_1}\!\leftarrow \rtimes \to_{\mathcal{R}_2} r_2\sigma$. Whenever $p = \epsilon$ we say $r_1\sigma \ {}_{\mathcal{R}_1}\!\leftarrow \rtimes \to_{\mathcal{R}_2} r_2\sigma$ is an *overlay*. A relation $\to$ is confluent if ${}^*\!\leftarrow \cdot \to^* \subseteq \to^* \cdot {}^*\!\leftarrow$. Two relations $\to_1$ and $\to_2$ commute if ${}^*_1\!\leftarrow \cdot \to_2^* \subseteq \to_2^* \cdot {}^*_1\!\leftarrow$. A relation $\to$ has the *diamond property* if $\leftarrow \cdot \to \subseteq \to \cdot \leftarrow$ and it is *strongly confluent* if $\leftarrow \cdot \to \subseteq \to^= \cdot {}^*\!\leftarrow$. We say that $\to_1$ and $\to_2$ *strongly commute* if ${}_1\!\leftarrow \cdot \to_2 \subseteq \to_2^= \cdot {}^*_1\!\leftarrow$. The following well-known results [1, Chapter 2] connect the diamond property and strong confluence (strong commutation) with confluence (commutation).

▶ **Lemma 1.** *Let $\to$, $\to_1$ and $\to_2$ be binary relations.*

1. *If $\to$ has the diamond property then $\to$ is confluent.*
2. *If $\to$ is strongly confluent then $\to$ is confluent.*
3. *If $\to_1 \subseteq \to_2 \subseteq \to_1^*$ and $\to_2$ is confluent then $\to_1$ is confluent.*
4. *Two strongly commuting relations commute.*
5. *Suppose $\to_1 \subseteq \to_{1'} \subseteq \to_1^*$ and $\to_2 \subseteq \to_{2'} \subseteq \to_2^*$. If $\to_{1'}$ and $\to_{2'}$ commute then $\to_1$ and $\to_2$ commute.* ◀

When applying this lemma to prove that (almost) development closed critical pairs imply confluence for left-linear TRSs, we instantiate $\to$ in the first (second) item with $\multimap$ to obtain confluence of $\multimap$. Then we can use the third item with the property $\to \subseteq \multimap \subseteq \to^*$ to establish confluence of $\to$. The fourth and fifth items are used for the commutation version.

We used proof terms ([7, Chapter 8]) to represent multi-steps for both the formalization in [3] and the extension presented here. We only recap some concepts here. For a more basic introduction including examples see [2, 3]. Proof terms are built from function symbols, variables, and rule symbols. We use Greek letters for rule symbols. If α is a rule symbol then $\mathsf{lhs}(\alpha)$ ($\mathsf{rhs}(\alpha)$) denotes the left-hand (right-hand) side of the rewrite rule denoted by α. Furthermore $\mathsf{var}(\alpha)$ denotes the list $(x_1, \ldots, x_n)$ of variables appearing in α in some fixed order. The length of this list is the arity of α. The list $\mathsf{vpos}(\alpha) = (p_1, \ldots, p_n)$ denotes the corresponding variable positions in $\mathsf{lhs}(\alpha)$ such that $\mathsf{lhs}(\alpha)|_{p_i} = x_i$. Given a rule symbol α with $\mathsf{var}(\alpha) = (x_1, \ldots, x_n)$ and terms $t_1, \ldots, t_n$, we write $\langle t_1, \ldots, t_n \rangle_\alpha$ for the substitution $\{x_i \mapsto t_i \mid 1 \leqslant i \leqslant n\}$. Given a proof term A, its source $\mathsf{src}(A)$ and target $\mathsf{tgt}(A)$ are computed by the following equations for $\mathsf{st} \in \{\mathsf{src}, \mathsf{tgt}\}$:

$$\mathsf{st}(x) = x \qquad \mathsf{st}(f(A_1, \ldots, A_n)) = f(\mathsf{st}(A_1), \ldots, \mathsf{st}(A_n))$$
$$\mathsf{src}(\alpha(A_1, \ldots, A_n)) = \mathsf{lhs}(\alpha)\langle \mathsf{src}(A_1), \ldots, \mathsf{src}(A_n) \rangle_\alpha$$
$$\mathsf{tgt}(\alpha(A_1, \ldots, A_n)) = \mathsf{rhs}(\alpha)\langle \mathsf{tgt}(A_1), \ldots, \mathsf{tgt}(A_n) \rangle_\alpha$$

Proof terms A and B are said to be *co-initial* if they have the same source. A proof term A over a TRS $\mathcal{R}$ is a witness of the multi-step $\mathsf{src}(A) \multimap_{\mathcal{R}} \mathsf{tgt}(A)$. Every multi-step is witnessed by a proof term.

▶ **Lemma 2.** *For any substitution σ, proof term context C, and proof term A we have*

$$\mathsf{src}(A\sigma) = \mathsf{src}(\mathsf{src}(A)\sigma) \qquad \mathsf{tgt}(A\sigma) = \mathsf{tgt}(\mathsf{tgt}(A)\sigma)$$
$$\mathsf{src}(C[A]) = \mathsf{src}(C[\mathsf{src}(A)]) = \mathsf{src}(C)[\mathsf{src}(A)] \qquad \mathsf{tgt}(C[A]) = \mathsf{tgt}(C[\mathsf{tgt}(A)])$$ ◀

The following lemma will be used to complete the proof of strong confluence of almost development closed TRSs.

▶ **Lemma 3.** *If $s \to^* t$ then $\mathsf{tgt}(C[s\sigma]) \to^* \mathsf{tgt}(C[t\sigma])$ for arbitrary proof term contexts C and arbitrary substitutions over proof terms σ.* ☑

Proof. Straightforward induction on proof term contexts and using the fact that the rewrite relation is closed under contexts and substitutions. ◀

The following labeling is used to measure the amount of overlap between co-initial proof terms:

$$\mathsf{src}^\sharp(A) = \begin{cases} A & \text{if } A \in \mathcal{V} \\ f(\mathsf{src}^\sharp(A_1), \dots, \mathsf{src}^\sharp(A_n)) & \text{if } A = f(A_1, \dots, A_n) \\ \mathsf{lhs}^\sharp(\alpha)\langle \mathsf{src}^\sharp(A_1), \dots, \mathsf{src}^\sharp(A_n)\rangle_\alpha & \text{if } A = \alpha(A_1, \dots, A_n) \end{cases}$$

where $\mathsf{lhs}^\sharp(\alpha) = \varphi(\mathsf{lhs}(\alpha), \alpha, 0)$ with $\varphi(t, \alpha, i) = t$ if $t \in \mathcal{V}$ and $\varphi(t, \alpha, i) = f_{\alpha^i}(\varphi(t_1, \alpha, i+1), \dots, \varphi(t_n, \alpha, i+1))$ if $t = f(t_1, \dots, t_n)$. The (partial) function ℓ extracts labels from function symbols: $\ell(f_{\alpha^n}) = \alpha^n$. The amount of overlap $\blacktriangle(A, B)$ between co-initial proof terms A and B is defined as $\blacktriangle(A, B) = |\mathcal{P}\mathsf{os}_L(A) \cap \mathcal{P}\mathsf{os}_L(B)|$ where $\mathcal{P}\mathsf{os}_L(A) = \{p \in \mathcal{P}\mathsf{os}(\mathsf{src}^\sharp(A)) \mid \ell(\mathsf{src}^\sharp(A)(p)) \text{ is defined}\}$. This corresponds exactly to the measure used in [9, 10]. The set $\mathsf{overlaps}(A, B)$ consists of all pairs (p, q) of function symbol positions in the common source s of A and B such that

(a) $\ell(\mathsf{src}^\sharp(A)(p)) = \alpha^0$, $\ell(\mathsf{src}^\sharp(B)(q)) = \beta^0$, and

(b) either $p \leqslant q$ and $\ell(\mathsf{src}^\sharp(A)(q)) = \alpha^{|q \setminus p|}$ or $q < p$ and $\ell(\mathsf{src}^\sharp(B)(p)) = \beta^{|p \setminus q|}$

for some rule symbols α and β. We define the following order on overlaps: $(p_1, q_1) \leqslant (p_2, q_2)$ iff $p_1 \leqslant p_2$ and $q_1 \leqslant q_2$. An *innermost overlap* of co-initial proof terms A and B is a maximal element in $\mathsf{overlaps}(A, B)$ with respect to $\leqslant$.

3 Development Closed Critical Pairs

In this section we briefly recap the already formalized confluence criterion based on development closed critical pairs. Compared to the presentation in [3] we were able to remove some unnecessary results about deletion and joins of proof terms from the formalization. The price for this shortening is a greater focus on possibly less intuitive results involving substitutions and contexts with proof terms. The big advantage however is that the extension to almost development closed critical pairs later on is straightforward. The following definition and theorem are due to van Oostrom [9, 10].

▶ **Definition 4.** *A TRS $\mathcal{R}$ is development closed if for every critical pair $s \rtimes t$ of $\mathcal{R}$ we have $s \mathrel{-\!\!\circ\!\!\to_{\mathcal{R}}} t$.*

▶ **Theorem 5.** *If a TRS $\mathcal{R}$ is left-linear and development closed then $\mathrel{-\!\!\circ\!\!\to_{\mathcal{R}}}$ has the diamond property.*

Formalized proof. Assume $t \mathrel{\circ\!\!-} s \mathrel{-\!\!\circ\!\!\to} u$ and let A be a proof term representing $s \mathrel{-\!\!\circ\!\!\to} t$ and let B be a proof term representing $s \mathrel{-\!\!\circ\!\!\to} u$. We show $t \mathrel{-\!\!\circ\!\!\to} v \mathrel{\circ\!\!-} u$ for some term v by well-founded induction on the amount of overlap between A and B.

- Base case: If $\blacktriangle(A,B) = 0$ then $A \,/\, B$ and $B \,/\, A$ are well-defined and represent the multi-steps $t \mathrel{-\!\!\circ\!\!\to} \mathsf{tgt}(A \,/\, B)$ and $u \mathrel{-\!\!\circ\!\!\to} \mathsf{tgt}(B \,/\, A)$ respectively. Since $\mathsf{tgt}(A \,/\, B) = \mathsf{tgt}(B \,/\, A)$ this proves the base case of the induction.
- Step case: Assume $\blacktriangle(A,B) > 0$. The induction hypothesis states that if A' and B' are two co-initial proof terms such that $\blacktriangle(A',B') < \blacktriangle(A,B)$ then there exists a term v and and multi-steps $\mathsf{tgt}(A') \mathrel{-\!\!\circ\!\!\to} v \mathrel{\leftarrow\!\!\circ\!\!-} \mathsf{tgt}(B')$. We show that $t \mathrel{-\!\!\circ\!\!\to} v \mathrel{\leftarrow\!\!\circ\!\!-} u$:
  1. First we select an innermost overlap (p,q) and assume without loss of generality that $q \leqslant p$. Let $q' = p \backslash q$ and α and β be the rule symbols at positions p and q in $\mathsf{src}(A)$ and $\mathsf{src}(B)$ such that $\ell(\mathsf{src}^\sharp(A)(p)) = \alpha^0$ and $\ell(\mathsf{src}^\sharp(B)(q)) = \beta^0$. Furthermore let $\mathsf{vpos}(\alpha) = (p_1,\dots,p_n)$, $\mathsf{var}(\alpha) = (x_1,\dots,x_n)$, $\mathsf{vpos}(\beta) = (q_1,\dots,q_m)$, and $\mathsf{var}(\beta) = (y_1,\dots,y_m)$ where we assume $\{x_1,\dots,x_n\} \cap \{y_1,\dots,y_m\} = \varnothing$ without loss of generality.
  2. Then we split the proof term A into two proof terms: First the single step $s \to t'$ represented by $\Delta_1 = s[\alpha(s|_{pp_1},\dots,s|_{pp_n})]_p$ and second the residual $A \,/\, \Delta_1$ witnessing $t' \mathrel{-\!\!\circ\!\!\to} t$ for some term t'. We do the same for B obtaining $\Delta_2 = s[\beta(s|_{qq_1},\dots,s|_{qq_m})]_q$ witnessing $s \to u'$ and the residual $B \,/\, \Delta_2$ witnessing $u' \mathrel{-\!\!\circ\!\!\to} u$ for some term u'.
  3. We define the substitution

$$\begin{aligned}\tau = \{\, x_i \mapsto \mathsf{lhs}(\beta)|_{q'p_i} \mid 1 \leqslant i \leqslant n \text{ and } q'p_i \in \mathcal{P}\mathsf{os}(\mathsf{lhs}(\beta)) \,\} \\ \cup \{\, y_j \mapsto lhs(\alpha)|_{q_j \backslash q} \mid 1 \leqslant j \leqslant m \text{ and } q_j \backslash q \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(\mathsf{lhs}(\alpha)) \,\}\end{aligned}$$

     which yields the critical peak $\mathsf{lhs}(\beta)[\mathsf{rhs}(\alpha)\tau]_{q'} \leftarrow \mathsf{lhs}(\beta)[\mathsf{lhs}(\alpha)\tau]_{q'} = \mathsf{lhs}(\beta)\tau \to \mathsf{rhs}(\beta)\tau$ [3, Lemma 7.2] and the position $q_\beta \in \mathcal{P}\mathsf{os}(B)$ such that $B = B[\beta(B_1,\dots,B_m)]_{q_\beta}$ and $\mathsf{src}(B)[\,]_q = \mathsf{src}(B[\,]_{q_\beta})$. I.e, q_β is the unique position of the rule symbol β in B corresponding to the critical peak.
  4. By the development closedness assumption we know that there exists a multi-step $\mathsf{lhs}(\beta)[\mathsf{rhs}(\alpha)\tau]_{q'} \mathrel{-\!\!\circ\!\!\to} \mathsf{rhs}(\beta)\tau$. Let D' be a proof term representing this multi-step.
  5. Next we define the substitution

$$\rho = \{\, y_j \mapsto B_j \mid 1 \leqslant j \leqslant m \,\} \cup \{\, x_i \mapsto \mathsf{lhs}(\beta)\langle B_1,\dots,B_m\rangle_\beta|_{q'p_i} \mid 1 \leqslant i \leqslant n \,\}$$

     and show that the proof term $B[D'\rho]_{q_\beta}$ witnesses a multi-step $t' \mathrel{-\!\!\circ\!\!\to} u$ [3, Lemma 7.7].
  6. We show $\blacktriangle(A \,/\, \Delta_1, B[D'\rho]_{q_\beta}) < \blacktriangle(A,B)$ [3, Lemma 7.8].
  7. The previous items allow us to apply the induction hypothesis to obtain multi-steps $t \mathrel{-\!\!\circ\!\!\to} v$ and $u \mathrel{-\!\!\circ\!\!\to} v$ for some common term v. ◀

4 Almost Development Closed Critical Pairs

Van Oostrom [9, 10] realized that the previous result could be strengthened analogously to Toyama's extension [8] for *almost* parallel closed term rewrite systems. The main observation is that by proving confluence of $\mathcal{R}$ via strong confluence of $\mathrel{-\!\!\circ\!\!\to}_{\mathcal{R}}$ instead of the diamond property of $\mathrel{-\!\!\circ\!\!\to}_{\mathcal{R}}$, the condition on overlays can be weakened to $\mathrel{-\!\!\circ\!\!\to} \cdot {}^*\!\leftarrow$ instead of $\mathrel{-\!\!\circ\!\!\to}$.

▶ **Definition 6.** *A* TRS $\mathcal{R}$ *is almost development closed if for every critical pair* $s \rtimes t$ *of* $\mathcal{R}$
1. $s \mathrel{-\!\!\circ\!\!\to} t$ *if* $s \rtimes t$ *is not an overlay,*
2. $s \mathrel{-\!\!\circ\!\!\to} \cdot {}^*\!\leftarrow t$ *if* $s \rtimes t$ *is an overlay.*

Since $s \mathrel{-\!\!\circ\!\!\to} t$ implies $s \mathrel{-\!\!\circ\!\!\to} \cdot {}^*\!\leftarrow t$ one can also simply drop the requirement that $s \rtimes t$ is an overlay in the second item.

▶ **Theorem 7.** *If a* TRS $\mathcal{R}$ *is left-linear and almost development closed then* $\mathrel{-\!\!\circ\!\!\to}$ *is strongly confluent.* ☑

Strong confluence of $\multimap_{\mathcal{R}}$ immediately yields confluence of the TRS $\mathcal{R}$ by Lemma 1.

▶ **Corollary 8.** *Left-linear, almost development closed* TRS*s are confluent.* ◀

4.1 Original Proof

In [9, 10] van Oostrom indicates that the induction part of the proof of Theorem 5 can be easily adapted for proving Theorem 7, only the base case becomes more difficult since the possibility of overlays needs to be taken into account here. To be precise, in [9] it is stated that the second part of the proof of Theorem 5 *"can be essentially followed, proving strong confluence instead of the diamond property ... and changing the measure defined above by not counting the function symbols in critical intersections for overlays."* And according to [10] *"The idea is not to take symbols taking part in overlays between the development steps into account, for the amount of interference. This changes nothing in the second (induction) part of the proof."* Hence a natural first step to formalizing Theorem 7 seems to be defining the new measure – let us call it $▲'$ – as follows:

▶ **Definition 9.** $▲'(A,B) = \{p \mid \ell(\mathsf{src}^\sharp(A)(p)) = \alpha^m,\ \ell(\mathsf{src}^\sharp(A)(p)) = \beta^n \text{ and } m \neq n \text{ for some } \alpha,\ \beta,\ m \text{ and } n\}$

As indicated in [9, 10] proving strong confluence of $\multimap$ should now proceed as in the proof of Theorem 5, where the inductive case should be easy using the new measure. However, as the following example shows, things are not that easy. The problem is that function positions, that were previously not counted because they were involved in overlays, might be counted after constructing $B[D'\rho]_{q_\beta}$.

▶ **Example 10.** The TRS consisting of the five rewrite rules

$$\begin{aligned} &\boldsymbol{\alpha}\colon\ \mathsf{f}(\mathsf{g}(x),\mathsf{a}) \to \mathsf{f}(x,\mathsf{a}) && \boldsymbol{\gamma}\colon\ \mathsf{g}(\mathsf{a}) \to \mathsf{b} && \boldsymbol{\epsilon}\colon\ \mathsf{f}(\mathsf{b},\mathsf{a}) \to \mathsf{f}(\mathsf{a},\mathsf{a}) \\ &\boldsymbol{\beta}\colon\ \mathsf{f}(\mathsf{g}(\mathsf{g}(y_1)),y_2) \to \mathsf{f}(\mathsf{g}(y_1),y_2) && \boldsymbol{\delta}\colon\ \mathsf{g}(\mathsf{b}) \to \mathsf{g}(\mathsf{a}) \end{aligned}$$

is left-linear and development closed[2] and hence also almost development closed.

Consider the proof terms $A = \boldsymbol{\alpha}(\boldsymbol{\gamma})$ and $B = \boldsymbol{\beta}(\mathsf{a},\mathsf{a})$. We have

$$\mathsf{src}^\sharp(A) = \mathsf{f}_{\boldsymbol{\alpha}^0}(\mathsf{g}_{\boldsymbol{\alpha}^1}(\mathsf{g}_{\boldsymbol{\gamma}^0}(\mathsf{a}_{\boldsymbol{\gamma}^1})),\mathsf{a}_{\boldsymbol{\beta}^1}) \qquad \text{and} \qquad \mathsf{src}^\sharp(B) = \mathsf{f}_{\boldsymbol{\beta}^0}(\mathsf{g}_{\boldsymbol{\beta}^1}(\mathsf{g}_{\boldsymbol{\beta}^2}(\mathsf{a})),\mathsf{a})$$

and hence $▲(A,B) = 3$. The function symbols at positions ϵ and 1 do not count in the new measure since they correspond to the overlay between rules $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$. So $▲'(A,B) = 1$. Now we pick the innermost overlap between $\boldsymbol{\gamma}$ in A and $\boldsymbol{\beta}$ in B. So $\Delta_1 = \mathsf{f}(\mathsf{g}(\boldsymbol{\gamma}),\mathsf{a})$ and $\Delta_2 = \boldsymbol{\beta}(\mathsf{a},\mathsf{a})$. The critical peak is $\mathsf{f}(\mathsf{g}(\mathsf{b}),y_2) \leftarrow \mathsf{f}(\mathsf{g}(\mathsf{g}(\mathsf{a})),y_2) \to \mathsf{f}(\mathsf{g}(\mathsf{a}),y_2)$. It can be closed by simply applying rule $\boldsymbol{\delta}$ at position 1 – as a proof term take $D' = \mathsf{f}(\boldsymbol{\delta},y_2)$. Since $\rho = \{y_1 \mapsto \mathsf{a}, y_2 \mapsto \mathsf{a}, x \mapsto \mathsf{g}(\mathsf{a})\}$ and $q_\beta = \epsilon$ we have $B[D'\rho]_{q_\beta} = D'\rho = \mathsf{f}(\boldsymbol{\delta},\mathsf{a})$ and $A \,/\, \Delta_1 = \boldsymbol{\alpha}(\mathsf{b})$, and hence

$$\begin{aligned} &\mathsf{src}^\sharp(B[D'\rho]_{q_\beta}) = \mathsf{f}(\mathsf{g}_{\boldsymbol{\delta}^0}(\mathsf{b}_{\boldsymbol{\delta}^1}),\mathsf{a}) && \mathsf{src}^\sharp(A \,/\, \Delta_1) = \mathsf{f}_{\boldsymbol{\alpha}^0}(\mathsf{g}_{\boldsymbol{\alpha}^1}(\mathsf{b}),\mathsf{a}_{\boldsymbol{\alpha}^1}) \\ &▲'(A \,/\, \Delta_1, B[D'\rho]_{q_\beta}) = ▲(A \,/\, \Delta_1, B[D'\rho]_{q_\beta}) = 1 \end{aligned}$$

Note that the measure $▲'$ did not decrease, showing that proving Theorem 7 in this way is impossible.

[2] This is easily verified using CSI [4] together with CeTA.

4.2 Formalized Proof

We found that keeping the measure ▲ and doing a simple case distinction in the inductive step suffices to show strong confluence of $\multimap$.

Proof of Theorem 7 (Adaptations). The proof requires only minimal changes to the proof of Theorem 5. We only highlight the differences here. Assume $t \mathrel{\multimap\!\!\leftarrow} s \multimap u$ and let A be a proof term representing $s \multimap t$ and let B be a proof term representing $s \multimap u$. We show $t \multimap v \mathrel{{}^*\!\leftarrow} u$ for some term v by well-founded induction on the amount of overlap between A and B.

- Base case: Just like in the proof of Theorem 5 we obtain the residuals $A \mathbin{/} B$ and $B \mathbin{/} A$. Since $\mathsf{tgt}(A \mathbin{/} B) = \mathsf{tgt}(B \mathbin{/} A)$ and $\multimap \subseteq \to^*$ this implies $t \multimap v \mathrel{{}^*\!\leftarrow} u$ for $v = \mathsf{tgt}(A \mathbin{/} B) = \mathsf{tgt}(B \mathbin{/} A)$.
- Step case: Items 1–3 of the proof of Theorem 5 remain almost exactly the same. Since strong confluence is an asymmetric condition, we cannot simply assume without loss of generality that $q \leqslant p$. However, the two cases $q < p$ and $p < q$ still work as in the proof of Theorem 5 by constructing a proof term for $\mathsf{tgt}(\Delta_1) \multimap \mathsf{tgt}(B \mathbin{/} \Delta_2)$ and $\mathsf{tgt}(\Delta_2) \multimap \mathsf{tgt}(A \mathbin{/} \Delta_1)$ respectively and showing that the measure decreases for the new steps. In both cases this allows us to apply the induction hypothesis and obtain steps $t \multimap v \mathrel{{}^*\!\leftarrow} u$. If $p = q$ then we have an overlay and the remainder of the proof changes as follows. A graphical representation of this case is displayed in Figure 1.
  4. By the almost development closedness assumption there exists a term v', a proof term D' witnessing $\mathsf{rhs}(\alpha)\tau \multimap v'$, and a rewrite sequence $\mathsf{rhs}(\beta)\tau \to^* v'$.
  5. We define the substitution ρ as in item 5 of the previous proof and show that $B[D'\rho]_{q_\beta}$ witnesses a multi-step $t' \multimap w$ for some term w.
  6. Again $▲(A \mathbin{/} \Delta_1, B[D'\rho]_{q_\beta}) < ▲(A, B)$ just like in the previous proof.
  7. We apply the induction hypothesis to obtain a term v, multi-step $t \multimap v$, and rewrite sequence $w \to^* v$.
  8. It remains to show that there exists a rewrite sequence $u \to^* w$. Since $u = \mathsf{tgt}(B) = \mathsf{tgt}(B \mathbin{/} \Delta_1)$ we know $u = \mathsf{tgt}(B[\mathsf{rhs}(\beta)\tau\rho]_{q_\beta})$ using properties of τ and ρ. Moreover since $w = \mathsf{tgt}(B[D'\rho]_{q_\beta})$ and $\mathsf{tgt}(D') = v'$ we know $w = \mathsf{tgt}(B[v'\rho]_{q_\beta})$ by an application of Lemma 2. From item 4 we know that there exists a rewrite sequence $\mathsf{rhs}(\beta)\tau \to^* v'$ so together with Lemma 3 we obtain the desired rewrite sequence $u \to^* w$. ◀

In the formalized proof we combined the cases $q = p$ and $q < p$ since in both cases steps 4–8 from the proof above can be applied, and dropping the additional case distinction saves a few lines of code.

4.3 Commutation

Theorem 7 can easily be extended to commutation. For proving commutation it is important to keep track of the underlying TRS for each proof term involved in the proof. In the formalization this is done via the predicate `wf_pterm` $\mathcal{R}$. The predicate checks whether all rule symbols belong to a certain TRS $\mathcal{R}$ (and whether the correct number of arguments is provided for each rule symbol). It is easy to see that whenever $A \in$ `wf_pterm` $\mathcal{R}_1$ and $B \in$ `wf_pterm` $\mathcal{R}_2$ and $A \mathbin{/} B$ is defined then $A \mathbin{/} B \in$ `wf_pterm` $\mathcal{R}_1$ ☑. Similar results hold for contexts and subsitutions of proof terms, e.g. if $A \in$ `wf_pterm` $\mathcal{R}_1$ and σ a substitution from variables to proof terms over $\mathcal{R}_1$ then $A\sigma \in$ `wf_pterm` $\mathcal{R}_1$.

▶ **Theorem 11.** *Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be left-linear* TRS*s. If*

1. *$s \multimap_{\mathcal{R}_2} \cdot \mathrel{{}^*_{\mathcal{R}_1}\!\leftarrow} t$ for all critical pairs $s \mathrel{{}_{\mathcal{R}_1}\!\leftarrow} \rtimes \to_{\mathcal{R}_2} t$, and*
2. *$s \multimap_{\mathcal{R}_1} t$ for all critical pairs $s \mathrel{{}_{\mathcal{R}_2}\!\leftarrow} \rtimes \to_{\mathcal{R}_1} t$ which are not overlays*

then $\to_{\mathcal{R}_1}$ and $\to_{\mathcal{R}_2}$ commute. ☑

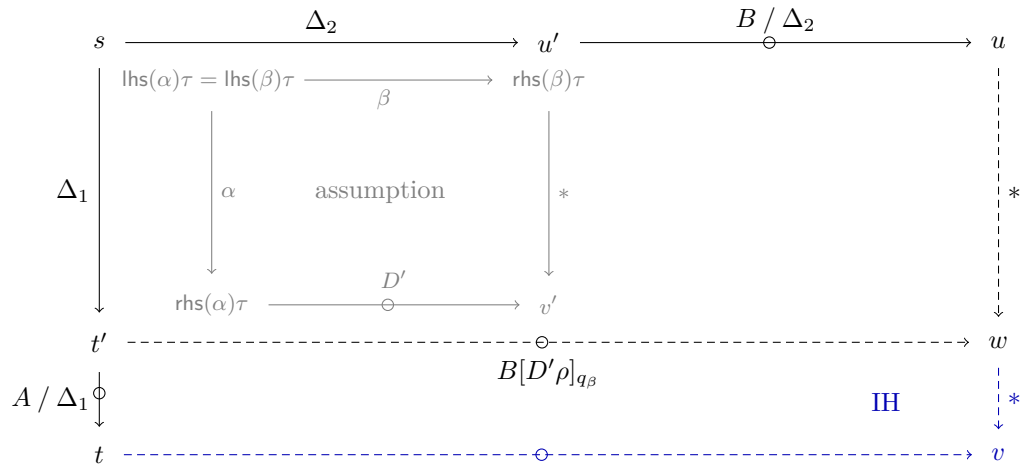


Figure 1 Overlay case in the proof of Theorem 7.

Proof (Adaptations). According to Lemma 1 it suffices to show strong commutation of $\multimap_{\mathcal{R}_1}$ and $\multimap_{\mathcal{R}_2}$. Assume $t \mathrel{\multimap\!\!\circ_{\mathcal{R}_1}} s \multimap_{\mathcal{R}_2} u$ and let $A \in$ `wf_pterm` $\mathcal{R}_1$ be a proof term representing $s \multimap_{\mathcal{R}_1} t$ and let $B \in$ `wf_pterm` $\mathcal{R}_2$ be a proof term representing $s \multimap_{\mathcal{R}_2} u$. We show $t \multimap_{\mathcal{R}_2} v \,{}^*_{\mathcal{R}_1}\!\!\leftarrow u$ for some term v by induction on $\blacktriangle(A, B)$. The base case now additionally requires that $A \,/\, B \in$ `wf_pterm` $\mathcal{R}_1$ and $B \,/\, A \in$ `wf_pterm` $\mathcal{R}_2$, which is easy to show as mentioned above. For the step case similar observations hold. In particular $B[D'\rho]_{q_\beta} \in$ `wf_pterm` $\mathcal{R}_2$ since by assumption $D' \in$ `wf_pterm` $\mathcal{R}_2$ and the substitution ρ maps to subterms of B which are also in `wf_pterm` $\mathcal{R}_2$. Hence all arrows pointing to the right in Figure 1 can be labeled with $\mathcal{R}_2$ and all arrows pointing down can be labeled with $\mathcal{R}_1$. Consequently, the proof of Theorem 7 can be followed again. ◀

5 Conclusion

We described extensions of our recent formalization of the development-closedness criterion to almost development closed critical pairs and commutation. During the process of formalizing these extensions we were able to simplify the formalization in [3] to the one presented here in Section 3. This version allowed for a straightforward adaptation to almost development closed critical pairs. The amount of Isabelle code before and after implementing the extension stayed roughly the same, since some previous results could be dropped while only one really new lemma (Lemma 3) had to be added in addition to the case distinction described in Section 4. Some more work was required to provide an executable "check"-function to integrate the result into CeTA.[3] Extending Theorem 7 to the commutation version (Theorem 11) was even more straightforward and required only minimal adaptations by providing more information about which proof term belongs to which of the two involved TRSs.

References

1 Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998. doi:10.1017/CBO9781139172752.

[3] $\sim$ 150 lines of Isabelle code for the check-function together with a proof that it corresponds to Theorem 7 ☑.

2 Christina Kohl and Aart Middeldorp. ProTeM: A proof term manipulator (system description). In Hélène Kirchner, editor, *Proc. 3rd International Conference on Formal Structures for Computation and Deduction*, volume 108 of *Leibniz International Proceedings in Informatics*, pages 31:1–31:8, 2018. doi:10.4230/LIPIcs.FSCD.2018.31.

3 Christina Kohl and Aart Middeldorp. A formalization of the development closedness criterion for left-linear term rewrite systems. In Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve Zdancewic, editors, *Proc. 12th International Conference on Certified Programs and Proofs*, pages 197–210, 2023. doi:10.1145/3573105.3575667.

4 Julian Nagele, Bertram Felgenhauer, and Aart Middeldorp. CSI: New evidence — a progress report. In Leonardo de Moura, editor, *Proc. 26th International Conference on Automated Deduction*, volume 10395 of *Lecture Notes in Artificial Intelligence*, pages 385–397, 2017. doi:10.1007/978-3-319-63046-5_24.

5 Julian Nagele and René Thiemann. Certification of confluence proofs using CeTA. In Takahito Aoto and Delia Kesner, editors, *Proc. 3rd International Workshop on Confluence*, pages 19–23, 2014. Available from http://cl-informatik.uibk.ac.at/iwc/iwc2014.pdf.

6 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.

7 TeReSe, editor. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

8 Yoshihito Toyama. Commutativity of term rewriting systems. In Kazuhiro Fuchi and Laurent Kott, editors, *Programming of Future Generation Computers II*, pages 393–407. North-Holland, 1988.

9 Vincent van Oostrom. Development closed critical pairs. In Gilles Dowek, Jan Heering, Karl Meinke, and Bernhard Möller, editors, *Proc. 2nd International Workshop on Higher-Order Algebra, Logic, and Term Rewriting*, volume 1074 of *Lecture Notes in Computer Science*, pages 185–200, 1995. doi:10.1007/3-540-61254-8_26.

10 Vincent van Oostrom. Developing developments. *Theoretical Computer Science*, 175(1):159–181, 1997. doi:10.1016/S0304-3975(96)00173-9.