10th International Conference on Interactive Theorem Proving

ITP 2019, September 9–12, 2019, Portland, OR, USA

Edited by

John Harrison

John O'Leary

Andrew Tolmach

Editors

John Harrison
Amazon AWS, Portland, OR, USA
jrh013@gmail.com

John O'Leary
Intel Corporation, Hillsboro, Oregon, USA
john.w.oleary@intel.com

Andrew Tolmach
Portland State University, Portland, OR, USA
tolmach@pdx.edu

ACM Classification 2012
Theory of computation → Logic

ISBN 978-3-95977-122-1

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/978-3-95977-122-1.

Publication date
September, 2019

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

Digital Object Identifier: 10.4230/LIPIcs.ITP.2019.0

LIPIcs – Leibniz International Proceedings in Informatics

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

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

10th International Conference on Interactive Theorem Proving (ITP 2019).
Editors: John Harrison, John O'Leary, and Andrew Tolmach
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Short Papers

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, in Portland OR, USA, is the tenth to be held under the ITP name, following Edinburgh 2010, Nijmegen 2011, Princeton 2012, Rennes 2013, Vienna 2014, Nanjing 2015, Nancy 2016, Brasilia 2017 and Oxford 2018; those in 2010, 2014 and 2018 were under the umbrella organization of the Federated Logic Conference (FLoC).

This year's conference attracted a total of 72 submissions (61 long papers and 11 short papers); with the exception of the very first ITP in 2010 (which received 74 submissions) this is the largest number of submissions received by ITP or its predecessor conferences. 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: 33 papers (29 long papers and 4 short). As a consequence of limited time for presentation at the conference, many interesting papers had to be rejected. 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 keynote talks by June Andronick (Data 61, CSIRO), Kevin Buzzard (Imperial College) and Martin Dixon (Intel).

The present volume collects all the accepted papers contributed to the conference as well as abstracts of the three invited presentations. This year, for the first time, we are publishing the proceedings in the LIPIcs series, motivated by its commitment to open access. We thank all those at Dagstuhl for their responsive feedback on all matters associated with the production of the finished proceedings.

Finally, we are grateful to Portland State University for logistical support, to several corporate donors who helped to support the conference, and to the ITP Steering Committee for their guidance throughout.

July 2019

John Harrison
John O'Leary
Andrew Tolmach

10th International Conference on Interactive Theorem Proving (ITP 2019).
Editors: John Harrison, John O'Leary, and Andrew Tolmach
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Program Committee

Andreas Abel	Gothenburg University, Sweden
David Aspinall	The University of Edinburgh, Scotland
Jeremy Avigad	Carnegie Mellon University, USA
Mauricio Ayala-Rincon	Universidade de Brasilia, Brasil
Yves Bertot	Inria, France
Sandrine Blazy	University of Rennes 1 – IRISA, France
Arthur Charguéraud	Inria, France
Koen Claessen	Chalmers University of Technology, Sweden
Gilles Dowek	Inria and ENS Paris-Saclay, France
Amy Felty	University of Ottawa, Canada
Jean-Christophe Filliatre	CNRS, France
Ruben Gamboa	University of Wyoming, USA
Shilpi Goel	Centaur Technology, Inc., USA
John Harrison	Amazon AWS, USA (co-chair)
Jean-Baptiste Jeannin	University of Michigan, USA
Cezary Kaliszyk	University of Innsbruck, Austria
Gerwin Klein	Data61, CSIRO and UNSW Sydney, Australia
Joe Leslie-Hurd	Intel, USA
Assia Mahboubi	Inria, France
Guillaume Melquiond	Inria, France
Leonardo de Moura	Microsoft, USA
Magnus Myreen	Chalmers University of Technology, Sweden
Tobias Nipkow	Technical University of Munich, Germany
John O'Leary	Intel, USA (co-chair)
Sam Owre	SRI, USA
Lawrence Paulson	University of Cambridge, UK
Christine Rizkallah	UNSW Sydney, Australia
Alexey Solovyev	Independent mobile software developer
Sofiene Tahar	Concordia University, Canada
Andrew Tolmach	Portland State University, USA (co-chair)
Christian Urban	King's College London, UK
Josef Urban	Czech Technical University in Prague, Czech Republic

10th International Conference on Interactive Theorem Proving (ITP 2019).
Editors: John Harrison, John O'Leary, and Andrew Tolmach
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

External Reviewers

Waqar Ahmad
Asad Ahmed
Idir Ait Sadoune
Ariane A. Almeida
Sidney Amani
Callum Bannister
Lasse Blaauwbroek
Chad Brown
Ali Bukhari
David Butler
Evelyne Contejean
Pierre-Evariste Dagand
Thaynara Arielly de Lima
Flavio L. C. De Moura
Larry Diehl
Christian Doczkal
Simon Doherty
Yannick Forster
Thibault Gauthier
Amjad Gawanmeh
Georges Gonthier
Ganesh Gopalakrishnan
Benjamin Gregoire
Maximilian Paul Louis Haslbeck
Ahmed Irfan
Guilhem Jabber
Jacques-Henri Jourdan
Manfred Kerber
Quentin Ladeveze
Peter Lammich
Xavier Leroy
Michael McInerney
Michael Norrish
Julian Parsert
Edward Pierzchalski
Nir Piterman
Johannes Åman Pohjola
Andrei Popescu
Thiago Mendonça Ferreira Ramos
Adnan Rashid
Thomas Sewell
Umair Siddique
Marielle Stoelinga
Rob Sumners
René Thiemann
Alwen Tiu
Aaron Tomb
Prathamesh Turaga
Vincent Van Oostrom
Marco Vassena

10th International Conference on Interactive Theorem Proving (ITP 2019).
Editors: John Harrison, John O'Leary, and Andrew Tolmach
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

A Million Lines of Proof About a Moving Target

June Andronick
CSIRO's Data61, Sydney, Australia
UNSW, Sydney, Australia
June.Andronick@data61.csiro.au

Abstract

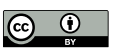
In the last ten years, we have been porting, maintaining, and evolving the world's largest proof base, the formal proof in Isabelle/HOL of the seL4 microkernel. But actually, there is no such thing as "the seL4 proof"; there are a number of proofs (functional correctness, binary translation validation, integrity and confidentiality proofs, etc) about a number of instances of seL4 (depending on the hardware platform it runs on, the features it includes, the extensions it supports). We will give an overview of the current state of these proofs, and, importantly, the challenges we face in keeping to maintain, evolve and extend them, and the processes we have put in place to manage their dependence on the evolving implementation.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Software and its engineering → Software evolution; Software and its engineering → Operating systems

Keywords and phrases Proof maintentance, proof evolution, seL4, Isabelle/HOL

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.1

Category Invited Talk



What Makes a Mathematician Tick?

Kevin Buzzard
Imperial College, London, U.K.
k.buzzard@imperial.ac.uk

Abstract

Formalised mathematics has a serious image problem in mathematics departments. Mathematicians working in "mainstream" areas such as modern algebra, analysis, geometry etc have absolutely no desire to work formally, it slows them down and they cannot see the point. The mathematical community has its own methods for deciding whether a proof (in pdf format) is correct or not; these methods rely on the views of a cabal of experts – our high priests. Our proof of the odd order theorem is "John Thompson got a Fields Medal for the work". This proof is of a rather different nature to the formalised proof of Gonthier et al. Our methods are arcane and mysterious; there is also ample evidence that they are, in general, extremely accurate when it comes to the important stuff.

I will talk about my attempts, as a "mainstream mathematician", to introduce formalised mathematics to my community.

2012 ACM Subject Classification Mathematics of computing

Keywords and phrases Formalization of mathematics

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.2

Category Invited Talk

10th International Conference on Interactive Theorem Proving (ITP 2019).
Editors: John Harrison, John O'Leary, and Andrew Tolmach; Article No. 2; pp. 2:1–2:1

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

An Increasing Need for Formality

Martin Dixon
Intel Corp., Hillsboro, Oregon, USA

Abstract

The talk will touch on a number of practical opportunities for formal modeling and methods that Intel sees in HW security research including: instruction sets; the proliferation of programmable agents within SoC's; and negative space testing.

2012 ACM Subject Classification Security and privacy → Security in hardware; Security and privacy → Formal methods and theory of security

Keywords and phrases Hardware security, formal modeling, instruction sets, SoC's, negative space testing

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.3

Category Invited Talk

10th International Conference on Interactive Theorem Proving (ITP 2019).
Editors: John Harrison, John O'Leary, and Andrew Tolmach; Article No. 3; pp. 3:1–3:1

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

A Verified Compositional Algorithm for AI Planning

Mohammad Abdulaziz
Technical University of Munich, Germany
mohammad.abdulaziz@in.tum.de

Charles Gretton
Australian National University, Canberra, Australia
charles.gretton@anu.edu.au

Michael Norrish
Data61, CSIRO, Canberra, Australia
Australian National University, Canberra, Australia
michael.norrish@data61.csiro.au

Abstract

We report on our HOL4 verification of an AI planning algorithm. The algorithm is compositional in the following sense: a planning problem is divided into multiple smaller abstractions, then each of the abstractions is solved, and finally the abstractions' solutions are composed into a solution for the given problem. Formalising the algorithm, which was already quite well understood, revealed nuances in its operation which could lead to computing buggy plans. The formalisation also revealed that the algorithm can be presented more generally, and can be applied to systems with infinite states and actions, instead of only finite ones.

Our formalisation extends an earlier model for slightly simpler transition systems, and demonstrates another step towards formal treatments of more and more of the algorithms and reasoning used in AI planning, as well as model checking.

2012 ACM Subject Classification Computing methodologies → Artificial intelligence; Computing methodologies → Planning for deterministic actions; Computing methodologies → Planning with abstraction and generalization; Software and its engineering → Software verification

Keywords and phrases AI Planning, Compositional Algorithms, Algorithm Verification, Transition Systems

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.4

Supplement Material All of our HOL4 scripts are available online at `https://doi.org/10.5281/zenodo.3298914`.

Funding *Mohammad Abdulaziz*: This author is supported by the DFG Koselleck Grant NI 491/16-1.

1 Introduction

State spaces of problems in fields such as artificial intelligence (AI) planning and model checking can be modelled as digraphs, where vertices and edges represent states and transitions, respectively. Explicitly representing such state spaces is infeasible in realistic systems. Instead, the digraph modelling the state space is described with a *propositionally factored* representation, using languages such as STRIPS by Fikes [17] or SMV by McMillan *et al.* [27]. We work in the space of tools and algorithms for solving problems represented in this way.

When working with such factored representations, controlling the *state space explosion* is critically important. A powerful, general approach to this problem is the *compositional* approach. Here, a solution to a problem instance is found, or approximated, by composing

solutions to one or many (possibly exponentially) smaller derived sub-problems, or "abstractions". Practically, this approach is one of the few known feasible approaches to solve problems concerning the state space of the given factored system. This is because it avoids constructing and performing computations on the large digraph modelling the state space, and only constructs and processes abstracted state spaces.

A planning problem is a *reachability* problem in a digraph representing the state space: given an initial state, is it possible to construct a sequence of actions that reaches a goal state? AI planning has many applications, including safety-critical ones, such as aerospace applications [34, 35]. Thus, it would be of great utility to use formal methods to increase the reliability of AI planning software, techniques, and frameworks. Indeed, this was realised by many early authors who used formal methods in AI planning applications [9]. However, all prior work was limited to using model checking techniques to formally verify planning domain model properties and plan properties, and none of the previous authors embarked on verifying a planning algorithm. In this paper we present the first formal verification of a planning algorithm. We use HOL4 [33] to formally verify the correctness of a compositional planning algorithm, which we published earlier [5], showing that the algorithm is indeed correct. One might wonder: why use a theorem prover to verify the algorithm, instead of a model checker like earlier applications of formal methods to planning? This is due to (i) the complexity of verifying a planning algorithm compared to verifying properties of planning models and plans as in earlier work, and (ii) the limitations of model checking formalisms, which are inadequate for representing the algorithm, let alone verifying it. Also, HOL4 has a transition systems theory library suitable to reasoning about planning algorithms [6].

The algorithm we verify works by dividing a planning problem into multiple isomorphic abstractions, solving each of those abstractions separately, and finally composing those solutions in a solution to the concrete problem. Each abstraction is an under-approximation of the problem that is isomorphic to a *descriptive quotient* (hereafter, quotient) of the problem. In our earlier work, this quotient was computed based on symmetries in the planning problem. This earlier work empirically established that this algorithm performs extremely well on benchmark planning problems which have symmetries.

As experienced practitioners might expect, formalisation in a theorem prover yields concrete benefits. In our case, we (i) gain a precise (and hitherto unappreciated) characterisation of what we required of the planning algorithm that solves the generated sub-problems; (ii) fix our algorithm to remove our dependency on that assumption; (iii) extend the algorithm's applicability to problems whose state variables are of arbitrary types, and not necessarily Boolean, thus showing its applicability to numerical and hybrid planning; and (iv) we prove its validity for a more general class of quotients, quotients which are not necessarily computed using problem symmetries.

Finally, we note that elements of our formalisation can be easily modified to accommodate the compositional model-checking algorithm by Ip and Dill [22, 23], which is used to perform model checking on systems with multiple isomorphic components in the Murphi verification system.

Contributions

Our paper makes the following contributions:

- We provide formal definitions of the notion of planning problems and develop a theory library concerning them (Section 2.2). This is a substantial extension of an existing HOL4 library on factored transition systems which was developed to verify algorithms to compute upper bounds on transition system state space diameters [1, 2, 3, 6].

- We formally define a state-of-the-art planning algorithm for planning, namely planning via descriptive quotients, that is used for efficiently solving planning problems with symmetries.
- We develop a significant theory to establish the correctness of the connection between the abstracted sub-problems and the original. In particular, we must answer two questions:
  - when and how can sub-plans that solve (abstracted) sub-problems be concatenated to solve the original concrete problem's goal?
  - how should a descriptive quotient solution be instantiated – i.e.,"lifted back" to the level of the original concrete problem – so as to create multiple plans for solving the symmetric sub-problems of the original?
- We believe our work is the first verification of a symmetry-breaking technique or a quotient-based technique for problems on transition systems. Our verification forced us to identify an important assumption about the behaviour of the planner used to solve the abstracted sub-problems.

2 Preliminaries

2.1 Standard HOL4 Types and Operations

HOL4 provides a rich library of operations over standard types such as lists and sets, giving a powerful combination of facilities from mathematics and functional programming. Here, we briefly describe those that we use below.

In the theory of lists: lists are either empty ("nil") written `[]`, or a head element h followed by the rest of the list t, written $h::t$. We write $l_1 \mathbin{+\!\!+} l_2$ to represent the concatenation of the lists l_1 and l_2. Lifting this to lists of lists, we write $\mathsf{FLAT}\ ll$ to mean the concatenation of all the lists contained within ll. We write $\mathsf{MEM}\ e\ l$ to mean that e is an element of list l. More generally, we can denote the set of all the elements contained in a list by writing $\mathsf{set}\ l$. Finally, we can write $\mathsf{MAP}\ f\ l$ to represent the pointwise application of function f to all elements of the list l, returning a list of equal length, but with elements possibly of a different type.

Most of the set notation we use should be familiar. Apart from set comprehensions and standard operators such as union and intersections, we also write $f(\!|x|\!)$ to mean the image of set x under function f, and f^{-1} for f's inverse (taking care to only use this when f is a bijection on the relevant sets).

We make extensive use of the HOL4 theory of finite maps, which are functions whose domains are finite. The domain of a map f is written $\mathcal{D}(f)$. Applying a map f to a domain element d is written $f\ `\ d$. We write $f \sqsubseteq g$ to mean that map f is a submap of g – i.e., f and g agree on all elements in $\mathcal{D}(f)$. Finally, we can combine two maps, writing $f \uplus g$. If the maps f and g have overlapping domains, the result takes elements in the overlap to f's values (the union "biases left").

Below, all statements appearing with a turnstile ($\vdash$) are HOL4 theorems, automatically pretty-printed to LaTeX, and using this notation.

2.2 Factored Transition Systems in HOL4

We now review basic concepts about propositionally factored representations of transition systems and how they are formalised in HOL4. The distinctive feature of these represenations is that sets of edges are compactly described in terms of "actions". This representation is equivalent to representations commonly used in the AI planning and model checking communities (e.g. STRIPS [17] and SMV [27, 13]).

▶ **Definition 1** (States and Actions). *A state, x, is a finite map from variables to values, i.e. a finite set of mappings $v \mapsto b$, where v is a variable and b is a value. An action is a pair of finite maps, (p, e), where p represents the* preconditions *and e represents the* effects*. The domain of an action is the union of the domains of its preconditions and effects, i.e. $\mathcal{D}(\pi) \equiv \mathcal{D}(p) \cup \mathcal{D}(e)$, for $\pi = (p, e)$. (Note how we are overloading/extending the syntax for the domain of a finite map ($\mathcal{D}(fm)$) to also mean the domain of an action ($\mathcal{D}(\pi)$), and (below) the domain of a system.)*

▶ **Definition 2** (Factored System). *A propositionally factored system, δ, is a set of actions. We write $\mathcal{D}(\delta)$ for the domain of δ, which is the union of the domains of all the actions in δ.*

To make the types explicit, a propositionally factored system in HOL4 has states as finite maps $\alpha \mapsto \beta$ (polymorphic in both domain (α) and codomain (β)).[1] *An action is then a pair of such states $(\alpha \mapsto \beta) \times (\alpha \mapsto \beta)$, and a factored transition system δ is a set of such actions.*

The valid states of a system δ, written $\mathbb{U}(\delta)$, are those that have the same domain as the system:

$$\mathbb{U}(\delta) \stackrel{\text{def}}{=} \{\, x \mid \mathcal{D}(x) = \mathcal{D}(\delta) \,\}$$

The valid plans of a system (δ^*) are those composed of actions drawn from δ:

$$\delta^* \stackrel{\text{def}}{=} \{\, \vec{\pi} \mid \mathsf{set}\ \vec{\pi} \subseteq \delta \,\}$$

▶ **Definition 3** (Execution). *When an action (p, e), denoted by π, is executed at state x, it produces a successor state $\mathsf{ex}(x, \pi)$, formally defined as $\mathsf{ex}(x, \pi) = \mathsf{if}\ p \subseteq x\ \mathsf{then}\ e \uplus x\ \mathsf{else}\ x$. We lift* ex *to lists of actions $\vec{\pi}$ as the second argument. So $\mathsf{ex}(x, \vec{\pi})$ denotes the state resulting from successively applying each action from $\vec{\pi}$ in turn, starting at x, which corresponds to a path in the state space. In HOL4 action execution and action sequence execution are defined as follows:*

$$\mathsf{state\text{-}succ}\ x\ (p,e) \stackrel{\text{def}}{=} \textbf{if}\ p \sqsubseteq x\ \textbf{then}\ e \uplus x\ \textbf{else}\ x$$

$$\mathsf{ex}(x, \pi :: \vec{\pi}) \stackrel{\text{def}}{=} \mathsf{ex}(\mathsf{state\text{-}succ}\ x\ \pi, \vec{\pi})$$
$$\mathsf{ex}(x, []) \stackrel{\text{def}}{=} x$$

The result of executing an action (p,e) on a state x depends on whether the preconditions of the action are satisfied by the state or not, which is modelled by the $p \sqsubseteq x$ relation. If the state satisfies the preconditions, then the state resulting from the execution is the same as the original state, but amended by the effects of the executed action. Otherwise, the result of the execution does not affect a change to the state. The finite map union operation, $e \uplus x$, models amending the state by the action effects e.

Our formal definition of action execution follows that from our earlier paper [5]. Having a total execution function (as above) is somewhat unusual for classical deterministic planning. The choice is more typical in robotics, and in settings where automated planning is undertaken under uncertainty. For example, the *de facto* standard in robotic planning is to plan in a partially observable Markov decision process [20, 10], in which the robot cannot generally know for sure if an action will have an effect or not. However, as machine learning becomes increasingly pervasive, both in the task of learning system models [8, 31], and in the task of computing plans [38], we can expect it to become increasingly common place for planned

[1] To model STRIPS or SMV transition systems, β would be instantiated with *bool*.

actions in classical deterministic settings to have no effect, since learning agents tradeoff model accuracy for model complexity. In addition to the totality of our definition of execution, we note that our definition is more general than other formalisms as it allows an action to execute in a state when the action is defined using symbols that are not part of that state. These properties of our definition made our proofs smoother, and helped us derive more general theorems.

A sanity check of our execution semantics is the following theorem, which states that the result of executing a valid action sequence on a valid state is also a valid state.

$$\vdash \overrightarrow{\pi} \in \delta^* \wedge x \in \mathbb{U}(\delta) \Rightarrow \mathsf{ex}(x, \overrightarrow{\pi}) \in \mathbb{U}(\delta)$$

When the codomain of a state is Boolean, we give examples of states and actions using sets of literals. For example, $\{v_1, \overline{v_2}\}$ is a state where state variables v_1 is (maps to) true, and v_2 is false and its domain is $\{v_1, v_2\}$. $(\{v_1, \overline{v_2}\}, \{v_3\})$ is an action that if executed in a state where v_1 and $\overline{v_2}$ hold, it sets v_3 to true. $\mathcal{D}((\{v_1, \overline{v_2}\}, \{v_3\})) = \{v_1, v_2, v_3\}$.

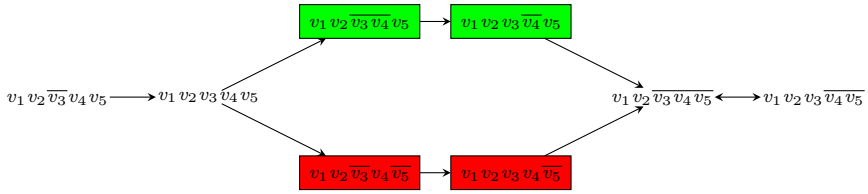


Figure 1 The largest connected component of the state space of the problem from Example 2. It shows the presence of symmetries between different states.

▶ **Example 1.** *An example factored system δ is $\{\pi_1, \pi_2, \pi_3\}$, where the actions π_1, π_2 and π_3 are defined as $(\emptyset, \{v_3\})$, $(\{v_1, v_3\}, \{\overline{v_3}, \overline{v_4}\})$, and $(\{v_2, v_3\}, \{\overline{v_3}, \overline{v_5}\})$, respectively. The largest connected component of its state space is shown in Figure 1.*

Note that, unlike in our original algorithm [5], the codomain of states is not restricted to be *bool* since a lot of the theory we develop here applies to factored systems regardless of the codomain of the state. Indeed, because we do not restrict the codomains to *bool* we are able to prove that the algorithm verified here can be used for planning problems with infinite states.

▶ **Definition 4** (Planning Problem). *A planning problem Π is a 3-tuple $\langle I, \delta, G \rangle$, with I the initial state of the problem, G a partial state representing a set of goal states, and δ a set of actions. We define the domain of the problem, $\mathcal{D}(\Pi)$, to be domain of its actions, $\mathcal{D}(\delta)$. The set of valid states, written $\mathbb{U}(\Pi)$, with respect to a planning problem Π, corresponds to the set $\mathbb{U}(\delta)$. In HOL4, we formalise this as a record type:*

```
(α, β) planningProblem = <|
    I : α ↦ β;
    δ : (α ↦ β) × (α ↦ β) → bool;
    G : α ↦ β
|>
```

Problem Π is valid *if the initial state is a valid state and the goal describes an assignment constraint on a subset of the problem's domain. In HOL4:*

$$\mathsf{valid\text{-}prob}\ \Pi \stackrel{\mathsf{def}}{=} \Pi.I \in \mathbb{U}(\Pi.\delta) \wedge \mathcal{D}(\Pi.G) \subseteq \mathcal{D}(\Pi)$$

Henceforth, we will work only with valid problems. We refer to the initial state, actions or goal of problem Π *as* $\Pi.I$*,* $\Pi.\delta$ *or* $\Pi.G$ *respectively. We may also omit the* Π *if it is clear from the context, e.g.* I *for* $\Pi.I$ *and* δ_i *for* $\Pi_i.\delta$*.*

Finally, an action sequence $\vec{\pi}$ *is a plan/solution for a planning problem* Π *iff that sequence is valid, and if all goal assignments are present in the state reached by executing that action sequence from the initial state:*

$$\Pi \text{ solved-by } \vec{\pi} \stackrel{\text{def}}{=} \vec{\pi} \in \Pi.\delta^* \wedge \Pi.G \sqsubseteq \mathsf{ex}(\Pi.I, \vec{\pi})$$

▶ **Example 2.** *An example planning problem is* Π_1 *with* $\Pi_1.I \equiv \{v_1, v_2, v_3, v_4, v_5\}$*,* $\Pi_1.G \equiv \{\overline{v_4}, \overline{v_5}\}$*, and actions* $\Pi_1.\delta$ *assigned to be* δ *from Example 1. The state space of that problem is that of the factored system* δ*, which represents its actions. A solution to that problem is the action sequence* $[\pi_1; \pi_2; \pi_1; \pi_3; \pi_1]$*.*

2.3 Motivating Planning via Descriptive Quotient

Better scalability is the core motivation for planning using a descriptive quotient. The algorithm treats the situation where a concrete problem can be decomposed into a set of isomorphic sub-problems. We need only find a solution for one sub-problem, and then it is a simple matter of instantiating that solution for each problem in the series to arrive at a solution for the concrete problem. These ideas can be made clear if we consider the GRIPPER problem, which happens to be a benchmark problem of the *International Planning Competition* [26]. A robot with left and right grippers must move a set of N indistinguishable packages from a common *source* location to a common *destination*. The left and right grippers are symmetric, because if we changed their names, by interchanging the terms "left" and "right" in the problem description, we are left with an identical problem. Packages are also interchangeable, and symmetric in this sense. The descriptive quotient here describes the problem of moving one package with one gripper to the destination, and then returning the robot to the source location with its gripper unencumbered. A plan for the quotient represents a solution for a part of the gripper problem, for one package. If we instantiate that quotient plan to move each package, and concatenate the instantiated plans, we arrive at a plan for the concrete problem. Some first package is moved to the goal, then a second, a third, and so on until all packages are in their goal location, and the problem is solved.

When in use, and compared to other planning algorithms, the algorithm we study here comes with some overhead. Specifically, it has four inputs, and only the first of which is common to all planning algorithms. These are: (i) a planning problem, (ii) an under-approximation of that problem, also known as the descriptive quotient, (iii) a plan for the descriptive quotient and (iv) a set of instantiations of the descriptive quotient. The last three objects are peculiar to the algorithm we investigate and are computed from the first input in a preprocessing step, based on symmetries in the given planning problem [5]. After this preprocessing step, a plan is calculated for the descriptive quotient problem which is usually much smaller than the concrete problem at hand. Then, the quotient's solution is instantiated to solve sub-problems of the given problem. Lastly, those sub-problem solutions are concatenated to form a solution for the entire problem.

The primary strength of planning via descriptive quotient is that the state space of a quotient is small relative to that of the concrete problem. This algorithm is thus relatively efficient at planning compared to an algorithm that searches for a plan in the state space of the concrete problem. However, is it effective compared to other methods that exploit symmetries for planning? The state-of-the-art method to break symmetries for planning is *orbit search* [30]. That method exploits the fact that a symmetry between state variables

induces a symmetry between states. Orbit search exploits that during the search for a plan since one only needs to visit one state out of every set of symmetric states. However, pruning the state space that way still gives rise to a search space that could be exponentially larger than the descriptive quotient's state space. For instance, the descriptive quotient of a GRIPPER problem with 20 packages is solved by breadth-first search expanding only 6 states [5]. On the other hand, a state-of-the-art system implementing orbit search reports expanding $60K$ states solving the same GRIPPER problem [30]. This difference is highlighted in the next example.

▶ **Example 3.** *In the problem Π_1 from our example earlier, state variables v_4 and v_5 are symmetric, i.e. Π_1 would stay the same if we permute them. Also v_1 and v_2 are symmetric. This variable symmetry induces symmetries between states as shown in Figure 1, where the two green states are symmetric with the red ones, i.e. permuting them does not change the state space. Ideally, the orbit search method would construct a state space where symmetric states are contracted as the one shown in Figure 3, which is clearly smaller in size than the original state space in Figure 1.*[2]

On the other hand, following our previously published algorithm, a descriptive quotient, Π'_1, of the problem Π_1 is computed by replacing every variable in Π with a symbol, where symmetric variables are replaced with the same symbol. Thus, Π'_1 has initial state, actions and goals that are $\{p_1, p_2, p_3\}$, $\{(\{p_1, p_2\}, \{\overline{p_2}, \overline{p_3}\}), (\emptyset, \{p_2\})\}$, and $\{\overline{p_3}\}$, respectively. The largest component of the state space of Π'_1 is shown in Figure 2. It is clearly smaller than the original state space shown in Figure 1, as well as the state space constructed by orbit search shown in Figure 3.

Figure 2 The largest connected component in the state space of the descriptive quotient of the problem in Example 2.

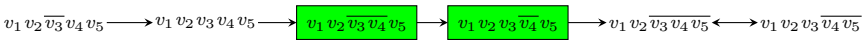


Figure 3 The state space which the orbit search algorithm could construct and in which it would search for a solution to the problem from Example 2. In the case that there were multiple symmetric states in the original problem, here only one *canonical* state from that set appears.

3 Sub-Plan Concatenation

The algorithm we describe and verify synthesises a concrete plan by concatenating a series of sub-plans. Each sub-plan solves one sub-problem of the concrete problem at hand. The first step of formally develop these ideas is describing sufficient conditions which enable one to synthesise a concrete plan according to a concatenation operation.

3.1 Needed Assignments

To concatenate plans safely, the algorithm needs to constrain states encountered between the execution of two concatenated plans to be compatible with the resources that might be used by a plan for the second problem. Compatibility is guaranteed if the intermediate

[2] For a comprehensive description of orbit search planning consult Pochter et al. [30].

state is consistent with the *needed assignments* of the second sub-problem. To understand this concept, suppose we have a plan $\vec{\pi}$ for a planning problem Π. What can we change in $\Pi.I$ and still guarantee that $\vec{\pi}$ solves the amended problem? Needed assignments are those assignments in $\Pi.I$ which cannot be changed.

▶ **Definition 5** (Needed Assignments). *Needed assignments, $\mathcal{N}(\Pi)$, are assignments in the preconditions of actions and goal conditions that also occur in I, i.e., $\mathcal{N}(\Pi) = (\mathsf{pre}(\delta) \cap I) \cup (G \cap I)$, where $\mathsf{pre}(\delta) \equiv \bigcup\{p \mid (p,e) \in \delta\}$. Formally, we begin by characterising a planning problem's needed variables, those state variables that shall be the subject of needed assignments:*

$$
\begin{array}{l}
\mathcal{D}(\mathcal{N}(\Pi)) \stackrel{\text{def}}{=} \\
\quad \{\, v \mid \\
\quad\; v \in \mathcal{D}(\Pi.I) \wedge \\
\quad\; (\exists\, p\ e.\ (p,e) \in \Pi.\delta \wedge v \in \mathcal{D}(p) \wedge p \text{ ` } v = \Pi.I \text{ ` } v) \vee \\
\quad\;\; v \in \mathcal{D}(\Pi.G) \wedge \Pi.I \text{ ` } v = \Pi.G \text{ ` } v)\,\}
\end{array}
$$

Then, the set of needed assignments associated with a problem Π is:

$$\mathcal{N}(\Pi) \stackrel{\text{def}}{=} \Pi.I\!\downarrow_{\mathcal{D}(\mathcal{N}(\Pi))}$$

where $x\!\downarrow_{vs}$ denotes the state x restricted/projected to assignments to variables vs.

▶ **Example 4.** *For Π_1 from our earlier example, we have that $\mathcal{N}(\Pi_1) = \{v_3, v_1, v_2\}$.*

We are then able to prove the following sanity-check theorem:

▶ **Proposition 1.** *For problem Π, a plan $\vec{\pi}$ will work from* any *state x that provides the needed assignments of that problem, even if x disagrees with the initial state of the problem on the assignments to some other – i.e.* not-needed *– state variables.*

$$
\begin{array}{l}
\vdash \mathsf{valid\text{-}prob}\ \Pi \wedge \mathcal{N}(\Pi) \sqsubseteq x \wedge \mathsf{sat\text{-}pre}\ (\mathcal{N}(\Pi), \vec{\pi}) \Rightarrow \\
\qquad \Pi\ \mathsf{solved\text{-}by}\ \vec{\pi} \Rightarrow \Pi.G \sqsubseteq \mathsf{ex}(x, \vec{\pi})
\end{array}
$$

The assumption $\mathsf{sat\text{-}pre}\ (\mathcal{N}(\Pi), \vec{\pi})$ says that if $\vec{\pi}$ is executed from $\mathcal{N}(\Pi)$, the preconditions of all actions in $\vec{\pi}$ shall be satisfied.

3.2 Concatenating Two Plans

Suppose we have a plan for each of two given problems. We now establish a core condition that, if satisfied, allows us to concatenate those plans to obtain an execution that satisfies the goal conditions of both problems. It may be that some state variables are common to both problems. We shall then require that for some total ordering of the problems, the preceding problem goal includes the needed assignments of the succeeding problem. Formally, we have the *preceding problem* relation:

▶ **Definition 6** (Preceding Problems).

$$\Pi_1 \lhd \Pi_2 \stackrel{\text{def}}{=} \Pi_1.G\!\downarrow_{\mathcal{D}(\mathcal{N}(\Pi_2))} = \mathcal{N}(\Pi_2)\!\downarrow_{\mathcal{D}(\Pi_1)} \wedge \Pi_1.G\!\downarrow_{\mathcal{D}(\Pi_2)} = \Pi_2.G\!\downarrow_{\mathcal{D}(\Pi_1)}$$

In words, (i) The needed assignments of Π_2 which a plan for Π_1 could possibly affect occur in G_1, and (ii) G_2 contains all the assignments in G_1 which a plan for Π_2 could affect.

▶ **Example 5.** *Consider a problem Π_2 s.t. $\Pi_2.I \equiv \{\overline{v_4}, \overline{v_5}, v_6\}$, $\Pi_2.\delta \equiv \{(\{\overline{v_5}\}, \{v_4, \overline{v_6}\}), (\{\overline{v_4}\}, \{v_5, \overline{v_6}\})\}$ and $\Pi_2.G \equiv \{\overline{v_4}, \overline{v_5}, \overline{v_6}\}$, respectively. $\mathcal{N}(\Pi_2) = G_1 = \{\overline{v_4}, \overline{v_5}\}$. Since $G_1\!\downarrow_{\mathcal{D}(\mathcal{N}(\Pi_2))} = G_1$, $(I_2\!\downarrow_{\mathcal{D}(\Pi_1)})\!\downarrow_{\mathcal{D}(\mathcal{N}(\Pi_2))} = G_1$, $G_1\!\downarrow_{\mathcal{D}(\Pi_2)} = G_1$ and $G_2\!\downarrow_{\mathcal{D}(\Pi_1)} = G_1$, we have $\Pi_1 \lhd \Pi_2$.*

Our precedence relation guarantees the following two properties.

▶ **Proposition 2.** *If a planning problem* Π_1 *precedes another planning problem* Π_2*, i.e.* $\Pi_1 \lhd \Pi_2$*, then a plan for* Π_1 *always preserves the needed assignments of* Π_2*.*

$$\vdash \Pi_1 \lhd \Pi_2 \wedge \Pi_1.G \sqsubseteq \mathsf{ex}(x,\vec{\pi}) \wedge \vec{\pi} \in \Pi_1.\delta^* \wedge \mathcal{N}(\Pi_2) \sqsubseteq x \wedge \mathsf{valid\text{-}prob}\ \Pi_1 \Rightarrow \mathcal{N}(\Pi_2) \sqsubseteq \mathsf{ex}(x,\vec{\pi})$$

▶ **Proposition 3.** *If* $\Pi_1 \lhd \Pi_2$*, then the a plan for* Π_2 *does not invalidate a goal of* Π_1*.*

$$\vdash \Pi_1 \lhd \Pi_2 \wedge \Pi_2.G \sqsubseteq \mathsf{ex}(x,\vec{\pi}) \wedge \vec{\pi} \in \Pi_2.\delta^* \wedge \mathsf{valid\text{-}prob}\ \Pi_2 \wedge \Pi_1.G \sqsubseteq x \Rightarrow \Pi_1.G \sqsubseteq \mathsf{ex}(x,\vec{\pi})$$

3.3 Concatenating Many Plans

The above analysis can be leveraged now to understand the situation where we have plans for many problems, and where a concatenation of those plans achieves and maintains goal conditions for all problems. We shall suppose that the set of problems are totally ordered according to our precedence relation.

▶ **Lemma 1.** *Consider a sequence* $\Pi_1 \ldots \Pi_N$ *satisfying* $\Pi_j \lhd \Pi_k$ *for all* $1 \leq j < k \leq N$*, and a state* x *that satisfies the initial state of every problem* Π_i*, for* $1 \leq i \leq N$*. For* $1 \leq i \leq N$ *let* $\vec{\pi}_i$ *be a plan for* Π_i *for which* $\mathsf{sat\text{-}pre}(\mathcal{N}(\Pi_i), \vec{\pi}_i)$ *holds. Then, not only is each* $\vec{\pi}_i$ *a plan for* Π_i*, but executing the entire concatenation* $\vec{\pi}_1 +\!\!+ \vec{\pi}_2 +\!\!+ \ldots \vec{\pi}_N$ *from* x also *satisfies the goals of each* Π_i*.*

$$\begin{array}{l}
\vdash \lhd_l\ \Pi_l \Rightarrow \\
\quad (\forall \Pi. \\
\qquad \mathsf{MEM}\ \Pi\ \Pi_l \Rightarrow \\
\qquad\quad \mathsf{valid\text{-}prob}\ \Pi \wedge \Pi.I \sqsubseteq x \wedge \Pi\ \mathsf{solved\text{-}by}\ \mathit{solve}\ \Pi \wedge \\
\qquad\quad \mathsf{sat\text{-}pre}\ (\mathcal{N}(\Pi), \mathit{solve}\ \Pi)) \Rightarrow \\
\quad\ (\textbf{let} \\
\qquad \mathit{sub_prob_plans} = \mathsf{MAP}\ \mathit{solve}\ \Pi_l\ ; \\
\qquad \mathit{concatenated_plans} = \mathsf{FLAT}\ \mathit{sub_prob_plans} \\
\quad\ \textbf{in} \\
\qquad \forall \Pi.\ \mathsf{MEM}\ \Pi\ \Pi_l \Rightarrow \Pi.G \sqsubseteq \mathsf{ex}(x, \mathit{concatenated_plans}))
\end{array}$$

In the HOL4 statement above (i) $\lhd_l$ is a predicate that lifts precedence to lists of problems, where for a list of problems $\Pi_1 \ldots \Pi_N$, it denotes that $\Pi_j \lhd \Pi_k$ holds, for all $j < k \leq N$, and (ii) *solve* is a function that maps every planning problem to a plan that solves it.

Before we discuss the proof of this lemma, we define the following union operation on planning problems and a lifted union operation for lists of planning problems.

▶ **Definition 7** (Planning Problem Union)**.**

$$\Pi_1 \cup \Pi_2 \stackrel{\text{def}}{=} \\ \quad \langle|I := \Pi_1.I \uplus \Pi_2.I;\ \delta := \Pi_1.\delta \cup \Pi_2.\delta;\ G := \Pi_1.G \uplus \Pi_2.G|\rangle$$

$$\bigcup \Pi_l \stackrel{\text{def}}{=} \mathsf{FOLDR} \cup \Pi_\emptyset\ \Pi_l$$

$\Pi_\emptyset$ is the "empty problem", whose initial and goal states have an empty domain, i.e. states mapping nothing to nothing, and that does not have actions.

The following theorem shows that the semantics of the planning problem union operations are as intended.

$$\vdash (\forall \Pi.\ \mathsf{MEM}\ \Pi\ \Pi_l \Rightarrow \mathsf{valid\text{-}prob}\ \Pi) \Rightarrow \mathsf{valid\text{-}prob}\ (\bigcup\ \Pi_l)$$

Informally, a sketch of the proof of Lemma 1 follows.

Proof. The proof is by induction on the list Π_l. The base case is trivial. In the step case we have the theorem for list of problems Π_l, and we need to show that it applies to Π_l with the problem Π pre-pended to it. The key idea of the proof is to deal with $\bigcup\ \Pi_l$ as one planning problem. Since Π precedes every problem in Π_l, we have that Π precedes $\bigcup\ \Pi_l$. From this, the inductive hypothesis, Proposition 1, Proposition 2, and Proposition 3, the result follows. ◀

Before this verification, we missed the condition $\mathsf{sat\text{-}pre}\ (\mathcal{N}(\Pi), f\ \Pi)$ from the assumptions of Lemma 1. This condition forbids plans with actions whose preconditions are unsatisfied during isolated execution in the corresponding problem – i.e. such actions are ignored by the execution function when considering the problem in isolation. The importance of this condition shall be discussed in detail in Section 6.

4 Covering via Concatenation

Having just developed conditions for plan synthesis via concatenation, it remains to understand how a concrete problem may be broken up into an ordered list of sub-problems, so that a concatenation of sub-problems plans corresponds to a plan for the concrete problem. First, this will require that we formally treat the question of what it is to be a sub-problem. We then establish a concept of coverage, so that when a concrete problem is *covered* by a list of sub-problems, we have the core sufficient condition to concatenate sub-problem plans according to a schema analogous to Lemma 1.

A problem is a sub-problem of another, if the constituents – states and actions – of the former are subsets/submaps of corresponding constituents of the latter.

▶ **Definition 8** (Sub-problem). *Problem Π_1 is a sub-problem of Π_2, written $\Pi_1 \subseteq \Pi_2$, if $I_1 \subseteq I_2$, and if $\delta_1 \subseteq \delta_2$.*

$$\Pi_1 \subseteq \Pi_2 \stackrel{\text{def}}{=} \Pi_1.I \sqsubseteq \Pi_2.I \wedge \Pi_1.\delta \subseteq \Pi_2.\delta$$

▶ **Definition 9** (Covering Problems). *A list of planing problems Π_l covers a problem Π iff (i) every member of Π_l is a sub-problem of Π and (ii) every goal of Π is a goal for some member of Π_l.*

$$\begin{aligned}&\mathsf{covers}\ \Pi_l\ \Pi \stackrel{\text{def}}{=}\\ &\quad(\forall x.\\ &\qquad x \in \mathcal{D}(\Pi.G) \Rightarrow\\ &\qquad\quad \exists \Pi'.\ \mathsf{MEM}\ \Pi'\ \Pi_l \wedge x \in \mathcal{D}(\Pi'.G) \wedge \Pi.G\ `\ x = \Pi'.G\ `\ x) \wedge\\ &\quad \forall \Pi'.\ \mathsf{MEM}\ \Pi'\ \Pi_l \Rightarrow \Pi' \subseteq \Pi\end{aligned}$$

▶ **Example 6.** *Let the problem Π_1'' be s.t. $\Pi_1''.I = \{v_3, v_1, v_4\}$, $\Pi_1''.\delta = \{(\{v_1, v_3\}, \{\overline{v_3},\ \overline{v_4}\}), (\emptyset, \{v_3\})\}$, and $\Pi_1''.G = \{\overline{v_4}\}$. Let the problem Π_1''' be s.t. $\Pi_1'''.I = \{v_3, v_2, v_5\}$, $\Pi_1'''.\delta = \{(\{v_2, v_3\}, \{\overline{v_3},\ \overline{v_5}\}), (\emptyset, \{v_3\})\}$, and $\Pi_1'''.G = \{\overline{v_5}\}$. The list $[\Pi_1''; \Pi_1''']$ covers the problem Π_1 since $\Pi_1'' \subseteq \Pi_1$ and $\Pi_1''' \subseteq \Pi_1$, and since Π_1'' covers the goal $\overline{v_4}$ in Π_1, Π_1''' covers the goal $\overline{v_5}$ in Π_1.*

We now establish sufficient conditions for the concatenation of sub-problem plans to solve the corresponding concrete problem. This result is a consequence of Lemma 1.

▶ **Theorem 1.** *Consider a set* $\Pi_1 \ldots \Pi_N$ *of problems that covers* Π*, satisfying* $\Pi_j \lhd \Pi_k$ *for all* $j < k \leq N$*. For* $1 \leq i \leq N$ *let* $\overrightarrow{\pi}_i$ *be a plan for* Π_i*. Then* $\mathsf{rem\text{-}cless}(\mathcal{N}(\Pi_1), \overrightarrow{\pi}_1) +\!\!+ \mathsf{rem\text{-}cless}(\mathcal{N}(\Pi_2), \overrightarrow{\pi}_2) +\!\!+ \ldots \mathsf{rem\text{-}cless}(\mathcal{N}(\Pi_N), \overrightarrow{\pi}_N)$ *is a plan for* Π*.*

$$
\begin{array}{l}
\vdash \mathsf{covers}\ \Pi_l\ \Pi \wedge \lhd_l\ \Pi_l \Rightarrow \\
\quad (\forall \Pi.\ \mathsf{MEM}\ \Pi\ \Pi_l \Rightarrow \mathsf{valid\text{-}prob}\ \Pi \wedge \Pi\ \mathsf{solved\text{-}by}\ f\ \Pi) \Rightarrow \\
\quad\ (\textbf{let} \\
\quad\quad \mathit{inst\_plans} = \mathsf{MAP}\ (\lambda \Pi'.\ \mathsf{rem\text{-}cless}\ (\mathcal{N}(\Pi'), [], f\ \Pi'))\ \Pi_l\ ; \\
\quad\quad \mathit{concatenated\_plans} = \mathsf{FLAT}\ \mathit{inst\_plans} \\
\quad\ \textbf{in} \\
\quad\quad \Pi\ \mathsf{solved\text{-}by}\ \mathit{concatenated\_plans})
\end{array}
$$

Note that in the theorem above, the sub-problem plans can be concatenated to solve the concrete problem after removing actions with unsatisfied preconditions. Such actions are removed by the function rem-cless. This is required to ensure that the assumption sat-pre satisfied, as is required for every sub-problem in Lemma 1. This function was not in our originally published algorithm, and is defined as follows:

$$
\begin{array}{l}
\mathsf{rem\text{-}cless}\ (x, pfx, (p,e) :: \overrightarrow{\pi}) \stackrel{\mathsf{def}}{=} \\
\quad \textbf{if}\ p \sqsubseteq \mathsf{ex}(x, pfx)\ \textbf{then}\ \mathsf{rem\text{-}cless}\ (x, pfx +\!\!+ [(p,e)], \overrightarrow{\pi}) \\
\quad \textbf{else}\ \mathsf{rem\text{-}cless}\ (x, pfx, \overrightarrow{\pi}) \\
\mathsf{rem\text{-}cless}\ (x, pfx, []) \stackrel{\mathsf{def}}{=} pfx
\end{array}
$$

The following two theorems show that rem-cless: (i) provides a list of actions whose preconditions are always satisfied during execution, and (ii) does not effect the results of execution in isolation in a sub-problem.

$$\vdash \mathsf{sat\text{-}pre}\ (x, \mathsf{rem\text{-}cless}\ (x, [], \overrightarrow{\pi}))$$

$$\vdash \mathsf{ex}(x, \overrightarrow{\pi}) = \mathsf{ex}(x, \mathsf{rem\text{-}cless}\ (x, [], \overrightarrow{\pi}))$$

During formalisation work related to Theorem 1, we discovered an error in our original conception of the definition of what a sub-problem is. Before this verification, we omitted the requirement that $\Pi_1 . I \sqsubseteq \Pi_2 . I$, opting for the erroneous condition $\mathcal{D}(\Pi_1) \subseteq \mathcal{D}(\Pi_2)$. This faulty definition allows for sub-problems of the same problem to have conflicting initial states, in which case the assumption of having more than one sub-problem becomes an insufficient assumption to prove the algorithm's soundness.

5 Concatenating Instantiations of a Quotient Plan

Theorem 1 establishes sufficient conditions enabling the synthesis of a concrete plan by concatenating plans for sub-problems. In fact, our compositional approach allows an additional efficiency: since it treats the scenario where each sub-problem is isomorphic, only one plan need ever be computed. That one plan is then *instantiated* for a covering set of isomorphic sub-problems. Finally, a concrete plan is synthesised by concatenating the instantiated sub-problem plans. This algorithm requires a canonical sub-problem, the quotient problem, which is isomorphic to each sub-problem of the concrete problem at hand. To permit sub-plan concatenation, successive sub-problems must satisfy the sub-problem precedence relation. To ensure this, our algorithm augments the quotient, ensuring that shared resources are left as they are found between sub-plan executions.

5.1 Formalising Instantiations

An instantiation maps constituents from a planning problem Π_1 to those from another problem Π_2, by mapping the state variables that underlie the mapped constituent. In particular, it is a function that explicitly maps the quotient into a sub-problem of the concrete problem by mapping: (i) quotient state variables to state variables of the concrete problem, (ii) quotient states to concrete problem states, and (iii) the quotient's actions to concrete problem actions.

To formulate that in HOL4, for state variables, instantiation is a function from $\mathcal{D}(\Pi_1)$ to $\mathcal{D}(\Pi_2)$. For states, it was a surprising challenge to define in HOL4 what an instantiation is. Because states are finite maps, the instantiation $\pitchfork\!\!\langle\!|x|\!\!\rangle$ of a state x is an application of an image of the instantiation $\pitchfork$ to the domain of the state. Instantiation here is therefore described as a function image application. For example, for a state $\{o_1 \mapsto T, o_2 \mapsto F\}$ and an instantiation function $\pitchfork$, the instantiation of that state using that function is the state $\{\pitchfork(o_1) \mapsto T, \pitchfork(o_2) \mapsto F\}$. This is equivalent to composing the inverse of the instantiation function with the state.

▶ **Definition 10** (State Instantiation). *Instantiation of state x with instantiation $\pitchfork$ is defined as the composition of x with the inverse of $\pitchfork$:*

$$\pitchfork\!\!\langle\!|x|\!\!\rangle \stackrel{\text{def}}{=} x \circ \pitchfork^{-1}$$

Overloading the $\langle\!|\,|\!\rangle$ notation, below we define the instantiation operation, for (i) an action, (ii) a factored system, (iii) a planning problem, and (iv) an action sequence, respectively.

$$\pitchfork\!\!\langle\!|(p,e)|\!\!\rangle \stackrel{\text{def}}{=} (\pitchfork\!\!\langle\!|p|\!\!\rangle, \pitchfork\!\!\langle\!|e|\!\!\rangle)$$

$$\pitchfork\!\!\langle\!|\delta|\!\!\rangle \stackrel{\text{def}}{=} (\lambda\,\pi.\ \pitchfork\!\!\langle\!|\pi|\!\!\rangle)\langle\!|\delta|\!\!\rangle$$

$$\pitchfork\!\!\langle\!|\Pi|\!\!\rangle \stackrel{\text{def}}{=} \Pi\ \texttt{with <|}\ I\ \texttt{:=}\ \pitchfork\!\!\langle\!|\Pi.I|\!\!\rangle\texttt{;}\ \delta\ \texttt{:=}\ \pitchfork\!\!\langle\!|\Pi.\delta|\!\!\rangle\texttt{;}\ G\ \texttt{:=}\ \pitchfork\!\!\langle\!|\Pi.G|\!\!\rangle\ \texttt{|>}$$

$$\pitchfork\!\!\langle\!|\vec{\pi}|\!\!\rangle \stackrel{\text{def}}{=} \mathsf{MAP}\ (\lambda\,\pi.\ \pitchfork\!\!\langle\!|\pi|\!\!\rangle)\ \vec{\pi}$$

▶ **Example 7.** *Recall from Example 3 the quotient Π'_1 of the concrete problem Π_1. Let instantiation $\pitchfork$ be $\{p_1 \mapsto v_1, p_2 \mapsto v_3, p_3 \mapsto v_4\}$. The problem Π''_1 from Example 6 is the same as $\pitchfork\!\!\langle\!|\Pi'_1|\!\!\rangle$, i.e. it is the instantiation of Π'_1 using $\pitchfork$.*

Let $\mathsf{valid\text{-}inst}\ \pitchfork$ mean that $\pitchfork$ is a bijection. We have the following theorems.

$$\vdash \mathsf{valid\text{-}inst}\ \pitchfork\ \Rightarrow\ \mathsf{ex}(\pitchfork\!\!\langle\!|x|\!\!\rangle, \pitchfork\!\!\langle\!|\vec{\pi}|\!\!\rangle) = \pitchfork\!\!\langle\!|\mathsf{ex}(x,\vec{\pi})|\!\!\rangle$$

$$\vdash \mathsf{valid\text{-}inst}\ \pitchfork\ \wedge\ \Pi\ \mathsf{solved\text{-}by}\ \vec{\pi}\ \Rightarrow\ \pitchfork\!\!\langle\!|\Pi|\!\!\rangle\ \mathsf{solved\text{-}by}\ \pitchfork\!\!\langle\!|\vec{\pi}|\!\!\rangle$$

$$\vdash \mathsf{valid\text{-}inst}\ \pitchfork\ \Rightarrow\ \mathcal{D}(\mathcal{N}(\pitchfork\!\!\langle\!|\Pi|\!\!\rangle)) = \pitchfork\!\!\langle\!|\mathcal{D}(\mathcal{N}(\Pi))|\!\!\rangle$$

Note that, in our original treatment [5], we did not explicitly state bijectivity of instantiations as a condition. This is because it is a consequence of the fact that the instantiations we considered then were transversals of equivalence classes of state variables under symmetry – a.k.a. *orbits*. Orbits form a partition of the domain of a planning problem, and since a transversal maps every orbit to one of its members, transversals are bijective.

Our algorithm also requires the following additional condition on sets of instantiations:

▶ **Definition 11** (Valid Set of Instantiations). *Any two different instantiations from a set of instantiations Δ should not map different variables from the domain of the quotient to the same variable in their range.*

$$\begin{aligned}&\mathsf{pwise\text{-}valid}\ \Delta\ vs \stackrel{\mathsf{def}}{=}\\ &\quad \forall \pitchfork_1\ \pitchfork_2\ v_1\ v_2 .\\ &\qquad \pitchfork_1 \in \Delta \wedge \pitchfork_2 \in \Delta \wedge v_1 \in vs \wedge v_2 \in vs \wedge v_1 \neq v_2 \Rightarrow\\ &\qquad\quad \pitchfork_1\ v_1 \neq \pitchfork_2\ v_2\end{aligned}$$

This condition guarantees that different instantiations of the same state are consistent with each other – i.e., a distinct variable is mapped to the same value in all the instantiated states. Again, we did not state this assumption explicitly in our original treatment since it holds for instantiations that are transversals of the state variable orbits, because a set of orbits forms a partition of the domain of the planning problem.

5.2 Planning via an Augmented Quotient

We now establish the correctness of the the target algorithm, which synthesises a concrete plan by concatenating a set of covering instantiations of the solution to an augmented quotient. The algorithm inputs are (i) a quotient of the concrete problem, (ii) a solution to that quotient, and (iii) a set of instantiations of the quotient which cover the concrete problem. In order to leverage the previous results in formally verifying that the target algorithm is correct, the key remaining task is to deal with that, as yet, we have no ordering of instantiations of isomorphic sub-problems. Theorem 1 is not directly applicable without a notion of precedence. We shall establish below the conditions so that any two instantiations of the augmented quotient can participate in the precedence relation together. Therefore, any ordering of such instantiations is admissible for the purposes of leveraging the algorithm verified in Theorem 1.

Two necessary concepts to state conditions guaranteeing that instantiations can participate in the precedence relation are *common variables* and *sustainable variables*. Given a set of instantiations, the common variables are variables mapped to the same value by at least two instantiations. A sustainable variable holds the same assignment in the initial state as it does in a goal state. Formally, they are defined as follows:

▶ **Definition 12** (Common Variables). *For a set of instantiations* Δ*, the set of* common variables*, written* $\bigcap_v \Delta$ *vs, comprises all elements from the given set of variables vs that occur in the ranges of more-than-one member of* Δ*.*

$$\begin{aligned}&\textstyle\bigcap_v\ \Delta\ vs \stackrel{\mathsf{def}}{=}\\ &\quad \{ v \mid \exists \pitchfork_1\ \pitchfork_2 .\ \pitchfork_1 \in \Delta \wedge \pitchfork_2 \in \Delta \wedge \pitchfork_1 \neq \pitchfork_2 \wedge v \in vs \wedge \pitchfork_1\ v = \pitchfork_2\ v \}\end{aligned}$$

▶ **Example 8.** *Let instantiation* $\pitchfork'$ *be* $\{p_1 \mapsto v_2, p_2 \mapsto v_3, p_3 \mapsto v_5\}$*. Take* Δ *to be* $\{\pitchfork, \pitchfork'\}$*. For* Δ*, we have* $\bigcap_v \Delta\ \{p_1, p_2, p_3\} = \{p_1\}$*.*

▶ **Definition 13** (Sustainable Variables). *A set of variables vs is sustainable in a problem* Π *iff* $I\downarrow_{vs} = G\downarrow_{vs}$*.*

$$\mathsf{sustainable}\ \Pi\ vs \stackrel{\mathsf{def}}{=} \Pi.I\downarrow_{vs}\ =\ \Pi.G\downarrow_{vs}$$

Our headline result relies on the following argument. If the intersection of – the needed variables of the quotient problem, with the common variables from instantiations Δ – are sustained in Π, then any pair of distinct instantiations of Π are elements in the precedence relation.

$$\begin{aligned}\vdash\ &\mathsf{valid\text{-}inst}\ \pitchfork_1 \wedge \mathsf{valid\text{-}inst}\ \pitchfork_2 \wedge\\ &\mathsf{pwise\text{-}valid}\ \Delta\ \mathcal{D}(\Pi) \wedge \mathsf{valid\text{-}prob}\ \Pi \wedge \pitchfork_1 \in \Delta \wedge\\ &\pitchfork_2 \in \Delta \wedge \pitchfork_1 \neq \pitchfork_2 \wedge \mathsf{sustainable}\ \Pi\ (\textstyle\bigcap_v\ \Delta\ \mathcal{D}(\Pi) \cap \mathcal{D}(\mathcal{N}(\Pi))) \Rightarrow\\ &\quad \pitchfork_1(\!|\Pi|\!) \lhd \pitchfork_2(\!|\Pi|\!)\end{aligned}$$

From this and from Theorem 1 we derive our headline theorem, stating soundness conditions for planning via a quotient. In this theorem we refer to a planning problem, Π', as being a descriptive quotient of some other problem Π. The stated conditions of the theorem define how some problem Π' qualifies as a descriptive quotient of Π.

▶ **Theorem 2.** *Consider a problem Π, a descriptive quotient Π', a solution $\overrightarrow{\pi}'$ to Π', and a set of instantiations Δ. Suppose $\{\pitchfork(\!|\Pi'|\!) \mid \pitchfork \in \Delta\} (= \mathbf{\Pi})$ covers Π, and $\bigcap_v \Delta\ \mathcal{D}(\Pi'_1) \cap \mathcal{D}(\mathcal{N}(\Pi'_1))$ are sustainable in Π'. Then any concatenation of the plans $\{\mathsf{rem\text{-}class}(\mathcal{N}(\pitchfork(\!|\Pi'|\!)), [], \pitchfork(\!|\overrightarrow{\pi}'|\!)) \mid \pitchfork \in \Delta\}$ solves Π.*

Note that the theorem above requires, for a quotient, that the intersection of its needed variables with the common variables between instantiations are sustainable. If this requirement is not satisfied by a quotient, the concatenated quotient plan instantiations might not solve the concrete problem, as shown below.

▶ **Example 9.** *Note that $\mathcal{D}(\mathcal{N}(\Pi'_1)) = \{p_1, p_2\}$, and recall that $\bigcap_v \Delta\ \{p_1, p_2, p_3\} = \{p_1\}$. Thus the intersection of the quotient's needed variables with the common variables between instantiations, $\bigcap_v \Delta\ \mathcal{D}(\Pi') \cap \mathcal{D}(\mathcal{N}(\Pi'_1))$, is $\{p_1\}$. The quotient Π'_1 does not sustain that intersection since the assignment of p_1 in $\Pi'_1.I$ does not occur in the goal $\Pi'_1.G$. Now, to see the problem this might cause, we instantiate the descriptive quotient Π'_1 with $\pitchfork'$, which yields problem Π'''_1 from Example 6. Thus $[\pitchfork(\!|\Pi'_1|\!); \pitchfork'(\!|\Pi'_1|\!)]$ covers the problem Π_1. A plan for the descriptive quotient Π'_1 is $\overrightarrow{\pi}' \equiv [(\{p_1, p_2\}, \{\overline{p}_3, \overline{p}_1\})]$ and its two instantiations are $\pitchfork(\!|\overrightarrow{\pi}'|\!) = [\pi_2]$ and $\pitchfork'(\!|\overrightarrow{\pi}'|\!) = [\pi_3]$. However, the two possible concatenations of $\pitchfork(\!|\overrightarrow{\pi}'|\!)$ and $\pitchfork'(\!|\overrightarrow{\pi}'|\!)$ do not solve Π_1 because both plans, $\pitchfork(\!|\overrightarrow{\pi}'|\!)$ and $\pitchfork'(\!|\overrightarrow{\pi}'|\!)$, require v_3 initially, but do not establish it.*

To guarantee that the intersection of the quotient's needed variables with the instantiations' common variables are sustainable, the goal of a quotient Π' is augmented with assignments that guarantee that the quotient sustains those variables. In particular, the quotient's goal should be augmented by the assignment of the variables $\bigcap_v \Delta\ \mathcal{D}(\Pi') \cap \mathcal{D}(\mathcal{N}(\Pi'_1))$ in the quotient's initial state. This step should be performed before the quotient is solved. The next example gives a concrete example of this augmentation.

▶ **Example 10.** *To solve Π_1 via solving Π'_1, we augment the goal $\Pi'_1.G$ with the initial state assignment of the variables in $\bigcap_v \Delta\ \{p_1, p_2, p_3\} \cap \mathcal{D}(\mathcal{N}(\Pi'_1))$, i.e. $\Pi'_1.I{\downarrow}_{\bigcap_v \Delta\ \{p_1,p_2,p_3\} \cap \mathcal{D}(\mathcal{N}(\Pi'_1))}$. The resulting problem, Π^q_1, is equal to Π'_1 except that it has the literal $\{p_1\}$ added to its goals, so $\Pi^q_1.G = \{p_1, \overline{p}_3\}$. A plan for Π^q_1 is $\overrightarrow{\pi}^q \equiv [(\{p_1, p_2\}, \{\overline{p}_3, \overline{p}_1\}); (\emptyset, \{p_1\})]$, and two instantiations of it are $\pitchfork(\!|\overrightarrow{\pi}^q|\!) = [\pi_2; \pi_1]$ and $\pitchfork'(\!|\overrightarrow{\pi}^q|\!) = [\pi_3; \pi_1]$. Concatenating $\pitchfork(\!|\overrightarrow{\pi}^q|\!)$ and $\pitchfork'(\!|\overrightarrow{\pi}^q|\!)$ in any order solves Π_1.*

The fact that goal augmentation works is shown in the following theorem.

$\vdash$ **let**
$\quad \Pi^q$ `=` Π' `with` G `:=` $\Pi'.I{\downarrow}_{\bigcap_v\ (\mathsf{set}\ \Delta)\ \mathcal{D}(\Pi')\ \cap\ \mathcal{D}(\mathcal{N}(\Pi'))}\ \uplus\ \Pi'.G$
in
$\quad \mathsf{sustainable}\ \Pi^q\ (\bigcap_v\ (\mathsf{set}\ \Delta)\ \mathcal{D}(\Pi^q)\ \cap\ \mathcal{D}(\mathcal{N}(\Pi^q)))$

Our headline result now follows straightforwardly from the manner in which the quotient augmentation operates. Because a quotient with an augmented goal sustains the common needed variables, the algorithm from Theorem 2 can be used to synthesise a concrete problem solution by instantiating and concatenating the augmented quotient solutions.

```
⊢ let
    Π^q = Π' with G := Π'.I |_{⋂_v (set Δ) D(Π') ∩ D(N(Π'))} ⊎ Π'.G ;
    inst_plans = MAP (λ⫛. rem-class (N(⫛(⟦Π^q⟧)), [], ⫛(⟦π⃗^q⟧))) Δ ;
    concatenated_plans = FLAT inst_plans
  in
    ALL-DISTINCT Δ ∧ (∀⫛. MEM ⫛ Δ ⇒ valid-inst ⫛) ∧
    valid-prob Π' ∧
    INJ (λ⫛. ⫛(⟦Π^q⟧)) (set Δ) U(:(α, β) planningProblem) ∧
    pwise-valid (set Δ) D(Π^q) ∧ covers (MAP (λ⫛. ⫛(⟦Π^q⟧)) Δ) Π ∧
    Π^q solved-by π⃗^q ⇒
      Π solved-by concatenated_plans
```

In closing, it is worth noting that Theorem 2 assumes nothing in the way the augmented quotient is computed. We believe this in itself is an important extension to our earlier work [5], which was limited to situations where the quotient under consideration is computed according to identified symmetric variables – i.e. as per Π' in Example 3. Our new results describe an algorithm that is applicable to descriptive quotients computed in problems that may not have symmetries. It is applicable provided the descriptive quotient is isomorphic to a set of sub-problems covering the concrete problem.

Also, the planning problem is of type (α, β) *planningProblem*. This denotes that indeed the algorithm for composing solutions is applicable to planning problems whose state variables can be assigned to values of any type β, without any constraints on that type. Additionally the cardinality of the set of actions in the problem or the quotient is unconstrained. Thus the planning problem and its quotient are not necessarily propositionally factored systems, making planning via descriptive quotients applicable to planning problems with infinite states, like numeric planning.

Lastly we note that the new algorithm, which includes a call to the function rem-class, suffers almost no run-time penalty compared to the original algorithm which did not include a call to rem-class. This is because the run-time of rem-class is linear in the length of the quotient plan, whose length in most benchmarks is linear in the problem size. Indeed, the overall run-time is dominated by finding a plan for the quotient.

6 Fixing the Algorithm via Formalisation

One benefit of our work is the discovery and correction of an easy-to-miss bug in our original algorithm [5]. We now describe that bug, first intuitively and then using a detailed example. Suppose a plan is found for a quotient system, and that plan contains a *spurious* action: an action whose precondition is not satisfied when that action is scheduled to execute – i.e., we have not applied the rem-class function. Now consider the case that the plan is instantiated multiple times, and the results of this are concatenated together to form a concrete plan. When we execute the first instantiation of the quotient's plan, no error occurs. However, that execution may have a "side effect", so that later instantiations of the spurious action now have an effect. It can be the case that such a spurious effect interferes with the plan execution, rendering the concrete plan invalid as follows.

For notational economy, let action schemata π_1, π_2, π_3, and π_4 be defined as $\pi_1(x, y, z) \equiv (\{x, y\}, \{\overline{y}, \overline{z}\})$, $\pi_2(x) \equiv (\emptyset, \{x\})$, $\pi_3(x) \equiv (\emptyset, \{\overline{x}\})$, and $\pi_4(x, y) \equiv (\{\overline{x}\}, \{y\})$, respectively. Consider a planning problem Π where $\Pi.I \equiv \{v_1, v_2, v_3, v_4, v_5, v_6, \overline{v_7}\}$, $\Pi.\delta \equiv \{\pi_1(v_1, v_3, v_4)$, $\pi_1(v_2, v_3, v_5), \pi_2(v_3), \pi_4(v_6, v_7), \pi_3(v_6)\}$, and $\Pi.G \equiv \{\overline{v_4}, \overline{v_5}, \overline{v_6}, \overline{v_7}\}$. Also consider Π', a quotient of Π, where $\Pi'.I \equiv \{p_1, p_2, p_3, p_4, \overline{p_5}\}$, $\Pi'.\delta \equiv \{\pi_1(p_1, p_2, p_3), \pi_2(p_2), \pi_4(p_4, p_5), \pi_3(p_4)\}$,

and $\Pi'.G \equiv \{\overline{p_3}, \overline{p_4}, \overline{p_5}\}$. Consider the two instantiations $\Cap$ and $\Cap'$ defined as $\Cap \equiv \{p_1 \mapsto v_1, p_2 \mapsto v_3, p_3 \mapsto v_4, p_4 \mapsto v_6, p_5 \mapsto v_7\}$, and $\Cap' \equiv \{p_1 \mapsto v_2, p_2 \mapsto v_3, p_3 \mapsto v_5, p_4 \mapsto v_6, p_5 \mapsto v_7\}$. Let the set of instantiations Δ be $\{\Cap, \Cap'\}$. The problem Π is covered by $\Cap(\!|\Pi'|\!)$ and $\Cap'(\!|\Pi'|\!)$, since they are sub-problems of Π and they cover its goal $\Pi.G$. The first step of the algorithm would be to augment the quotient's goal with $\Pi'.I\downarrow_{\bigcap_v \Delta\ \mathcal{D}(\Pi') \cap \mathcal{D}(\mathcal{N}(\Pi'))}$. We have $\bigcap_v \Delta\ \mathcal{D}(\Pi') = \{p_1, p_4, p_5\}$, i.e. there are three common variables between $\Cap$ and $\Cap'$. Also, the needed variables of the quotient are $\mathcal{D}(\mathcal{N}(\Pi')) = \{p_1, p_2\}$, since both p_1 and p_2 occur with the same assignments in the quotient's action preconditions and its initial state. Thus the goal of Π' is augmented with the literal $\{p_1\}$ resulting in the problem Π^q which is the same as Π' except that its goal $\Pi^q.G$ is $\{p_1, \overline{p_3}, \overline{p_4}, \overline{p_5}\}$. Next, the algorithm searches for a plan for Π^q. One such plan is $\vec{\pi}^q \equiv [\pi_1(p_1, p_2, p_3); \pi_2(p_1); \pi_4(p_4, p_5); \pi_3(p_4)]$. Note: when $\vec{\pi}^q$ is executed at the initial state $\Pi^q.I$, the action $\pi_4(p_4, p_5)$ will have no effect since its precondition, $\overline{p_4}$, will not hold before when it executes.

The next step is computing instantiations of $\vec{\pi}^q$, which are $\Cap(\!|\vec{\pi}^q|\!) = [\pi_1(v_1, v_3, v_4); \pi_2(v_3); \pi_4(v_6, v_7); \pi_3(v_6)]$ and $\Cap'(\!|\vec{\pi}^q|\!) = [\pi_1(v_2, v_3, v_5); \pi_2(v_3); \pi_4(v_6, v_7); \pi_3(v_6)]$. Then the algorithm returns the concatenation of $\Cap(\!|\vec{\pi}^q|\!)$ and $\Cap'(\!|\vec{\pi}^q|\!)$ in any order as a solution to Π. However, any concatenation of $\Cap(\!|\vec{\pi}^q|\!)$ and $\Cap'(\!|\vec{\pi}^q|\!)$ does *not* solve Π since the last occurrence of $\pi_4(v_6, v_7)$ in the concatenation will execute successfully. This is because the first occurrence of $\pi_3(v_6)$ sets the precondition of $\pi_4(v_6, v_7)$, and the execution of $\pi_4(v_6, v_7)$ will set v_7 to true, which contradicts the goal of Π.

The verified algorithm, however, returns a concatenation of the action sequences rem-class $(\mathcal{N}(\Cap(\!|\Pi^q|\!)), [], \Cap(\!|\vec{\pi}^q|\!))$ and rem-class $(\mathcal{N}(\Cap'(\!|\Pi^q|\!)), [], \Cap'(\!|\vec{\pi}^q|\!))$. This is a solution for Π since rem-class removes $\pi_4(v_6, v_7)$ from both $\Cap(\!|\vec{\pi}^q|\!)$ and $\Cap'(\!|\vec{\pi}^q|\!)$ since its preconditions are not met.

Interestingly, the possible bad scenario never showed up in any of the thousands of standard planning benchmarks on which we conducted our earlier experiments. We were lucky that the planner we used never produced plans with spurious actions. Nonetheless, we cannot afford to leave possible bugs latent in such corner cases if AI algorithms are to be deployed in a safety sensitive applications. Needless to say, our discovery of this bug further strengthens the argument for using formal verification for AI algorithms.

7 Related Work

The compositional approach to AI planning is very effective. A prominent example is planning using abstractions based on projection, exploiting acyclicity in variable dependencies [25, 39]. Also *factored planning* abstracts a problem into multiple "factors", which are obtained using a tree decomposition of a graph representation of variable dependencies [7, 11, 24].

Despite that extensive literature, to our knowledge, this is the first verification of a compositional planning algorithm. Indeed, most applications of formal methods to the area of AI planning were in the context of reasoning about planning domain models and plans and verifying properties of them, and not verifying planning algorithms. For instance, model checkers were used to validate that classical planning domain models satisfy given specifications [29, 21, 36]. Also, model checkers were used to verify safety and temporal properties of plans [19, 18]. Similar applications of model checking also exist for other planning formalisms, such as temporal planning [9]. Since the only formal technique used by earlier work was model checking, the limitation to only verifying model and plan properties, versus verifying planning algorithms, should come as no surprise. This is due to the limitations on what can be represented in model checkers and their formalisms.

The only application of theorem provers based formal methods to planning was by Abdulaziz and Lammich [4], who developed and formally verified a tool to validate planning domain models and plans. Other verification work using theorem provers relevant to our work is the verification of model checking algorithms, where the motivation is to obtain formally verified model checking algorithms and implementations [37, 32, 16, 12]. However, we note that although those model checking algorithms use abstractions based techniques, like partial order reduction, they are not compositional algorithms. Other related work is on formalising automata theory. For instance, textbook results in automata theory have been formalised on a number of occasions and in different logics [14, 15, 28, 40].

8 Conclusion

We verified a compositional AI planning algorithm that we published earlier, and found mistakes in its pen-and-paper formulation. This is similar to our earlier experience [1, 2, 3, 6], when we found mistakes in our and other people's work. We believe that planning may be particularly prone to such errors due to its heavy combinatorial nature, making it easy to miss corner cases, as well as the dense usage of notation in the planning literature. Although such errors can be corner cases, they cannot be tolerated in safety-critical applications such as outer space exploration, making a strong case for the utility of mechanical verification.

Furthermore, formalising the algorithm in a theorem prover made it easier to generalise our algorithm from planning problems with propositional state variables to problems in which state variables are not necessarily Boolean, finite or even countable. This raises the possibility of applying this algorithm to temporal planning, numeric planning, or hybrid planning. However, this might need extending the existing theory to reason about actions whose preconditions and effects are functions in state variables, versus assignments to state variables. Since we do not assume that the planning problem has a finite number of actions, we hypothesise that a lot of the theory developed here could be reused for richer planning formalisms by showing that planning problems from those formalisms could be reduced to problems represented in our theory.

We made a number of observations in our efforts which we believe provide insight into how HOL4 can be improved. A feature of HOL4 which we would cite as the most positive is the ease of modifying or adding tactics, since the entire system is completely implemented in SML. Also, automation tactics in general are reasonable, and surprisingly proved some lemmas completely automatically, modulo providing the methods with the appropriate lists of theorems. Two high-level issues we encountered were difficulties in searching for theorems (something we believe all systems struggle with) and the need to repeat theorem-hypotheses from goal to goal. This latter issue would be much-ameliorated by a mechanism akin to Isabelle's locales or Coq's sections.

References

1 Mohammad Abdulaziz, Charles Gretton, and Michael Norrish. Mechanising Theoretical Upper Bounds in Planning. In *Workshop on Knowledge Engineering for Planning and Scheduling*, 2014.

2 Mohammad Abdulaziz, Charles Gretton, and Michael Norrish. Verified Over-Approximation of the Diameter of Propositionally Factored Transition Systems. In *Interactive Theorem Proving*, pages 1–16. Springer, 2015.

3 Mohammad Abdulaziz, Charles Gretton, and Michael Norrish. A State Space Acyclicity Property for Exponentially Tighter Plan Length Bounds. In *International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI, 2017.

4 Mohammad Abdulaziz and Peter Lammich. A Formally Verified Validator for Classical Planning Problems and Solutions. In *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 474–479. IEEE, 2018.

5 Mohammad Abdulaziz, Michael Norrish, and Charles Gretton. Exploiting symmetries by planning for a descriptive quotient. In *Proc. of the 24th International Joint Conference on Artificial Intelligence, IJCAI*, pages 25–31, 2015.

6 Mohammad Abdulaziz, Michael Norrish, and Charles Gretton. Formally Verified Algorithms for Upper Bounding State Space Diameters. In *To appear in the Journal of Automated Reasoning*, 2017.

7 Eyal Amir and Barbara Engelhardt. Factored planning. In *IJCAI*, volume 3, pages 929–935. Citeseer, 2003.

8 Masataro Asai and Alex Fukunaga. Classical Planning in Deep Latent Space: Bridging the Subsymbolic-Symbolic Boundary. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 6094–6101, 2018.

9 Saddek Bensalem, Klaus Havelund, and Andrea Orlandini. Verification and validation meet planning and scheduling, 2014.

10 Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1):49–107, 2000.

11 Ronen I Brafman and Carmel Domshlak. Factored planning: How, when, and when not. In *AAAI*, volume 6, pages 809–814, 2006.

12 Julian Brunner and Peter Lammich. Formal Verification of an Executable LTL Model Checker with Partial Order Reduction. In *NASA Formal Methods Symposium*, pages 307–321. Springer, 2016.

13 Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. In *International Conference on Computer Aided Verification*, pages 334–342. Springer, 2014.

14 Robert L Constable, Paul B Jackson, Pavel Naumov, and Juan C Uribe. Constructively formalizing automata theory. In *Proof, language, and interaction*, pages 213–238, 2000.

15 Christian Doczkal, Jan-Oliver Kaiser, and Gert Smolka. A Constructive Theory of Regular Languages in Coq. In *International Conference on Certified Programs and Proofs*, pages 82–97. Springer, 2013.

16 Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A fully verified executable LTL model checker. In *International Conference on Computer Aided Verification*, pages 463–478. Springer, 2013.

17 Richard E Fikes and Nils J Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

18 Maria Fox, Richard Howey, and Derek Long. Exploration of the robustness of plans. In *AAAI*, pages 834–839, 2006.

19 Robert P Goldman, Ugur Kuter, and Tony Schneider. Using classical planners for plan verification and counterexample generation. In *Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

20 Marc Hanheide, Moritz Göbelbecker, Graham S. Horn, Andrzej Pronobis, Kristoffer Sjöö, Alper Aydemir, Patric Jensfelt, Charles Gretton, Richard Dearden, Miroslav Janícek, Hendrik Zender, Geert-Jan M. Kruijff, Nick Hawes, and Jeremy L. Wyatt. Robot task planning and explanation in open and uncertain worlds. *Artif. Intell.*, 247:119–150, 2017.

21 Klaus Havelund, Alex Groce, Gerard Holzmann, Rajeev Joshi, and Margaret Smith. Automated testing of planning models. In *International Workshop on Model Checking and Artificial Intelligence*, pages 90–105. Springer, 2008.

22 C Norris Ip and David L Dill. Verifying systems with replicated components in Murϕ. In *International Conference on Computer Aided Verification*, pages 147–158. Springer, 1996.

23 C Norris Ip and David L Dill. Verifying systems with replicated components in Murϕ. *Formal Methods in System Design*, 14(3):273–310, 1999.

24 Elena Kelareva, Olivier Buffet, Jinbo Huang, and Sylvie Thiébaux. Factored Planning Using Decomposition Trees. In *IJCAI*, pages 1942–1947, 2007.

25 C. A. Knoblock. Automatically Generating Abstractions for Planning. *Artificial Intelligence*, 68(2):243–302, 1994. γdoi:10.1016/0004-3702(94)90069-8.

26 Derek Long. The AIPS-98 Planning Competition. *AI MAgazine*, 21(2):13–32, 2000.

27 Kenneth L McMillan. Symbolic model checking. In *Symbolic Model Checking*, pages 25–60. Springer, 1993.

28 Lawrence C Paulson. A formalisation of finite automata using hereditarily finite sets. In *International Conference on Automated Deduction*, pages 231–245. Springer, 2015.

29 John Penix, Charles Pecheur, and Klaus Havelund. Using model checking to validate AI planner domain models. In *Proceedings of the 23rd Annual Software Engineering Workshop, NASA Goddard*, 1998.

30 Nir Pochter, Aviv Zohar, and Jeffrey S. Rosenschein. Exploiting Problem Symmetries in State-Based Planners. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*, 2011.

31 Buser Say and Scott Sanner. Planning in Factored State and Action Spaces with Learned Binarized Neural Network Transition Models. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden.*, pages 4815–4821, 2018.

32 Alexander Schimpf, Stephan Merz, and Jan-Georg Smaus. Construction of Büchi automata for LTL model checking verified in Isabelle/HOL. In *International Conference on Theorem Proving in Higher Order Logics*, pages 424–439. Springer, 2009.

33 Konrad Slind and Michael Norrish. A Brief Overview of HOL4. In *Theorem Proving in Higher Order Logics*, volume 5170 of *LNCS*, pages 28–32. Springer, 2008.

34 Benjamin Smith, William Millar, Julia Dunphy, Yu-Wen Tung, Pandu Nayak, Ed Gamble, and Micah Clark. Validation and verification of the remote agent for spacecraft autonomy. In *1999 IEEE Aerospace Conference. Proceedings (Cat. No. 99TH8403)*, volume 1, pages 449–468. IEEE, 1999.

35 Benjamin D Smith, Martin S Feather, and Nicola Muscettola. Challenges and Methods in Testing the Remote Agent Planner. In *AIPS*, pages 254–263, 2000.

36 Margaret H Smith, Gerard J Holzmann, Gordon C Cucullu, and BD Smith. Model Checking Autonomous Planners: Even the Best Laid Plans Must be Verified. In *2005 IEEE Aerospace Conference*, pages 1–11. IEEE, 2005.

37 Christoph Sprenger. A verified model checker for the modal μ-calculus in Coq. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 167–183. Springer, 1998.

38 Sam Toyer, Felipe W. Trevizan, Sylvie Thiébaux, and Lexing Xie. Action Schema Networks: Generalised Policies With Deep Learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 6294–6301, 2018.

39 Brian C. Williams and P. Pandurang Nayak. A Reactive Planner for a Model-based Executive. In *International Joint Conference on Artificial Intelligence*, pages 1178–1185. Morgan Kaufmann Publishers, 1997.

40 Chunhan Wu, Xingyuan Zhang, and Christian Urban. A Formalisation of the Myhill-Nerode Theorem Based on Regular Expressions (Proof Pearl). In *International Conference on Interactive Theorem Proving*, pages 341–356. Springer, 2011.

Proving Tree Algorithms for Succinct Data Structures

Reynald Affeldt
National Institute of Advanced Industrial Science and Technology, Tsukuba, Japan
reynald.affeldt@aist.go.jp

Jacques Garrigue
Graduate School of Mathematics, Nagoya University, Japan
garrigue@math.nagoya-u.ac.jp

Xuanrui Qi
Department of Computer Science, Tufts University, Medford, MA, United States
xqi01@cs.tufts.edu

Kazunari Tanaka
Graduate School of Mathematics, Nagoya University, Japan
tanaka.kazunari@k.mbox.nagoya-u.ac.jp

Abstract

Succinct data structures give space-efficient representations of large amounts of data without sacrificing performance. They rely on cleverly designed data representations and algorithms. We present here the formalization in Coq/SSReflect of two different tree-based succinct representations and their accompanying algorithms. One is the Level-Order Unary Degree Sequence, which encodes the structure of a tree in breadth-first order as a sequence of bits, where access operations can be defined in terms of Rank and Select, which work in constant time for static bit sequences. The other represents dynamic bit sequences as binary balanced trees, where Rank and Select present a low logarithmic overhead compared to their static versions, and with efficient insertion and deletion. The two can be stacked to provide a dynamic representation of dictionaries for instance. While both representations are well-known, we believe this to be their first formalization and a needed step towards provably-safe implementations of big data.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Data structures design and analysis; Software and its engineering → Formal software verification

Keywords and phrases Coq, small-scale reflection, succinct data structures, LOUDS, bit vectors, self balancing trees

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.5

Supplement Material The accompanying Coq development can be found on GitHub (see [3]).

Acknowledgements We acknowledge the support of the JSPS-CNRS bilateral program "FoRmal tools for IoT sEcurity" and the JSPS KAKENHI Grant Number 18H03204, thank the projects' participants, in particular Akira Tanaka for his comments on code extraction, and the anonymous reviewers for their comments that helped improve this paper.

1 Introduction

Succinct data structures [15] represent combinatorial objects (such as bit vectors or trees) in a way that is space-efficient (using a number of bits close to the information theoretic lower bound) and time-efficient (i.e., not slower than classical algorithms). This topic is attracting all the more attention as we are now collecting and processing large amounts of data in various domains such as genomes or text mining. As a matter of fact, succinct data

structures are now used in software products of data-centric companies such as Google [12].

The more complicated a data structure is, the harder it is to process it. A moment of thought is enough to understand that constant-time access to bit representations of trees requires ingenuity. Succinct data structures therefore make for intricate algorithms and their importance in practice make them perfect targets for formal verification [24].

In this paper, we tackle the formal verification of tree algorithms for succinct data structures. We first start by formalizing basic operations such as counting (rank) and searching (select) bits in arrays. This is an important step because the theory of these basic operations sustains most succinct data structures. Next, we formally define and verify a bit representation of trees called Level-Order Unary Degree Sequence (hereafter LOUDS). It is for example used in the Mozc Japanese input method [12]. The challenge there is that this representation is based on a level-order (i.e., breadth-first) traversal of the tree, which is difficult to describe in a structural way. Nonetheless, like most succinct data structures, this bit representation only deals with static data. Last, we further explore the advanced topic of dynamic bit vectors. The implementation of the latter requires to combine static bit vectors from succinct data structures with classical balanced trees. We show in particular how this can be formalized using a flavor of red-black trees where the data is in the leaves (rather than in the internal nodes, as in most functional implementations).

In both cases, our code can be seen as a verified functional specification of the algorithms involved. We were careful to use the right abstractions in definitions so that this specification could be easily translated to efficient code using arrays. For LOUDS we only rely on the rank and select functions; we have already provided an efficient implementation for rank [24]. For dynamic bit vectors, while the code we present here is functional, it closely matches the algorithms given in [15]. We did prove all the essential correctness properties, by showing the equivalence of each operation with its functional counterpart (functions on inductive trees for LOUDS, and on sequences of bits for dynamic bit vectors).

Independently of this verified functional specification, we identify two technical contributions, that arised while doing this formalization. One is the notion of level-order traversal up to a path in a tree, which solves the challenge of performing path-induction on a level-order traversal. Another is our experience report with using small-scale reflection to prove algorithms on inductive data, which we hope could provide insights to other researchers.

The rest of this paper is organised as follows. The next section introduces rank and select. Section 3 describes our formalization of LOUDS, including the notion of level-order traversal up to a path. Section 4 uses trees to represent bit vectors, defining not only rank and select, but also insertion and deletion. Section 5 reports on our experience. Section 6 compares with the litterature, and Section 7 concludes.

2 Two functions to build them all

The rank and select functions are the most basic blocks to form operations on succinct data structures: rank counts bits while select searches for their position. The rest of this paper (in particular Sect. 3.2 and Sect. 4) explains how they are used in practice to perform operations on trees. In this section, we just briefly explain their formalization and theory.

2.1 Counting bits with rank

The `rank` function counts the number of elements `b` (most often bits) in the prefix (i.e., up to some index `i`) of an array `s`. It can be conveniently formalized using standard list functions:

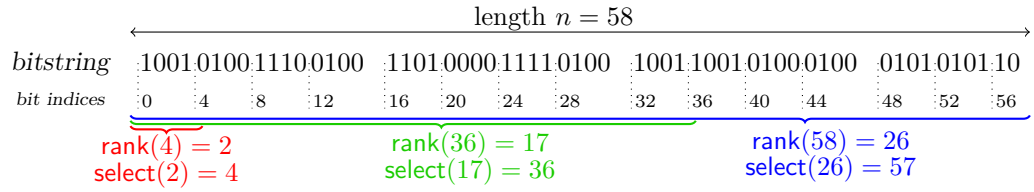


Figure 1 Examples of rank and select queries on a sample bitstring (bit indexed from 0 to 57).

```
Definition rank b i s := count_mem b (take i s).
```

Figure 1 provides several examples of rank queries. The mathematically-inclined reader can alternatively[1] think of rank as the cardinal of the number of indices of `b` bits in a tuple `B`:

```
Definition Rank (i : nat) (B : n.-tuple T) :=
  #|[set k : [1,n] | (k <= i) && (tacc B k == b)]|.
```

In this definition, `n.-tuple T` denotes[2] sequences of `T` of length `n`; `[1,n]` is the type of integers between 1 and n; and `tacc` accesses the tuple counting the indices from 1.

2.2 Finding bits with select

Intuitively, compared with rank, select performs the converse operation: it returns the index of the `i`-th occurrence of `b`, i.e., the *minimum* index whose rank is `i`. It is conveniently specified using the `ex_minn` construct of the SSReflect library [8]:

```
Variables (T : eqType) (b : T) (n : nat).
Lemma select_spec (i : nat) (B : n.-tuple T) :
  exists k, ((k <= n) && (Rank b k B == i)) || (k == n.+1) && (count_mem b B < i).
Definition Select i (B : n.-tuple T) := ex_minn (select_spec i B).
```

With this definition, select returns the index of the sought bit *plus one* (counting indices from 0); selecting the 0^{th} bit always returns 0; when no adequate bit is found, select returns the size of the array plus one. The need for the 0 case explains why it makes sense to return indices starting from 1. Figure 1 provides several examples to illustrate the select function.

2.3 The theory of rank and select

The rank and select functions are used in a variety of applications whose formal verification naturally calls for a shared library of lemmas. Our first work is to identify and isolate this theory. Its lemmas are not all difficult to prove. For instance, the fact that `Rank` cancels `Select` directly follows from the definitions:

```
Lemma SelectK n (s : n.-tuple T) (j : nat) :
  j <= count_mem b s -> Rank b (Select b j s) s = j.
```

However, as often with formalization, it requires a bit of work and try-and-error to find out the right definitions and the right lemmas to put in the theory of rank and select. For example, how appealing the definition of `Select` above may be, proving its equivalence with a functional version such as

[1] This is actually the definition that appears in Wikipedia at the time of this writing.
[2] The notation `.-tuple` is a SSReflect idiom for a suffix operator. Similarly we use `.+1` and `.-1` for successor and predecessor.

```
Fixpoint select i (s : seq T) : nat :=
  if i is i.+1 then
    if s is a :: s' then (if a == b then select i s' else select i.+1 s').+1
                    else 1
  else 0.
```

turns out to add much comfort to the development of related lemmas.

As a consequence, the resulting theory of rank and select sometimes looks technical and we therefore refer the reader to the source code [3] to better appreciate its current status. Here, we just provide for the sake of completeness the definition of two derived functions that are used later in this paper.

2.3.1 The succ and pred functions

In a bitstring, the succ function computes the position of the next 0-bit or 1-bit. It will find its use when dealing with LOUDS operations in Sect. 3.2.2. More precisely, given a bitstring `s`, `succ b s y` returns the index of the next `b` following index `y`. This operation is achieved by a combination of rank and select. First, a call to rank counts the number of `b`'s up to index `y`; let `N` be this number. Second, a call to select searches for the $(\texttt{N+1})^{\text{th}}$ `b` [15, p. 89]:

```
Definition succ (b : T) (s : seq T) y := select b (rank b y.-1 s).+1 s.
```

In particular, there is no `b` in the set $\{\texttt{s}_i \,|\, \texttt{y} \leq i < \texttt{succ b s y}\}$:

```
Lemma succP b n (s : n.-tuple T) (y : [1, n]) :
  b \notin \bigcup_(i : [1,n] | y <= i < succ b s y) [set tacc s i].
```

Conversely, the pred function computes the position of the previous bit and will find its use in Sect. 3.2.3. It is similar to succ, so that we only provide its definition for reference:

```
Definition pred (b : T) (s : seq T) y := select b (rank b y s) s.
```

3 LOUDS formalization

Operationally, a LOUDS encoding consists in turning a tree into an array of bits via a level-order traversal. Figure 2 provides a concrete example. The resulting array is the ordered concatenation of the bit representation of each node. Each node is represented by a list of bits that contains as many 1-bits as there are children and that is terminated by a 0-bit.

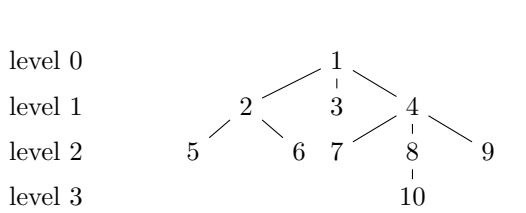
The significance of the LOUDS encoding is that it preserves the branching structure of the tree without pointers, making for a compact representation in memory. Moreover, read-only operations can be implemented using rank and select, which can be implemented in constant-time.

We explain how we formalize the LOUDS encoding in Sect. 3.1 and how we formally verify the correctness of operations on trees built out of rank and select in Sect. 3.2.

3.1 LOUDS encoding formalized in Coq

We define arbitrarily-branching trees by an inductive type:

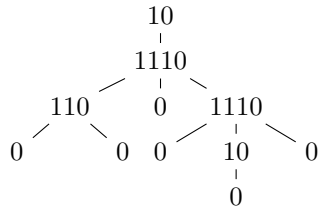
```
Variable A : Type.
Inductive tree := Node : A -> seq tree -> tree.
Definition forest := seq tree.
Definition children_of_node : tree -> forest := ...
Definition children_of_forest : forest -> forest := flatten \o map children_of_node.
```



level-ordered enumeration of nodes:

level 0	level 1	level 2	level 3
1	2, 3, 4	5, 6, 7, 8, 9	10

(a) A sample tree.



Bitstring encoding:

	level 0	level 1	level 2	level 3
10	1110	11001110	000100	0

(b) Its LOUDS encoding.

Figure 2 LOUDS encoding of a sample unlabeled tree.

where `\o` is the function composition operator (i.e., ∘), and where the type `A` is the type of labels. We also introduce the abbreviation `forest` for a list of trees, and functions to obtain children. With this definition of trees, a leaf is a node with an empty list of children. For example, the tree of Fig. 2 becomes in Coq:

```
Definition t : tree nat := Node 1
  [:: Node 2 [:: Node 5 [::]; Node 6 [::]];
      Node 3 [::];
      Node 4 [:: Node 7 [::];
                 Node 8 [:: Node 10 [::]];
                 Node 9 [::]]].
```

3.1.1 Height-recursive level-order traversal

The intuitive definition of level-order traversal iterates on a forest, returning first the toplevel nodes of the forest, then their children (applying `children_of_forest`), etc. We parameterize the definition with an arbitrary function `f` for generality.

```
Variables (A B : Type) (f : tree A -> B).
Fixpoint lo_traversal' n (s : forest A) :=
  if n is n'.+1 then map f s ++ lo_traversal' n' (children_of_forest s) else [::].
Definition lo_traversal t := lo_traversal' (height t) [:: t].
```

The parameter `n` is filled here with the maximum height of the forest, meaning that we iterate just the right number of times for the forest to become empty.

Yet, this definition is not fully satisfactory. One reason is that it is not structural: we are not recursing on a tree, but iterating on a forest, using its height as recursion index. Another one is that, as we will see in Sect. 3.2, the name *level-order* is misleading. For many proofs, we are not interested in complete traversal of the tree, level by level, but rather by partial traversal along a path in the tree, where the forest we consider actually overlaps levels.

3.1.2 A structural level-order traversal

At first it may seem that the non-structurality is inherent to level-order traversal. There is no clear way to build the sequence corresponding to the traversal of a tree from those of its children. However, Gibbons and Jones [7, 10] showed that this can be achieved by splitting the output into a list of levels. One can combine two such structured traversals by

zipping them, i.e., concatenating corresponding levels, and recover the usual traversal by flattening the list. Since concatenation of lists forms a monoid, zipping of traversals also forms a monoid.

```
Variable (A : Type) (e : A) (M : Monoid.law e).
Fixpoint mzip (l r : seq A) : seq A := match l, r with
  | (l1::ls), (r1::rs) => (M l1 r1) :: mzip ls rs
  | nil, s | s, nil    => s
  end.

Lemma mzipA : associative mzip.
Lemma mzip1s s : mzip [::] s = s.
Lemma mzips1 s : mzip s [::] = s.
Canonical mzip_monoid := Monoid.Law mzipA mzip1s mzips1.
```

Here `Monoid.Law`, from the `bigop` module of SSReflect, denotes an operator together with its neutral element (here `[::]`) and the required monoidal equations, which are also satisfied by `mzip`.

We now define our traversal by instantiating `mzip` to the concatenation monoid. The resulting `mzip_cat` is a structure of type `Monoid.law [::]` that can be used as an operator of type `seq (seq B) -> seq (seq B) -> seq (seq B)` enjoying the properties of a monoid.

```
Variables (A : eqType) (B : Type) (f : tree A -> B).
Definition mzip_cat := mzip_monoid (cat_monoid B).

Fixpoint level_traversal t :=
  [:: f t] :: foldr (mzip_cat \o level_traversal) nil (children_of_node t).

Lemma level_traversalE t :
  level_traversal t =
  [:: f t] :: \big[mzip_cat/nil]_(i <- children_of_node t) level_traversal i.

Definition lo_traversal_st t := flatten (level_traversal t).
Theorem lo_traversal_stE t : lo_traversal_st t = lo_traversal f t.
```

To let Coq recognize the structural recursion, we have to use the recursor `foldr` in the definition of `level_traversal`. Yet, the intended equation is the one expressed by `level_traversalE`, i.e., first output the image of the node, and then combine the traversals of the children. Then `lo_traversal_st` can be proved equal to the previously defined `lo_traversal`. Deforestation can furthermore improve the efficiency of `level_traversal`[3].

3.1.3 LOUDS encoding

Finally, the LOUDS encoding is obtained by instantiating `lo_traversal_st` with an appropriate function (called the *node description* of a node), and flattening once more:

```
Definition node_description s := rcons (nseq (size s) true) false.
Definition children_description t := node_description (children_of_node t).
Definition LOUDS t := flatten (lo_traversal_st children_description t).
```

Here, `rcons s x` adds `x` to the end of the sequence `s`, while `nseq n x` creates a sequence consisting of `n` copies of `x`. Note that we chose here not to add the usual "`10`" prefix [15, p. 212] shown in Fig. 2, as it appeared to just complicate definitions. It can be easily recovered by adding an extra root node, as "`10`" is the representation of a node with 1 child.

[3] See `level_traversal_cat` in [3, `tree_traversal.v`].

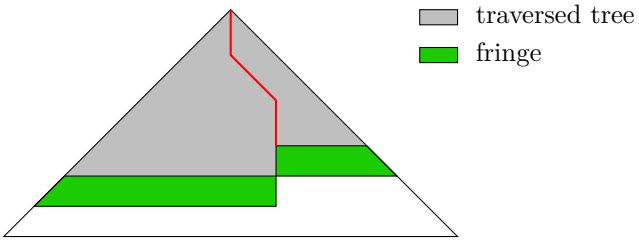


Figure 3 Level-order traversal of a tree up to a path.

For example, we can recover the encoding displayed in Fig. 2 with this definition of `LOUDS`:

```
Lemma LOUDS_t : LOUDS (Node 0 [:: t]) =
  [:: true; false; true; true; true; false;
      true; true; false; false; true; true; true; false;
      false; false; false; true; false; false; false].
```

We can also prove some properties of this representation, such as its size:

```
Lemma size_LOUDS t : size (LOUDS t) = 2 * number_of_nodes t - 1.
```

This is an easy induction, remarking that `size \o flatten \o flatten` is a morphism between `mzip_cat` and `+`.

3.2 LOUDS functions using rank and select

In this section, we formalize LOUDS functions and prove their correctness. These functions are essentially built out of rank and select. Their correctness statements establish a correspondence between operations on trees defined inductively and operations on their LOUDS encoding. We start by explaining how we represent positions in trees and then comment on the formal verification of LOUDS operations using representative examples.

3.2.1 Positions in trees

For a tree defined inductively, we represent the position of a node as usual: using a *path*, i.e., a list that records the branches taken from the root to reach the node. For example, the position of the node 8 in Fig. 2a is `[:: 2; 1]`. Not all positions are valid; we sort out the valid ones by means of the predicate `valid_position` (definition omitted for brevity).

In contrast, the position of nodes in the LOUDS encoding is not immediate. We define it as the length of the generated LOUDS up to the corresponding path. To do that, we first need to define a notion of level-order traversal up to a path, which collects all the nodes preceding the one referred by that path (which need not be valid):

```
Definition split {T} n (s : seq T) := (take n s, drop n s).
Variables (A : eqType) (B : Type) (f : tree A -> B).
Fixpoint lo_traversal_lt (s : forest A) (p : seq nat) : seq B := match p, s with
  | nil, _ | _, nil => nil
  | n :: p', t :: s' =>
    let (fs, ls) := split n (children_of_node t) in
    map f (s ++ fs) ++ lo_traversal_lt (ls ++ children_of_forest (s' ++ fs)) p'
  end.
```

This new traversal appears to be the key to clean proofs of LOUDS properties. In a previous attempt using the height-recursive level-order traversal of Sect. 3.1.1, proofs were unwieldy (one needed to manually set up inductions) and lemmas did not arise naturally. We expect this new traversal to have applications to other uses of level-order traversal.

This definition may seem scary, but it closely corresponds to the imperative version of level-order traversal, which relies on a queue: to get the next node, take it from the front of the queue, and add its children to the back of the queue. We define our traversal so that the node we have reached is the one at the front of the queue `s`. To move to its n^{th} child (indices starting from 0), we first output all the nodes in the queue, and its children up to the previous one, and proceed with a new queue containing the remaining children (starting from the n^{th}) and the children of the other nodes we have just output. Figure 3 shows how the traversal progresses. The point is that as soon as the queue spans all the fringe of the traversed tree, it is able to generate the remainder of the traversal. We can verify that `lo_traversal_lt` indeed qualifies as a level-order traversal by proving that its output converges to the full level-order traversal when the length of `p` reaches the height of the tree:

```
Theorem lo_traversal_ltE (t : tree A) (p : seq nat) :
  size p >= height t -> lo_traversal_lt [:: t] p = lo_traversal_st f t.
```

We also introduce a function that computes the fringe of the traversal up to `p`, i.e., the forest generating the remainder of the traversal.

```
Fixpoint lo_fringe (s : forest A) (p : seq nat) : forest A := ...
Lemma lo_traversal_lt_cat s p1 p2 :
  lo_traversal_lt s (p1 ++ p2) =
  lo_traversal_lt s p1 ++ lo_traversal_lt (lo_fringe s p1) p2.
```

We omit the definition but the lemma states exactly this property. It decomposes the traversal generated by a path, allowing induction from either end of the list representing the position.

Using the path-indexed traversal function, we can directly obtain the index of a node in the level-order traversal of a tree:

```
Definition lo_index (s : forest A) (p : seq nat) := size (lo_traversal_lt id s p).
```

The expression `lo_index [:: t] p` counts the number of nodes in the traversal of `t` before the position `p`. Similarly, we give an alternative definition of the LOUDS encoding, and use it to map a position in the tree to a position in its encoding (i.e., the index of the first bit of the representation of a node):

```
Definition LOUDS_lt s p := flatten (lo_traversal_lt children_description s p).
Definition LOUDS_position s p := size (LOUDS_lt s p).
```

Here the position in the whole tree is obtained as `LOUDS_position [:: t] p`, but we can also compute relative positions by using `LOUDS_position s p` where `s` is a generating forest whose front node is the one we start from. Note that both `lo_index` and `LOUDS_position` return indices starting from 0.

For example, in Fig. 2, the position of the node 8 is `[:: 2; 1]` in the inductively defined tree and `17` in the LOUDS encoding:

```
Definition p8 := [:: 2; 1].
Eval compute in LOUDS_position [:: Node 0 [:: t]] (0 :: p8). (* 17 *)
```

Finally, here is one of the essential lemmas for proofs on LOUDS, which relates `lo_index` and `LOUDS_position` using `select`:

```
Lemma LOUDS_position_select s p p' : valid_position (head dummy s) p ->
  LOUDS_position s p = select false (lo_index s p) (LOUDS_lt s (p ++ p')).
```

Namely if the index of `p` is n, then its position in the LOUDS encoding is the index of its n^{th} 0-bit (recall that `select` counts indices starting from 1). Here `p'` allows us to complete `p` to a path of sufficient length, so that `LOUDS_lt` converges to `LOUDS`.

3.2.2 Number of children using succ

As a first example, let use formalize the LOUDS function that counts the number of children of a node. For a tree defined inductively, this operation can be achieved by first walking down the path to the node and then looking at the list of its children.

```
Fixpoint subtree (t : tree) (p : seq nat) :=
  if p is n :: p' then subtree (nth t (children_of_node t) n) p' else t.
Definition children t p := size (children_of_node (subtree t p)).
```

To count the number of children of a node using a LOUDS encoding, one first has to notice that each node is terminated by a 0-bit. Given such a 0-bit (or equivalently the corresponding node), one can find the number of children by computing the distance with the next 0-bit [15, p. 214]. Finding this bit is the purpose of the succ function of Sect. 2.3.1:

```
Definition LOUDS_children (B : bitseq) (v : nat) : nat := succ false B v.+1 - v.+1.
```

The `.+1` offset comes from the fact `succ` computes on indices starting from 1.

`LOUDS_children` is correct because, when applied to the `LOUDS_position` of a position `p`, it produces the same result as the function `children`:

```
Theorem LOUDS_childrenE (t : tree A) (p p' : seq nat) :
  let B := LOUDS_lt [:: t] (p ++ 0 :: p') in
  valid_position t p -> LOUDS_children B (LOUDS_position [:: t] p) = children t p.
```

3.2.3 Parent and child node using rank and select

A path in a tree defined inductively gives direct ancestry information. In particular, removing the last index denotes the parent, and adding an extra index denotes the corresponding child. It takes more ingenuity to find parent and child using a LOUDS representation and functions from Sect. 2 alone. The idea is to count the number of nodes and branches up to the position in question [15, p. 215]. More precisely, given a LOUDS position `v`, let `Nv` be the number of nodes up to `v` (`rank false v B` computes this number). Then, `select true Nv B` looks for the `Nv`-th down-branch, which is the branch leading to the node of position `v`. Last, this branch belongs to a node whose position can be recovered using the `pred` function (from Sect. 2.3.1). Reciprocally, one computes the i^{th} child by using `rank true` and `select false`. This leads to the following definitions:

```
Definition LOUDS_parent (B : bitseq) (v : nat) : nat :=
  let j := select true (rank false v B) B in pred false B j.
Definition LOUDS_child (B : bitseq) (v i : nat) : nat :=
  select false (rank true (v + i) B).+1 B.
```

One can check the correctness of `LOUDS_parent` and `LOUDS_child` as follows. Consider a node reached by the path `rcons p i`. Its parent is the node reached by the path `p`, and conversely it is the i^{th} child of this node. We can formally prove that the LOUDS position of `p` (respectively `rcons p i`) and the position computed by `LOUDS_parent` (respectively `LOUDS_child`) coincide:

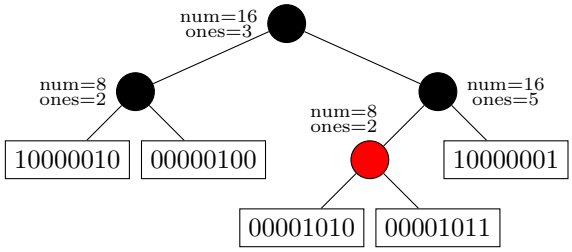


Figure 4 Example of tree representation of a dynamic bit vector.

```
Variables (t : tree A) (p p' : seq nat) (i : nat).
Hypothesis HV : valid_position t (rcons p i).
Let B := LOUDS_lt [:: t] (rcons p i ++ p').
Theorem LOUDS_parentE :
  LOUDS_parent B (LOUDS_position [:: t] (rcons p i)) = LOUDS_position [:: t] p.
Theorem LOUDS_childE :
  LOUDS_child B (LOUDS_position [:: t] p) i = LOUDS_position [:: t] (rcons p i).
```

The approach that we explained so far shows how to carry out the formal verification of the LOUDS operations that are listed in [15, Table 8.1]. However, how useful they may be for many big-data applications, these operations assume static compact data structures. The next section explains how to extend our approach to deal with dynamic structures.

4 Dynamic bit vectors

In some applications bit vectors need to support dynamic operations – not just static queries. We formalize such *dynamic bit vectors*, and implement and verify "dynamic operations" on them: inserting a bit into a bit vector, and deleting a bit from one.

In Sect. 4.1, we explain the data structure that allows for an efficient implementation of dynamic operations. In Sect. 4.2, we formalize the rank and select queries. Sections 4.3 and 4.4 are dedicated to the formalization of the more difficult insertion and deletion.

4.1 Representing dynamic bit vectors

The choice of representation for dynamic bit vectors is motivated by complexity considerations. Insertion into a linear array has time complexity $O(n)$, but we can improve this by using a balanced binary search tree to represent the bit array, which enables us to handle insertions in at most $O(w)$ time, with a trade-off of $O(n/w)$ bits of extra space, where w is a parameter controlling the width of each tree node and should no more than the size of a native machine word in bits[4] [15]: i.e., for a typical 64-bit machine, we would set w to 32 or 64.

On a side note, balanced binary trees are certainly not the most compact data structure that could be used here. In fact, various data structures with better complexity have been designed [16, 20], however those structures are complicated and are unlikely to offer practical improvements over the structure presented here [15]. As a result, we choose to work only with balanced binary trees, which are much easier to reason about.

[4] The complexity bounds referred to in this section are dependent on the model of computation used. Here, we assume that we are working with a sequential RAM machine, where we have $O(w) = O(\log n)$ as we can only address at most 2^w bits of memory.

In our formalization of the dynamic bit vector's algorithms, we use a red-black tree as our balanced tree structure. Each node holds a color and meta-data about the bit vector, and each leaf holds a *flat* (i.e., list-based) bit array. Following Navarro [15], we store two natural numbers in each node: the size and the rank of the left subtree (recorded as "num" and "ones" in Fig. 4).

```
Inductive color := Red | Black.
Inductive btree (D A : Type) : Type :=
 | Bnode of color & btree D A & D & btree D A
 | Bleaf of A.

Definition dtree := btree (nat * nat) (seq bool).
```

Our first step is to formalize the structural invariant of our tree representation of bit vectors, which is required to prove the correctness of queries and updates on it. It states that the numbers encoded in each node are the left child's size and rank, and that leaves contain a number of bits between `low` and `high`.

```
Variables low high : nat.  (* instantiated as w^2 / 2 and w^2 * 2 *)
Fixpoint wf_dtree (B : dtree) := match B with
  | Bnode _ l (num, ones) r => [&& num == size (dflatten l),
                                   ones == count_mem true (dflatten l),
                                   wf_dtree l & wf_dtree r]
  | Bleaf arr               => low <= size arr < high
  end.
```

Here, the function `dflatten` defines the semantics of our tree representation of a bit vector (`dtree`) by converting it to a flat representation of that vector:

```
Fixpoint dflatten (B : dtree) := match B with
  | Bnode _ l _ r => dflatten l ++ dflatten r
  | Bleaf s => s
  end.
```

4.2 Verifying basic queries

The basic query operations can be easily defined via traversal of the tree. We implement the queries rank, select_1, and select_0 as the Coq functions `drank`, `dselect_1`, and `dselect_0`. For example, `drank` is implemented as follows, using the (static) `rank` function from Sect. 2.1:

```
Fixpoint drank (B : dtree) (i : nat) := match B with
  | Bnode _ l (num, ones) r =>
    if i < num then drank l i else ones + drank r (i - num)
  | Bleaf s => rank true i s
  end.
```

We prove that our function `drank` indeed computes the query rank using a custom induction principle `dtree_ind`, corresponding to the predicate `wf_dtree`:

```
Lemma drankE (B : dtree) i : wf_dtree B -> drank B i = rank true i (dflatten B).
Proof. move=> wf; move: B wf i. apply: dtree_ind. (* ... *) Qed.
```

Note that our implementation is only correct on well-formed trees.

The formalization and verification of the select queries proceed along the same lines.

4.3 Implementing and verifying insertion

Insertion is significantly harder to implement than static queries. We need to maintain the invariant on the size of the leaves, which means that we have to split a leaf if it becomes too big, and in that case we may need to rebalance the tree, to maintain the red-black invariant, updating the meta-data on the way.

We translate the algorithm given by Navarro [15] directly into Coq. Here, `high` is the maximum number of bits a leaf can contain before it needs to be split up:

```
Definition dins_leaf s b i :=
  let s' := insert1 s b i in (* insert element b in sequence s at position i *)
  if size s + 1 == high then
    let n := size s' %/ 2 in let sl := take n s' in let sr := drop n s' in
    Bnode Red (Bleaf _ sl) (n, count_mem true sl) (Bleaf _ sr)
  else Bleaf _ s'.

Fixpoint dins (B : dtree) b i : dtree := match B with
  | Bleaf s => dins_leaf s b i
  | Bnode c l d r =>
      if i < d.1 then balanceL c (dins l b i) r (d.1.+1, d.2 + b)
                 else balanceR c l (dins r b (i - d.1)) d
  end.

Definition dinsert (B : btree D A) b i : btree D A :=
  match dins B b i with
  | Bleaf s => Bleaf _ s
  | Bnode _ l d r => Bnode Black l d r
  end.
```

`dins` recurses on the tree, searching for the leaf where the insertion must be done, calling then `dins_leaf`, which inserts a bit in the leaf, eventually splitting it if required. On its way back, `dins` calls balancing functions `balanceL` and `balanceR` to maintain the red-black invariant. We omit the code of the balancing functions (see [3]). Like the standard version, they fix imbalances possibly occurring on the left and on the right, respectively, but they must also adjust the meta-data in the nodes. `dinsert` is a simple wrapper over `dins` that completes the insertion by painting the root black. The real definitions are more abstract [3]; we chose to instantiate them in this paper for readability.

Verifying `dinsert` requires verifying three different properties: `dinsert` must (a) preserve the data, (b) maintain the structural invariants of the tree, and (c) return a balanced red-black tree. Properties (a) and (b) are related, in that the latter is required by the former.

```
Notation wf_dtree_l := (wf_dtree low high).
Definition wf_dtree' t := if t is Bleaf s then size s < high else wf_dtree_l t.
Lemma wf_dtree_dtree' t : wf_dtree_l t -> wf_dtree' t.
Lemma wf_dtree'_dtree t : wf_dtree' t -> wf_dtree 0 high t.

Lemma dinsertE (B : dtree) b i :
  wf_dtree' B -> dflatten (dinsert B b i) = insert1 (dflatten B) b i.
Lemma dinsert_wf (B : dtree) b i : wf_dtree' B -> wf_dtree' (dinsert B b i).
```

A subtle point here is that we may start from a tree formed of a single small leaf, i.e., a leaf smaller than `low`. To handle this situation we introduce `wf_dtree'`, which does not enforce the lower bound on this single leaf. This new predicate is entailed by the original invariant (it removes one check), but interestingly it also entails it if we set the lower bound to 0. Since

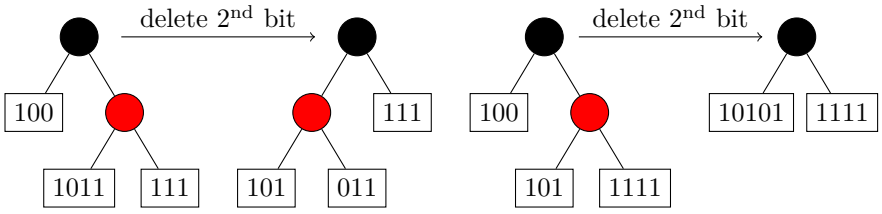


Figure 5 Base cases of delete (lower bound = 3).

the queries of Sect. 4.2 were proved with abstract lower and upper bounds, their proofs are readily usable through this weakening. However, we need to use `wf_dtree'` when we prove properties of `dinsert`, as it modifies the tree.

Proving (a) and (b) involves no theoretical difficulty. We explain in Sect. 5 some techniques to write short proofs: about 100 lines in total for both properties, including lemmas for `balanceL` and `balanceR`, which involve large case analyses.

Property (c) about `dinsert` never breaking the red-black tree invariant is notoriously more challenging. More importantly, we want to eliminate cases where the "height balance" at a node is broken. It is easy to model the property that no red node has a red child; the "height balance" property is modeled using the black-depth. We can thus model the red-black tree invariant with a recursive function that takes as arguments the "color context" `ctxt` (the color of the parent's node) and the black-depth of the node `bh`:

```
Fixpoint is_redblack (B : dtree) (ctxt : color) (bh : nat) := match B with
  | Bleaf _ => bh == 0
  | Bnode c l _ r => match c, ctxt with
    | Red, Red   => false
    | Red, Black => is_redblack l Red bh && is_redblack r Red bh
    | Black, _   => (bh > 0) && is_redblack l Black bh.-1
                             && is_redblack r Black bh.-1
  end end.
```

To show that `dinsert` preserves the red-black tree property, we define and prove a number of weaker structural lemmas that are basically equivalent to stating that a tree returned by `dins` is structurally valid if the root is painted black. We do not describe the proof in detail because the technique is well-known [18] and has been formalized in multiple sources (see Sect. 6). Using these weaker lemmas, we can prove the following structural validity lemma:

```
Lemma dinsert_is_redblack (B : dtree) b i n :
  is_redblack B Red n -> exists n', is_redblack (dinsert B b i) Red n'.
```

4.4 Deletion: searching for invariants

Deletion in dynamic bit vectors is difficult for two reasons. One is that, in order to maintain the upper and lower bounds on the size of leaves, which is required to attain simultaneously space and time efficiency, deleting a bit in a leaf may require some rearrangement of the surrounding nodes. Figure 5 shows the result of deleting a bit in a leaf of the tree, when this leaf has already the smallest allowed size. This can be resolved by borrowing a bit from a sibling (left case), or merging two siblings (right case), but depending on the configurations of nodes, this may require to first rotate the tree.

The other is that deletion in a functional red-black tree is a complex operation [11], and that finding how to adapt the invariants of the litterature to our specific case proved to be non-trivial. Therefore, we took a twofold approach. First, we searched for invariants in a concrete tree structure with invariants encoded using dependent types. Then, we removed dependent types and implemented delete and proved its correctness (more details in Sect. 5).

Contrary to insertion, knowing the color of the modified child is not sufficient to rebalance its parent correctly after deletion, and recompute its meta-data. We need to propagate two more pieces of information: whether the black-height decreased (`d_down` below), and the meta-data corresponding to the deleted bit (`d_del`). We encapsulate these in a "tree state":

```
Record deleted_dtree: Type := MkD { d_tree :> dtree; d_down: bool; d_del: nat*nat }.
```

Note that `deleted_dtree` is automatically coerced to `dtree`.

Now, we can define delete in the natural way, but we need to take care about balance operations and invariants on the size of leaves. Specifically, the balance operations must be reimplemented as `balanceL'` and `balanceR'`, which need to satisfy the following invariants, i.e., the resulting "balanced" tree is *deleted-red-black* (i.e., a red-black tree, either with the same black height, or with a black root and decreased black height), given that the unproblematic subtree is red-black, while the unbalanced one is deleted-red-black.

```
Definition balanceL' (c:color)(l:deleted_dtree)(d:nat*nat)(r:dtree):deleted_dtree :=
Definition balanceR' (c:color)(l:dtree)(d:nat*nat)(r:deleted_dtree):deleted_dtree :=

Definition is_deleted_redblack tr (c : color) (bh : nat) :=
  if d_down tr then is_redblack tr Red bh.-1 else is_redblack tr c bh.

Lemma balanceL'_Black_deleted_is_redblack l r n c :
  0 < n -> is_deleted_redblack l Black n.-1 -> is_redblack r Black n.-1 ->
  is_deleted_redblack (balanceL' Black l r) c n.
Lemma balanceL'_Red_deleted_is_redblack l r n :
  is_deleted_redblack l Red n -> is_redblack r Red n ->
  is_deleted_redblack (balanceL' Red l r) Black n.
(* similar statements with respect to balanceR' *)
```

Regarding leaves, we need special processing in the base cases of delete, as illustrated in Fig. 5. delete might have to "borrow" a bit from a sibling of a target leaf or combine target siblings (possibly after a rotation), to preserve the size invariants. Afterwards, delete will recursively rebalance the whole `dtree`.

Thus we implement delete (as `ddel`), and prove its correctness as follows:

```
Fixpoint ddel (B : dtree) (i : nat) : deleted_dtree := ...

Lemma ddeleteE B i : wf_dtree' B -> dflatten (ddel B i) = delete (dflatten B) i.
Lemma ddelete_wf (B : dtree) n i :
  is_redblack B Black n -> i < dsize B -> wf_dtree' B -> wf_dtree' (ddel B i).
Lemma ddelete_is_redblack B i n :
  is_redblack B Red n -> exists n', is_redblack (ddel B i) Red n'.
```

These statements are variants of the properties (a), (b) and (c) of Sect. 4.3. The proofs are complicated by the huge number of cases, handled using the proof techniques discussed in the next section.

5 Using small-scale reflection with inductive data

The small-scale reflection approach is known to be beneficial for mathematical proofs [14]. However, while SSReflect tactics are now widely used in the Coq community, it is not always clear how to write proofs of programs using inductive data structures in an idiomatic style, in particular in presence of deep case analysis.

In the first part of the paper, concerning level-order traversal, the question is not so acute, as the induction principle we need for LOUDS is not structural on the shape of trees, but rather on paths, represented as lists, which are already well supported by the SSReflect library. Thus the question was the more traditional one of which definitions to use, so that we can obtain natural lemmas. This proved to be a time consuming process, which led to gradually build a library of lemmas, resulting in proofs that match the intuition, using almost only case analysis and rewriting.

However, the second part, about dynamic bit vectors, uses heavily structural induction on binary trees, and required developing some proof techniques to streamline the proofs.

A basic idea of small-scale reflection is to use recursive Boolean predicates (i.e., recursive computable functions) rather than inductive propositions. We have already presented two examples: `wf_dtree` and `is_redblack`. Properly designed, they allow one to prune case analysis by reducing to `false` on impossible cases. On the other hand, they do not decompose naturally in inductive proofs, which led us first to apply a standard technique: define a specialized induction principle for trees satisfying `wf_dtree` (`dtree_ind` in Sect. 4.2). Using it, the correctness of static queries and non-structural modification operations (i.e., setting and clearing of bits) were easy to prove, as the case analysis was trivial.

Properties of `dinsert`, `ddel`, and their auxiliary functions are trickier to prove, as they require complex case analyses and delicate re-balancing of branches. Nevertheless, we essentially applied the same principle of solving goals through direct case analysis. With this approach, the correctness lemmas (which state that our operations are semantically correct) were largely automated, consistent with prior research [17]. The structural lemmas were harder to prove, mainly due to the sheer number of cases involved and the complexity of invariants. Our proofs proceed by first applying case analysis to the tree up to the required depth, and then decomposing all assumptions to repeatedly rewrite the goal using them until it is solved. This proof pattern is captured by the following tactic:

```
Ltac decompose_rewrite :=
  let H := fresh "H" in case/andP || (move=>H; rewrite ?H ?(eqP H)).
```

It is reminiscent of the `intuition` tactic, a generic tactic for intuitionistic logic which breaks both hypotheses and goals into pieces; here we rather rely on rewriting inside Boolean conjunctions to solve goals piecewise. For `dinsert`, this approach instantly finishes most of our proofs, especially those about red-black tree invariants; the few cases that require manual treatment being usually handled in one single `rewrite`. This is true for most auxiliary functions of `ddel` too, with one caveat: where `dinsert` has us generate a dozen cases, `ddel` requires hundreds. To cope with this, we had first to decompose the case analysis in steps, solving most cases on the way, which means losing some simplicity to speed up proof search. The proof is still mostly automatic: apply `decompose_rewrite`, and throw in relevant lemmas. When possible, it appears that using `apply` instead of `rewrite` speeds up by a factor of 2 or more, which matters when the lemma takes more than 1 minute to prove. We have only 3 such time-consuming case analyses, one for each invariant. Among the 12 lemmas involved in proving the invariants, only the inductive proof of well-formedness for `ddel` seems to show the limit of this approach, as it required specific handling for each case of the function definition.

Table 1 Implementation of dynamic bit vectors (see Table 2 for the whole implementation).

Contents (`Section` in `dynamic_redblack.v`)	Lines of code	Lines of proof
Definitions (`btree`, `dtree`)	19	18
Queries (`dtree`)	38	58
Insertion (`insert`, `dinsert`)	65	208
Set/clear a bit (`set_clear`)	25	120
Deletion (`delete`, `ddelete`)	98	215

Table 2 Formalization overview [3] (see Table 1 for the details about dynamic bit vectors).

File in [3]	Section in this paper
`rank_select.v`	Sections 2.1, 2.2, 2.3
`pred_succ.v`	Sect. 2.3.1
`tree_traversal.v`	Sections 3.1.1, 3.1.2
`louds.v`	Sections 3.1.3, 3.2
`dynamic_redblack.v`	Sect. 4

For comparison, Table 1 provides the size of code and proof required for each `Section` of our proof script. This does not include lemmas about the list-based reference implementation. Note that we count all Boolean predicates used to model properties as proofs.

The proofs of set and clear, which we did not describe here, may seem relatively verbose. We prove the same properties (a,b,c) as in Sect. 4.3, but the number of lines hides a disparity between proofs of (a) correctness and (c) red-blackness, which are almost immediate, as the structure of the tree is unchanged, and (b) invariants of the meta-data, for which switching a bit requires to propagate the difference back to the root, with extra local invariants.

Last, we mention our experience with alternative approaches. In parallel with our developement using small-scale reflection, we attempted to formalize dynamic bit vectors using dependent types, where all invariants are encoded in the type of the data itself. While this guarantees that we never forget an invariant, difficulties with the `Program` [23] environment led us to write some functions using tactics [3, `dynamic_dependent_tactic.v`]. As written in Sect. 4.4, this direct connection between code and proof actually helped us discover some tricky invariants. However, the resulting code does not lend itself to further analysis, hence our choice here to stick to a more conventional separation between code and proof. We did eventually succeed in re-implementing the dependently-typed version using the `Program` environment, but at the price of very verbose definitions [3, `dynamic_dependent_program.v`].

Table 2 gives an at-a-glance overview of our entire Coq development, with a list of files and their corresponding sections in this paper.

6 Related work

Coq has been used to formalize a constant-time, $o(n)$-space rank function that was furthermore extracted to efficient OCaml code [24] and C code [25]. This work focuses on the rank query for static bit arrays while our work extends the toolset for succinct data structures with more queries (select, succ, etc.) and dynamic structures.

The functions `level_traversal` and `lo_traversal_st` of Sect. 3.1.2 match functions given in squiggle notation in related work by Jones and Gibbons [10]. In this work, the `mzip` function of Sect. 3.1.2 also appears and is called "long zip with plussle". To the best of our knowledge, the function `lo_traversal_lt` is original to our work.

Larchey-Wendling and Matthes recently studied the certification and extraction of breadth-first traversals [13]. They too define `lo_traversal_st`, but then prove it equivalent to a queue based algorithm, which they extract to efficient OCaml code. Their goal is orthogonal to ours, as for succinct data structures what matters is not the efficiency of the traversal, but the correctness of the parent/child navigation functions, which by definition require a constant number of queries.

One may use any kind of balanced binary tree to represent dynamic bit vectors [15]. There are many purely-functional balanced binary search trees, such as AVL trees [2] and weight-balanced trees [1], but purely functional red-black trees [11, 18] are most widely studied and preferred by us. As a matter of fact, they have already been formalized in Coq [4, 5, 6], Agda [19], and Isabelle [17].

We had to re-implement red-black trees due to the difference of stored contents. Above Coq formalizations are intended to represent sets, and maintain the ordering invariant. Our trees represent vectors, and maintain both that the contents (as concatenation of the leaves) are unchanged, and that meta-data in inner nodes is correct (see Sect. 4.1). Still, we found many hints in related work. For example, in Sect. 4.3 about insertion, the balancing functions use Okasaki's well-known purely functional balance algorithm [18], and we formulate our invariants and propositions similarly to above Coq formalizations.

There are now many proofs of programs that use SSReflect, but we could not find much discussion trying to synthesize the new techniques put at work. Sergey et al. used SSReflect for teaching [21, 22], observing benefits for clarity and maintainability, but also giving examples of custom tactics needed to prove programs. Gonthier et al. [9] have shown how, in some cases, one can avoid relying on ad hoc tactics through an advanced technique involving overloading of lemmas. The techniques we describe in Sect. 5, while more rudimentary, are simple and efficient, yet we have not seen them described elsewhere.

7 Conclusion

We reported on an effort to formalize succinct data structures. We started with a foundational theory of the rank and select functions for counting and searching bits in immutable arrays. Using this theory, we formalized a standard compact representation of trees (LOUDS) and proved the correctness of its basic operations. Last, we formalized dynamic bit vectors: an advanced topic in succinct data structures.

Our work is a first step towards the construction of a formal theory of succinct data structures. We already overcame several technical difficulties while dealing with LOUDS trees: it took much care to find suitable recursive traversals and to sort out the off-by-one conditions when specifying basic operations. Similarly, the formalization of dynamic vectors could not be reduced to the matter of extending conservatively an existing formalization of balanced trees: we needed to re-implement them to accommodate specific invariants.

As for future work, we plan to enable code extraction for the functions we have been verifying, and prove their complexity, so as to complete previous work [24] and ultimately achieve a formally verified implementation of succinct data structures. We have already shown that the LOUDS representation of a tree with n nodes uses just $2n$ bits of data. For the LOUDS operations, constant time complexity is a direct consequence of their being implemented using a constant number of rank and select operations. For dynamic bit vectors, we will first need to properly define a framework for space and time complexity.

References

1 Stephen Adams. Functional Pearls: Efficient sets—a balancing act. *Journal of Functional Programming*, 3(4):553–561, October 1993.

2 G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics–Doklady*, 3(5):1259–1263, September 1962.

3 Reynald Affeldt, Jacques Garrigue, Xuanrui Qi, and Kazunari Tanaka. A Coq formalization of succinct data structures. `https://github.com/affeldt-aist/succinct`, 2018.

4 Andrew W. Appel. Efficient Verified Red-Black Trees. Available at `http://www.cs.princeton.edu/~appel/papers/redblack.pdf` (code included in Coq standard library, file `MSetRBT.v`, with extra modifications by Pierre Letouzey), September 2011.

5 Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013.

6 Jean-Christophe Filliâtre and Pierre Letouzey. Functors for Proofs and Programs. In *Proceedings of the 13th European Symposium on Programming (ESOP 2004), Barcelona, Spain, March 29–April 2, 2004*, volume 2986 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2004. Source code about red-black trees available at `https://github.com/coq-contribs/fsets/SetRBT.v`.

7 Jeremy Gibbons. *Algebras for Tree Algorithms*. PhD thesis, Programming Research Group, Oxford University, 1991. Available as Technical Monograph PRG-94.

8 Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Technical report, INRIA, 2008. Version 17 (Nov 2016).

9 Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. How to make ad hoc proof automation less ad hoc. *Journal of Functional Programming*, 23(4):357–401, 2013.

10 Geraint Jones and Jeremy Gibbons. Linear-time Breadth-first Tree Algorithms: An Exercise in the Arithmetic of Folds and Zips. Technical Report 71, Department of Computer Science, University of Auckland, 1993.

11 Stefan Kahrs. Red-black trees with types. *Journal of Functional Programming*, 11(4):425–432, July 2001.

12 Taku Kudo, Toshiyuki Hanaoka, Jun Mukai, Yusuke Tabata, and Hiroyuki Komatsu. Efficient dictionary and language model compression for input method editors. In *Proceedings of the Workshop on Advances in Text Input Methods (WTIM 2011)*, pages 19–25, 2011.

13 Dominique Larchey-Wendling and Ralph Matthes. Certification of Breadth-First Algorithms by Extraction. In *Proceedings of the 13th International Conference on Mathematics of Program Construction*, 2019.

14 Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Available at: `https://math-comp.github.io/mcb/`, 2016. With contributions by Yves Bertot and Georges Gonthier.

15 Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.

16 Gonzalo Navarro and Kunihiko Sadakane. Fully Functional Static and Dynamic Succinct Trees. *ACM Transactions on Algorithms*, 10(3), 2014.

17 Tobias Nipkow. Automatic Functional Correctness Proofs for Functional Search Trees. In *Interactive Theorem Proving (ITP 2016)*, volume 9807 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 2016.

18 Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

19 Julien Oster. An Agda implementation of deletion in Left-leaning Red-Black trees. Available at `https://www.reinference.net/llrb-delete-julien-oster.pdf`, March 2011.

20 Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct Dynamic Data Structures. In *Proceedings of the 7th International Workshop on Algorithms and Data Structures*, pages 426–437. Springer, 2001.

21 Ilya Sergey. Programs and Proofs: Mechanizing Mathematics with Dependent Types (Lecture Notes). Available at `https://ilyasergey.net/pnp/`.

22 Ilya Sergey and Aleksandar Nanevski. Introducing Functional Programmers to Interactive Theorem Proving and Program Verification (Teaching Experience Report). Available at `https://ilyasergey.net/papers/teaching-ssr.pdf`, 2015.

23 Matthieu Sozeau. *Un environnement pour la programmation avec types dépendants*. PhD thesis, Université de Paris-Sud, 2008.

24 Akira Tanaka, Reynald Affeldt, and Jacques Garrigue. Formal Verification of the rank Algorithm for Succinct Data Structures. In *18th International Conference on Formal Engineering Methods (ICFEM 2016), Tokyo, Japan, November 14–18, 2016*, volume 10009 of *Lecture Notes in Computer Science*, pages 243–260. Springer, November 2016.

25 Akira Tanaka, Reynald Affeldt, and Jacques Garrigue. Safe Low-level Code Generation in Coq using Monomorphization and Monadification. *Journal of Information Processing*, 26:54–72, 2018.

Data Types as Quotients of Polynomial Functors

Jeremy Avigad
Department of Philosophy, Carnegie Mellon University, Pittsburgh, PA, USA
http://www.andrew.cmu.edu/user/avigad/
avigad@cmu.edu

Mario Carneiro
Department of Philosophy, Carnegie Mellon University, Pittsburgh, PA, USA
di.gama@gmail.com

Simon Hudon
Department of Philosophy, Carnegie Mellon University, Pittsburgh, PA, USA
https://www.cmu.edu/dietrich/philosophy/people/postdoc-fellows/simon-hudon%20.html
simon.hudon@gmail.com

Abstract

A broad class of data types, including arbitrary nestings of inductive types, coinductive types, and quotients, can be represented as quotients of polynomial functors. This provides perspicuous ways of constructing them and reasoning about them in an interactive theorem prover.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Type theory; Theory of computation → Data structures design and analysis

Keywords and phrases data types, polynomial functors, inductive types, coinductive types

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.6

Supplement Material Lean formalizations are online at `https://github.com/avigad/qpf`.

Funding Work partially supported by AFOSR grant FA9550-18-1-0120 and the Sloan Foundation.

Acknowledgements We are grateful to Andrei Popescu, Dmitriy Traytel, and Jasmin Blanchette for extensive discussions and very helpful advice.

1 Introduction

Data types are fundamental to programming, and theoretical computer science provides abstract characterizations of such data types and principles for reasoning about them. For example, an *inductive type*, such as the type of lists of elements of type α, is freely generated by its constructors:

```
inductive list (α : Type)
| nil  : list
| cons : α → list → list
```

Such a declaration gives rise to a type constructor, `list`, constructors `nil` and `cons`, and a recursor:

```
list.rec {α β} : β → (α → list α → β → β) → list α → β
```

The recursor satisfies the following equations:

```
list.rec b f nil        = b
list.rec b f (cons a l) = f a l (list.rec b f l)
```

We also have an induction principle:

```
∀ {α} (P : list α → Prop), P nil → (∀ a l, P l → P (cons a l)) → ∀ l, P l
```

In words, to prove that a property holds of all lists, it is enough to show that it holds of the empty list and is preserved under the cons operation. Here we have adopted the syntax of the Lean theorem prover [19], so the curly braces around the type arguments α and β indicate that these arguments are generally left implicit and inferred from context. The type of the variable `P`, namely `list` $\alpha \to$ `Prop`, indicates that it is a predicate on lists.

Dual to the notion of an inductive type is the notion of a *coinductive* type, such as the type of streams of elements of α:

```
coinductive stream (α : Type)
| cons (head : α) (tail : stream) : stream
```

Our syntax is similar to that used for the inductive declaration, but the fundamental properties of such a type can be expressed naturally in terms of the *destructors* rather than the constructors. Roughly speaking, when one observes a stream of elements of α, one sees an element of α, the *head*, and another stream, the *tail*. In addition to the type constructor `stream` and the destructors `head` and `tail`, one obtains a corecursor:

```
stream.corec {α β} : (β → α × β) → β → stream α
```

It satisfies these defining equations:

```
head (stream.corec f b) = fst (f b)
tail (stream.corec f b) = stream.corec f (snd (f b))
```

We also have a coinduction principle:

```
∀ {α} (R : stream α → stream α → Prop),
    (∀ x y, R x y → head x = head y ∧ R (tail x) (tail y)) →
  ∀ x y, R x y → x = y
```

Intuitively, the corecursor allows one to construct a stream from an element b of β by giving an element of α, the head, and another element of β to continue the construction. The coinductive principle says we can prove two streams are equal by showing that they satisfy a relation on streams that implies the heads are the same and the tails are again related.

Inductive definitions date to the beginning of modern logic, with inductive characterizations of the natural numbers by Dedekind [20] and Frege [24], and generalizations by Knaster [31], Tarski [39], Kreisel [32], and many others (e.g. [3]). The general study of coinductive definition seems to have originated with Aczel [3, 4], and has since been extended by many others (e.g. [6, 9, 37]).

The algebraic study of data types begins with the observation that many constructions are *functorial* in their arguments. For example, an element x of $\mathsf{list}(\alpha)$ and a function $f : \alpha \to \beta$ give rise to an element of $\mathsf{list}(\beta)$, the result of applying f to each element in the list. Similarly, products $\alpha \times \beta$ and sums $\alpha + \beta$ are functorial in either argument. In category-theoretic notation, given a functor $F(\alpha)$, one would write $F(f)$ to denote the map from $F(\alpha)$ to $F(\beta)$. In Lean, given a functor F, we can write `f <$> x` to denote $F(f)(x)$, since the system can infer `F` from the type of `x`, namely, `F` α. In this paper, we will generally use category-theoretic notation and the language of set-valued functors when talking about the general constructions, to be consistent with the general literature. When we focus on dependent type theory and our formalizations, however, we will resort to type-theoretic syntax and take α to be a type rather than a set.

For any functor F from sets to sets, a function $F(\alpha) \to \alpha$ is known as an *F-algebra*, and specifying an inductive definition amounts to specifying such an algebra. For example, the natural numbers are defined by a constant $0 \in \mathbb{N}$ and a function from $\mathbb{N}$ to $\mathbb{N}$. Putting

these together yields a function $1 + \mathbb{N} \to \mathbb{N}$, where 1 denotes a singleton set and $+$ denotes a disjoint union. So the constructors for $\mathbb{N}$ amount to a function $F(\mathbb{N}) \to \mathbb{N}$, where $F(\alpha)$ is the functor $1 + \alpha$. The inductive character of the natural numbers means that $\mathbb{N}$ is an *initial* F-algebra, in the sense that for any F-algebra $F(\alpha) \to \alpha$, there is a function rec from $\mathbb{N}$ to α such that the following square commutes:

$$\begin{array}{ccc} F(\mathbb{N}) & \xrightarrow{F(\mathsf{rec})} & F(\alpha) \\ \downarrow & & \downarrow \\ \mathbb{N} & \xrightarrow{\mathsf{rec}} & \alpha \end{array}$$

Similarly, $\mathsf{list}(\alpha)$ is an initial algebra for the functor $F(\beta) = 1 + \alpha \times \beta$. Dually, a *coalgebra* for a functor $F(\alpha)$ is a function $\alpha \to F(\alpha)$. A coinductive definition corresponds to a *final coalgebra*, that is, a coalgebra $\lambda \to F(\lambda)$ with the property that for any colagebra $\alpha \to F(\alpha)$, there is a map from $\alpha \to \lambda$ making the corresponding square commute.

The question is then: Which set-valued functors have initial algebras and final coalgebras? Not all do: initial algebras and final coalgebras are both fixed points (that is, satisfy $F(\alpha) \simeq \alpha$), so the usual diagonalization argument shows that the power-set functor has neither. A sufficient condition for the existence of both is that the functor F is κ-*bounded* for some cardinal κ [6, 37].

To formalize these constructions in the context of simple type theory, developers of the *Isabelle* theorem prover [35] proposed the notion of a *bounded natural functor* [12, 15], or *BNF* for short. A functor $F(\alpha)$ is a BNF if it satisfies the following:

- There is a natural transformation set which for each α maps elements of $F(\alpha)$ to elements of the power set of α, such that for any pair of maps $f, g : \alpha \to \beta$ and any element of x of $F(\alpha)$, if f and g agree on $\mathsf{set}(x)$, then $F(f)(x) = F(g)(x)$.
- There is a fixed cardinal $\kappa \geq \aleph_0$ such that for every x, $|\mathsf{set}(x)| \leq \kappa$.
- F preserves weak pullbacks (see Section 5).

The generalization to multivariate functors is straightforward. BNFs are closed under composition and formation of initial and final coalgebras, and the class of BNFs includes data type constructions such as finite sets and finite multisets. This forms the basis for a powerful and extensible data type package [12, 15].

Here we present a variation on the BNF constructions based on the notion of a *quotient of a polynomial functor*, or QPF for short. Like BNFs, QPFs support definitions such as the following, which defines a well-founded tree on α to consist of a node labeled by an element of α together with a finite set of subtrees :

```
inductive tree (α : Type)
| mk : α → finset tree → tree
```

Here finset can be replaced by list, multiset, or stream, or, indeed, any QPF constructor. Moreover, replacing `inductive` by `coinductive` yields the corresponding coinductive type of arbitrary trees, not just the well-founded ones.

QPFs are more general than BNFs in a sense we will make precise in Section 5, but their main appeal is that they provide another perspective on the BNF constructions, and are amenable to formalization. Our approach is well-suited to dependent type theory, and the components of our constructions, including polynomial functors, W types, M types, and quotients, are familiar inhabitants of the type theory literature. At the same time, the use of dependent types is mild, and the constructions can easily be translated to the language of set theory or simple type theory.

We have found that working with QPFs is natural and intuitive. Indeed, after hitting upon the notion, we discovered that Adámek and Porst [5, Proposition 5.2] have shown that a functor is accessible if and only if it is a quotient of a polynomial functor (see also [6, Example 6.4]). But we have not seen constructions of initial algebras and final coalgebras carried out directly in these terms, and the approach via QPFs makes them easy to understand. We expect that the QPF perspective will also be conducive to generalizations, as discussed in Section 7.

The constructions described in this paper have been formalized in Lean and are available online, as indicated below the abstract to this paper. Lean's underlying logical framework, like that of Coq, is a variant of the *Calculus of Inductive Constructions*, which has a computational interpretation. This version provides non-nested inductive type families with only primitive recursors, following the specification of Dybjer [21]. On top of the core logic, Lean's library includes propositional extensionality, quotient types [18, 8], and a classical choice principle [8]. Our constructions make use of the first two, which imply function extensionality. We had to use the choice principle in only one place, when constructing the QPF instance for the final coalgebra of a multivariate constructor. We believe the use of Lean's built-in quotient types could be avoided, but we do not know whether it is possible to avoid propositional and function extensionality.

We are in the process of implementing a data type package for Lean based on these constructions. Lean currently supports nested inductive definitions, compiling them down to indexed inductive definitions and compiling recursive definitions down to well-founded recursion along a synthesized measure of size. Our approach, like the Isabelle approach, adds a wealth of new constructions: not only coinductive definitions, but also arbitrary nestings of inductive definitions, coinductive definitions, and quotient constructions. Moreover, it provides principles of recursion and corecursion based on the associated functorial map.

2 Polynomial functors

Let us start with the notion of a *polynomial functor*, also known as a *container* [1, 2, 25]. These are functors of the form $P(\alpha) = \Sigma_{x \in A} (B(x) \to \alpha)$, where B denotes a family of sets $(B(x))_{x \in A}$ and Σ denotes the *dependent sum*. Thus, an element of $P(\alpha)$ is a pair (a, f) with $a \in A$ and $f : B(a) \to \alpha$. Think of (a, f) as representing a structured object with data from α, where $a \in A$ specifies the *shape* of the element and $f \in B(a) \to \alpha$ specifies its *contents*. In the literature on containers, the polynomial functor given by the data A and B is usually denoted $A \triangleright B$. There is an obvious functorial action: given $g : \alpha \to \beta$, $P(g)$ maps (a, f) in $P(\alpha)$ to $(a, g \circ f)$ in $P(\beta)$, preserving the shape while transforming the contents. Below, we will say more generally that P is a polynomial functor if it is *isomorphic* to one of this form. It is easy to define these in Lean:

```
structure pfunctor := (A : Type u) (B : A → Type u)

variable P : pfunctor.{u}

def apply (α : Type u) := Σ x : P.A, P.B x → α

def map {α β : Type u} (g : α → β) : P.apply α → P.apply β :=
λ ⟨a, f⟩, ⟨a, g ∘ f⟩
```

In these definitions, `Type u` denotes a fixed but arbitrary universe of types. Lean's projection notation is a convenient syntactic device: since `P` has type `pfunctor`, Lean interprets `P.apply` as `pfunctor.apply P`. Similarly, the corner brackets denote *anonymous constructors*: since

`P.apply` reduces to a sigma type, Lean interprets ⟨`a, f ∘ g`⟩ as `sigma.mk a (f ∘ g)`. We also make use of a pattern-matching lambda, which translates the variables in the bound pattern to applications of destructors of a single bound variable.

Many familiar data types are polynomial functors. For instance, any list of elements of α is given by its length, n, and a function from $\{0, \ldots, n-1\}$ to α. Streams of elements of α have only one shape, and the contents are functions from $\mathbb{N}$ to α. The type of lazy lists of elements of α can be seen as the disjoint union of these two, so the set of shapes is the disjoint union of $\mathbb{N}$ and a singleton. A tree with nodes labeled by α has as shape the unlabeled tree and as contents a map from the nodes to α.

It is not hard to show that if $P(\alpha)$ and $Q(\alpha)$ are polynomial functors, then so is their composition, $P(Q(\alpha))$. Moreover, every polynomial functor P has an initial algebra, a familiar construct in dependent type theory known as a W *type* [34]. Elements of the data type W_P corresponding to P can be viewed as well-founded trees in which every node has a label a from A and children indexed by $B(a)$. In other words, an element of W_P is given by an element a in A and a function $f : B(a) \to W_P$. Such inductive types are given axiomatically by Lean's type-theoretic foundation, and can be declared as follows:

```
inductive W (P : pfunctor)
| mk (a : P.A) (f : P.B a → W) : W
```

The constructor forms an element of W from $a \in A$ and $f : B(a) \to W$. This is just a variant of the usual algebra map $P(W) \to W$ in which the argument, an element of $P(W)$, has been replaced by its two components. The built-in recursion principle for the type above says exactly that this map is the initial algebra.

Every polynomial functor P also has a final coalgebra, known as the associated M *type*. The data type M_P has the same description as above except that the trees are no longer required to be well founded. Abstractly, M types can be specified as follows:

```
def M (P : pfunctor.{u}) : Type u

def M_dest : M P → P.apply (M P)

def M_corec : (α → P.apply α) → (α → M P)

theorem M_dest_corec (g : α → P.apply α) (x : α) :
  M_dest (M_corec g x) = M_corec g <$> g x

theorem M_bisim (r : M P → M P → Prop)
  (h : ∀ x y, r x y →
      ∃ a f g, M_dest x = ⟨a, f⟩ ∧ M_dest y = ⟨a, g⟩ ∧ ∀ i, r (f i) (g i)) :
    ∀ x y, r x y → x = y
```

The principle `M_bisim` is a coinduction principle for trees. Here the anonymous constructor $\langle a, f \rangle$ is used to represent an element of $P(M)$ in terms of $a : A$ and $f : B(a) \to M$. A *bisimulation* relation between trees is a relation r such that $r(x, y)$ holds if and only if x and y have the same branching type at the top node and the children at the top node are again pointwise related by r. Two trees are *bisimilar* if there is a bisimulation between them, and the principle `M_bisim` says that any two trees that are bisimilar are in fact equal.

M types are not given axiomatically by the Calculus of Inductive Constructions, but as the specification above suggests, they can be defined in Lean. One way to go about it is to define for each n the type of trees of depth at most n, using a special node to denote potential continuations at the leaves:

```
inductive M_approx : ℕ → Type u
| continue  : M_approx 0
| intro {n} : ∀ a, (P.B a → M_approx n) → M_approx (n + 1)
```

We can then say what it means for an approximation of depth n to agree with one of depth $n+1$, and define an element of the M type to be a sequence of approximations such that each is consistent with the next:

```
inductive agree : ∀ {n : ℕ}, M_approx P n → M_approx P (n+1) → Prop
| continue (x : M_approx P 0) (y : M_approx P 1) : agree x y
| intro {n} {a} (x : P.B a → M_approx P n) (y : P.B a → M_approx P (n+1)) :
  (∀ i, agree (x i) (y i)) → agree (M_approx.intro a x) (M_approx.intro a y)

structure M := (approx : Π n, M_approx P n) (agrees : ∀ n, agree (x n) (x (n+1)))
```

We will see in Section 4 that these considerations extend to the multivariate case. In other words, there is a natural notion of an n-ary polynomial functor, and if $P(\vec{\alpha}, \beta)$ is an $(n+1)$-ary polynomial functor, then for each fixed tuple $\vec{\alpha}$, it has an initial algebra $W(\vec{\alpha})$ and a final coalgebra $M(\vec{\alpha})$. Moreover, $W(\vec{\alpha})$ and $M(\vec{\alpha})$ are polynomial functors in $\vec{\alpha}$.

In short, polynomial functors are closed under composition, initial algebras, and final coalgebras, and so seem to have all the virtues of BNFs. Why not take them as the basis for a data type package?

The answer is that the class of polynomial functors is not as general as the class of BNFs. For example, although the type $\mathsf{finset}(\alpha)$ of finite sets of elements of α and the type $\mathsf{multiset}(\alpha)$ of finite multisets of elements of α are both BNFs, cardinality considerations can be used to show that they are not polynomial functors. For another view of what goes wrong, note that if we map the finite set $\{1, 2\}$ under a function which sends both 1 and 2 to 3, we get the set $\{3\}$, which does not have the same shape.

But we can view $\mathsf{finset}(\alpha)$ as a *quotient* of a polynomial functor, namely, the quotient of $\mathsf{list}(\alpha)$ that identifies any two lists that have the same elements. Similarly, we can view $\mathsf{multiset}(\alpha)$ as the quotient of $\mathsf{list}(\alpha)$ by equivalence up to permutation. This points the way to a solution: rather than consider only polynomial functors, we should consider their quotients as well.

3 Quotients of polynomial functors

A natural way to say that a functor $F(\alpha)$ is a quotient of the polynomial functor $P(\alpha)$ is to say that, for every α, there is a surjective function abs from $P(\alpha)$ to $F(\alpha)$. Think of elements of $P(\alpha)$ as being concrete representations of more abstract objects in $F(\alpha)$. We can express the fact that abs is surjective by supplying a right inverse, repr, which maps any element of $F(\alpha)$ to some representative in $P(\alpha)$. Finally, we should assert that abs is a natural transformation between P and F, which is to say, it respects their functorial behavior:

$$\begin{array}{ccc} P(\alpha) & \xrightarrow{P(f)} & P(\beta) \\ {\scriptstyle \mathsf{abs}_\alpha}\downarrow & & \downarrow{\scriptstyle \mathsf{abs}_\beta} \\ F(\alpha) & \xrightarrow{F(f)} & F(\beta) \end{array}$$

Remember that given $f : \alpha \to \beta$, there is a map $P(f)$ from $P(\alpha)$ to $P(\beta)$. We require this map to be preserved by abs, so that $\mathsf{abs}_\beta \circ P(f) = F(f) \circ \mathsf{abs}_\alpha$. In other words, mapping a representation x and then abstracting it should yield the same result as abstracting it and then mapping it, making the square above commute.

In Lean, we can specify that F is a quotient of a polynomial functor as follows:

```
class qpf (F : Type u → Type u) [functor F] :=
(P         : pfunctor.{u})
(abs       : Π {α}, P.apply α → F α)
(repr      : Π {α}, F α → P.apply α)
(abs_repr  : ∀ {α} (x : F α), abs (repr x) = x)
(abs_map   : ∀ {α β} (f : α → β) (p : P.apply α), abs (f <$> p) = f <$> abs p)
```

One can show that every BNF can be represented in this way. (Briefly, if κ is the relevant cardinal bound, we can take the shapes in the polynomial functor to be the set of pairs of the form $(I, F(I))$ with $I \subseteq \kappa$, and the contents of such a shape to be indexed by I.)

To see that every QPF has an initial algebra, suppose that F is a quotient of P in the sense above. Let W be the initial P-algebra. We want to construct the least fixed point fix of F. By initiality, there is an isomorphism between W and $P(W)$, so every tree in W can be viewed as consisting of a shape, $a \in A$, and a sequence $f : B(a) \to W$ of subtrees. Via the abs function, these represent an element of $F(W)$. The problem is that multiple elements of $P(W)$ can represent the same element of $F(W)$, so that multiple elements of W can represent the same element of $F(W)$. So already W looks like an overapproximation to fix. Moreover, recursively, under the functorial map for F, $F(W)$ is again an overapproximation to $F(\mathsf{fix})$.

The solution is to say what it means for two elements of W to represent the same element of fix, and then define fix to be the quotient of W by that equivalence relation. The relation we are after is the smallest equivalence relation closed under the following two rules:

$$\frac{\mathsf{abs}(a, f) = \mathsf{abs}(a', f')}{\langle a, f\rangle \equiv \langle a', f'\rangle} \qquad \frac{\forall x\ (f(x) \equiv f'(x))}{\langle a, f\rangle \equiv \langle a, f'\rangle}$$

The key condition is the first one, which says that two trees represented by (a, f) and (a', f') are equivalent if their abstractions are the same element of $F(W)$. The second clause extends the relation recursively to trees with the same shape and equivalent subtrees. The relation is defined inductively in Lean as follows:

```
inductive Wequiv : q.P.W → q.P.W → Prop
| abs (a : q.P.A) (f : q.P.B a → q.P.W) (a' : q.P.A) (f' : q.P.B a' → q.P.W) :
    abs ⟨a, f⟩ = abs ⟨a', f'⟩ → Wequiv ⟨a, f⟩ ⟨a', f'⟩
| ind (a : q.P.A) (f f' : q.P.B a → q.P.W) :
    (∀ x, Wequiv (f x) (f' x)) → Wequiv ⟨a, f⟩ ⟨a, f'⟩
| trans (u v w : q.P.W) : Wequiv u v → Wequiv v w → Wequiv u w
```

Notationally, here `q` is the relevant QPF structure, `q.P` is the polynomial functor, and `q.P.W` denotes the associated `W` construction, a function of `q.P`. The third clause ensures that the relation is transitive, and hence an equivalence relation. We then define fix to be the quotient:

```
def fix (F : Type u → Type u) [functor F] [q : qpf F] :=
quotient (Wsetoid : setoid q.P.W)
```

`Wsetoid` bundles `Wequiv` with a proof that the latter is an equivalence relation. Any function $g : F(W) \to \beta$ gives rise to a function $g' : P(W) \to \beta$ defined by $g' = g \circ \mathsf{repr}$, and so we can use g to define functions by recursion on W:

```
def recF {α : Type u} (g : F α → α) : q.P.W → α
| ⟨a, f⟩ := g (abs ⟨a, λ x, recF (f x)⟩)
```

This is just an ordinary recursion on W, using repr to mediate the difference between P and F. Moreover, any function defined by such a recursion will respect the equivalence relation `Wequiv`, and so lifts to a function from fix to α.

```
def fix.rec {α : Type u} (g : F α → α) : fix F → α :=
quot.lift (recF g) (recF_eq_of_Wequiv g)
```

We can map any tree W to a canonical representative, in such a way that any two equivalent trees are mapped to the same representative. This gives us a choice-free way of mapping fix back to W. Composing maps $F(\mathsf{fix}) \to P(\mathsf{fix}) \to P(W) \to W \to \mathsf{fix}$ gives us the desired constructor. With these definitions, we can prove:

```
theorem fix.rec_eq {α : Type u} (g : F α → α) (x : F (fix F)) :
  fix.rec g (fix.mk x) = g (fix.rec g <$> x)

theorem fix.ind_rec {α : Type u} (g g' : fix F → α)
    (h : ∀ x : F (fix F), g <$> x = g' <$> x → g (fix.mk x) = g' (fix.mk x)) :
  ∀ x, g x = g' x

theorem fix.ind (p : fix F → Prop)
    (h : ∀ x : F (fix F), liftp p x → p (fix.mk x)) :
  ∀ x, p x
```

The last theorem above expresses the induction principle, defined in terms of a predicate lifting operation defined in Section 5. The second-to-last theorem implies the uniqueness of functions satisfying the defining equations for the recursor.

With the recursor, we can then define a destructor from fix to $F(\mathsf{fix})$ and prove that it is an inverse to the constructor. This completes the construction of the initial algebra for any unary quotient of polynomial functors.

We can analogously construct the greatest fixed point of $F(\alpha)$ as a suitable quotient of M_P. Remember that the M-type analogue of the principle of induction on a W type is the bisimulation principle, `M_bisim`, presented at the end of the last section. The corresponding version for the final algebra should look like this:

```
theorem cofix.bisim (r : cofix F → cofix F → Prop)
    (h : ∀ x y, r x y → liftr r (cofix.dest x) (cofix.dest y) :
  ∀ x y, r x y → x = y
```

The function liftr in the statement of the principle refers to the canonical method of lifting a binary relation r on α to a relation $\hat{r}$ on $F(\alpha)$, described in Section 5. One strategy of constructing the final coalgebra cofix, familiar from the literature on set-valued functors (e.g. [37]), is to define the relation R to be the union of all bisimulation relations on the underlying M type, and then define cofix to be the the quotient M/R. For that proof to go through, we need to know that the union of all bisimulation relations is an equivalence relation, which in turn requires showing that the composition of bisimulation relations is again a bisimulation. And *that* can be shown as a consequence of the fact that the function $F(\alpha)$ preserves weak pullbacks. This explains why this assumption appears in Isabelle's definition of BNFs.

Going back to an early paper by Aczel and Mendler [4], however, we were able to find a construction that avoids the additional assumption. The trick is to use an alternative notion of lift for binary relations on a set α. Given a binary relation r on α, let q_r be the quotient map corresponding to the least equivalence relation on α that includes r. We can then define the alternative notion of lift which holds of x and y in $F(\alpha)$ if and only if $F(q_r)(x) = F(q_r)(y)$. In other words, rather than lift the relation, we map the quotient function. We now define two elements of M to bear this lifted version of r if their F-abstractions do, and define cofix to be the quotient under the union of all such relations.

```
def is_precongr (r : q.P.M → q.P.M → Prop) : Prop :=
∀ {x y}, r x y → abs (quot.mk r <$> M_dest x) = abs (quot.mk r <$> M_dest y)

def Mcongr : q.P.M → q.P.M → Prop := λ x y, ∃ r, is_precongr r ∧ r x y

def cofix (F : Type u → Type u) [functor F] [q : qpf F]:= quot (@Mcongr F _ q)
```

It is especially convenient that Lean's fundamental quotient construction, `quot.mk r`, does not require `r` to be an equivalence relation. (The axioms governing quotients in Lean imply that the result is equivalent to quotienting by the equivalence relation generated by r.) We can show that quotient by a finer relation factors through the quotient by a coarser one:

```
def factor {α : Type*} (r s : α → α → Prop) (h : ∀ x y, r x y → s x y) :
  quot r → quot s :=
quot.lift (quot.mk s) (λ x y rxy, quot.sound (h x y rxy))

def factor_mk_eq {α : Type*} (r s: α → α → Prop) (h : ∀ x y, r x y → s x y) :
  factor r s h ∘ quot.mk r = quot.mk s := rfl
```

With this fact, we can use the bisimulation principle on M to derive the bisimulation principle on the quotient. When the dust settles, we have all the desired functions and properties:

```
def cofix.dest : cofix F → F (cofix F)

def cofix.corec {α : Type u} (g : α → F α) : α → cofix F

theorem cofix.dest_corec {α : Type u} (g : α → F α) (x : α) :
  cofix.dest (cofix.corec g x) = cofix.corec g <$> g x

theorem cofix.bisim_rel (r : cofix F → cofix F → Prop)
    (h : ∀ x y, r x y →
      quot.mk r <$> cofix.dest x = quot.mk r <$> cofix.dest y) :
  ∀ x y, r x y → x = y
```

Since identity under the mapped quotients of r is implied by the lift of r, this formulation of the bisimulation principle implies cofix.bisim above, as well as the following variation:

```
theorem cofix.bisim' {α : Type u} (q : α → Prop) (u v : α → cofix F)
    (h : ∀ x, q x → ∃ a f f',
      cofix.dest (u x) = abs ⟨a, f⟩ ∧
      cofix.dest (v x) = abs ⟨a, f'⟩ ∧
      ∀ i, ∃ x', q x' ∧ f i = u x' ∧ f' i = v x') :
  ∀ x, q x → u x = v x
```

It is, moreover, straightforward to show that quotients of polynomial functors are closed under composition and quotients:

```
def comp {G : Type u → Type u} [functor G] [qpf G]
         {F : Type u → Type u} [functor F] [qpf F] :
  qpf (functor.comp G F)

def quotient_qpf {F : Type u → Type u} [functor F] [qpf F]
                 {G : Type u → Type u} [functor G]
    {abs  : Π {α}, F α → G α}
    {repr : Π {α}, G α → F α}
    (abs_repr : Π {α} (x : G α), abs (repr x) = x)
    (abs_map  : ∀ {α β} (f : α → β) (x : F α abs (f <$> x) = f <$> abs x) :
  qpf G
```

In short, we have shown that unary QPFs support the same constructions as unary BNFs. We now turn to the multivariate case.

4 Multivariate constructions

A ternary functor on $F(\alpha, \beta, \gamma)$ on sets is one that is functorial in each argument. Our goal is to extend the notion of a QPF to such functors, and, indeed, functors of arbitrary arity. In this respect, dependent type theory offers a distinct advantage over simple type theory: whereas Isabelle's BNF package has to synthesize definitions of n-ary functors dynamically for each n, in dependent type theory we can treat an n-tuple of types as a first-class object parameterized by n. This facilitates the implementation of a data type package, as discussed in Section 6.

Formally, we define an n-tuple of types to be a function from a canonical finite type $\mathsf{fin}(n)$ of elements to an arbitrary type universe:

```
def typevec (n : ℕ) := fin n → Type*
```

We can then define the usual morphisms on the category of n-tuples, namely, n-tuples of functions, with composition and identity.

```
def arrow (α β : typevec n) := Π i : fin n, α i → β i

infixl ` ⟹ `:40 := arrow

def id {α : typevec n} : α ⟹ β := λ i x, x

def comp {α β γ : typevec n} (g : β ⟹ γ) (f : α ⟹ β) : α ⟹ γ :=
λ i x, g i (f i x)

infixr ` ⊙ `:80 := typevec.comp
```

Lean's notions of *functor* (a type constructor with a map function) and *lawful functor* (a functor satisfying the usual laws) carry over straightforwardly to the multivariate setting:

```
class mvfunctor {n : ℕ} (F : typevec n → Type*) :=
(map : Π {α β : typevec n}, (α ⟹ β) → (F α → F β))

infixr ` <$$> `:100 := mvfunctor.map

class is_lawful_mvfunctor {n : ℕ} (F : typevec n → Type*) [mvfunctor F] :=
(id_map   : Π {α : typevec n} (x : F α), id <$$> x = x)
(comp_map : Π {α β γ : typevec n} (g : α ⟹ β) (h : β ⟹ γ) (x : F α),
                (h ⊙ g) <$$> x = h <$$> g <$$> x)
```

Notice that we use the notation `f <$$> x` to denote the functorial map of the n-tuple of functions f on the element x, where x is an element of the multivariate $F(\vec{\alpha})$. With these definitions and notation, the definition of a multivariate QPF is almost exactly the same as the definition of a unary one:

```
class mvqpf {n : ℕ} (F : typevec.{u} n → Type*) [mvfunctor F] :=
(P         : mvpfunctor.{u} n)
(abs       : Π {α}, P.apply α → F α)
(repr      : Π {α}, F α → P.apply α)
(abs_repr  : ∀ {α} (x : F α), abs (repr x) = x)
(abs_map   : ∀ {α β} (f : α ⟹ β) (p : P.apply α),
               abs (f <$$> p) = f <$$> abs p)
```

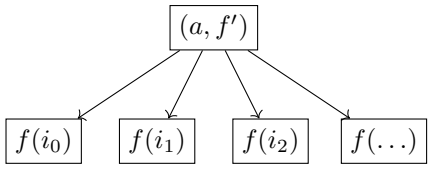
We need to show that if $F(\vec{\alpha}, \beta)$ is an $(n+1)$-ary QPF, then for each tuple $\vec{\alpha}$ it has both an initial algebra $\mathsf{fix}(\vec{\alpha})$ and a final coalgebra $\mathsf{cofix}(\vec{\alpha})$, and, moreover, that the latter are n-ary functors in $\vec{\alpha}$. The constructions require operations $\mathsf{append1}(\vec{\alpha}, \beta)$ for extending an n-tuple of types $\vec{\alpha}$ by a single type β, and operations $\mathsf{drop}(\vec{\alpha})$ and $\mathsf{last}(\vec{\alpha})$ that return the initial n-tuple and final elements of such an $(n+1)$-tuple. Similarly, we need an operation $\mathsf{append\text{-}fun}(f, g)$ that appends a function to an n-tuple of functions, and operations $\mathsf{drop\text{-}fun}$ and $\mathsf{last\text{-}fun}$ that destruct the resulting $(n+1)$-tuple. One minor problem is that constructions like these sometimes give rise to types that are provably but not definitionally equal. For example, $\mathsf{append1}(\mathsf{drop}(\vec{\alpha}), \mathsf{last}(\alpha))$ is provably equal to $\vec{\alpha}$, but we need an explicit cast from one to the other if we want expressions to type check. Such difficulties were mild, and they were a small price to pay for the benefits of being able to reason about arbitrary tuples uniformly.

With a formal theory of tuples of types and maps between them, unary notions carry over nicely to the multivariate setting. The definition of a multivariate polynomial functor $P(\vec{\alpha})$ is straightforward:

```
structure mvpfunctor (n : ℕ) := (A : Type.{u}) (B : A → typevec.{u} n)

variables {n : ℕ} (P : mvpfunctor.{u} n)

def apply (α : typevec.{u} n) : Type u := Σ a : P.A, P.B a ⟹ α

def map {α β : typevec n} (f : α ⟹ β) : P.apply α → P.apply β :=
λ ⟨a, g⟩, ⟨a, f ⊙ g⟩
```

As before, we can think of an element of $P(\vec{\alpha})$ as consisting of a shape, a, and a map $f : B(a) \Longrightarrow \vec{\alpha}$. All that has changed is that the contents now consist of tuples of functions.

Given an $(n+1)$-ary polynomial functor $P(\vec{\alpha}, \beta)$, we need to construct its initial algebra, $W(\vec{\alpha})$, and show that it is again a polynomial functor. Intuitively, each element of $W(\vec{\alpha})$ is a well-founded tree, in which each node is labeled by an element of A together with a function $f' : \mathsf{drop}(B(a)) \Longrightarrow \vec{\alpha}$, and the children of that node are given by a function $f : \mathsf{last}(B(a)) \to W(\vec{\alpha})$:



There are various ways to view such a tree. One is as an ordinary unary W type, where the set of shapes at each node is given by $A' = \Sigma_{a \in A}(\mathsf{drop}(B(a)) \to \vec{\alpha}$. Given an element $p = (a, f')$ of A', the set of indices $B'(p) = \mathsf{last}(B(p.\mathsf{fst}))$ depends only on the first component. This, however, introduces an artificial dependency of the index set of the branches on the contents f'. A slightly modified description is to view the $W(\vec{\alpha})$ as given inductively by the following constructor and recursion principle:

```
def W_mk {α : typevec n} (a : P.A) (f' : P.drop.B a ⟹ α)
  (f : P.last.B a → P.W α) : P.W α

def W_rec {α : typevec n} {C : Type*}
  (g : Π a : P.A, ((P.drop).B a ⟹ α) → ((P.last).B a → P.W α) →
    ((P.last).B a → C) → C) : P.W α → C
```

In words, an element of $W(\vec{\alpha})$ is given inductively by a triple (a, f', f) where a is in A, f' is a tuple of functions from $\mathsf{drop}(B(a))$ to $\vec{\alpha}$, and f' is a function from $\mathsf{last}(B(a))$ to $W(\vec{\alpha})$. The induction principle and defining equations for the recursor are as expected.

An alternative view of $W(\vec{\alpha})$ makes it clear that it is a polynomial functor. As the picture suggests, we can think of an element of $W(\vec{\alpha})$ as having the shape of a well-founded tree with labels from A, with children at a node labeled a indexed by the set $\mathsf{last}(B(a))$. In other words, the shape is just the ordinary W type given by these data. The *contents* of the tree amount to the sum total of all the functions f' at each node. We can combine these into one big function from the disjoint union of all the index sets $\mathsf{drop}(B(a))$ at all the nodes. This disjoint union can be conveniently described by an inductive definition:

```
inductive W_path : P.last.W → fin n → Type u
| root (a : P.A) (f : P.last.B a → P.last.W) (i : fin n) (c : P.drop.B a i) :
    W_path ⟨a, f⟩ i
| child (a : P.A) (f : P.last.B a → P.last.W) (i : fin n) (j : P.last.B a)
    (c : W_path (f j) i) : W_path ⟨a, f⟩ i
```

Here, `P.last` denotes the polynomial functor just described, and `P.last.W` is the corresponding W type, which we take to be the shape of $W(\vec{\alpha})$. The type `W_path` describes the index set associated to this shape as the sum of the index sets `P.drop.B a i` for each $i < n$, together with the index sets assigned to all the children. This gives us the desired representation of $W(\vec{\alpha})$ as a multivariate polynomial functor:

```
def Wp : mvpfunctor n := { A := P.last.W, B := P.W_path }
```

There are two things worth noting here. First, the type of `W_path` is equivalent to `P.last.W` $\implies$ `typevec n`. This makes use of the specific representation of `typevec n`, but this use is not essential; with another representation, we could still define `W_path` as above, and then compose it with the relevant isomorphism. The second thing to note is that the analogous inductive definition works just as well for M types, since it does not require the tree to be well founded.

Both characterizations of $W(\vec{\alpha})$ are essential. The first description, the inductive one, allows us to carry out the construction of the initial algebra. The second description of $W(\vec{\alpha})$, as a polynomial functor, enables us to show that the initial algebra of a QPF is again a QPF. Rather than define both objects and prove them isomorphic, we found it more convenient to take the second description to be the official definition of $W(\vec{\alpha})$ and use that to define the constructor and recursor specified by the first description.

Coordinating the different notions of a polynomial functor was the most difficult part of extending the constructions from the unary to the multivariate setting. With these characterizations of $W(\vec{\alpha})$, the construction of the initial algebra $\mathsf{fix}(\vec{\alpha})$ of a multivariate QPF $F(\vec{\alpha}, \beta)$ is almost line-by-line the same as the construction in the unary case, replacing unary primitives with their multivariate counterparts. Suppose $F(\vec{\alpha}, \beta)$ is a quotient of the polynomial functor of $P(\vec{\alpha}, \beta)$. The associated $W(\vec{\alpha})$ is again a polynomial functor, and $\mathsf{fix}(\vec{\alpha})$ is defined as a quotient of that. It is not hard to define the map on $\mathsf{fix}(\vec{\alpha})$ in terms of the map on $W(\vec{\alpha})$, and then use the QPF property of F to show that the maps commute with the abstraction function from $W(\vec{\alpha})$ to $\mathsf{fix}(\vec{\alpha})$. In short, we have that if $F(\vec{\alpha}, \beta)$ is a multivariate QPF, then so is $\mathsf{fix}(\vec{\alpha})$.

The construction of the final coalgebra $\mathsf{cofix}(\vec{\alpha})$ is similar: the approach above can be used to construct the M types $M(\vec{\alpha})$ as polynomial functors, and, once again, the unary construction carries over. Showing that multivariate QPFs are closed under compositions and quotients is once again straightforward.

5 Lifting predicates and relations

Let F be any set-valued functor. By definition, F allows us to map any function $f : \alpha \to \beta$ to a function from $F(\alpha) \to F(\beta)$, enabling us to reason about the behavior of f under F. For instance, $\mathsf{list}(f)$ applies f to every element of a list, and $\mathsf{finset}(f)$ maps any finite set s to $f[s]$, the image of s under f.

Sometimes it is useful to reason about the behavior of predicates and relations as well. A standard way of doing that is to consider their *lifts* [33, 36, 38], defined as follows. Let p be any predicate on α. Then there is an inclusion map $\iota : \{u \in \alpha \mid p(u)\} \to \alpha$, and saying that $p(u)$ holds is equivalent to saying that u is in the image of ι. To lift p to $F(\alpha)$, consider the map $F(\iota) : F(\{u \mid p(u)\}) \to F(\alpha)$, and given any element x of $F(\alpha)$, say that the lift $\hat{p}$ holds of x if and only if there is an element z of $F(\{u \mid p(u)\})$ such that $F(\iota)(z) = x$.

Similarly, if $r(u, v)$ is a binary relation between α and β, we can lift r to a relation $\hat{r}$ between $F(\alpha)$ and $F(\beta)$ as follows. Consider the set $\{(u, v) \in \alpha \times \beta \mid r(u, v)\}$ of pairs, and the two projections π_0 and π_1. Given x in $F(\alpha)$ and y in $F(\beta)$, say that $\hat{r}(x, y)$ holds if there is an element z of $F(\{(u, v) \mid r(u, v)\})$ such that $F(\pi_0)(z) = x$ and $F(\pi_1)(z) = y$.

It is straightforward to define these notions in the type-theoretic setting, with types and subtypes in place of sets and subsets.

```
def liftp {α : Type u} (p : α → Prop) : F α → Prop :=
λ x, ∃ z : F (subtype p), subtype.val <$> z = x

def liftr {α : Type u} (r : α → β → Prop) : F α → F β → Prop :=
λ x y, ∃ z : F {p : α × β // r p.fst p.snd},
  (λ t, t.val.fst) <$> z = x ∧ (λ t, t.val.snd) <$> z = y
```

If P is a polynomial functor, p is a predicate on α, and x is in $P(\alpha)$, it is easy to check that $\hat{p}(x)$ holds if and only if x is of the form (a, f) and for every $i \in B(a)$, $p(f(i))$. Similarly, for a relation r between α and β, x in $P(\alpha)$, and $y \in P(\beta)$, $\hat{r}(x, y)$ if and only if there are a, f, and f' such that x is of the form (a, f), y is of the form (a, f') and for every i, $r(f(i), f'(i))$. In words, $\hat{r}(x, y)$ holds if x and y have the same shape and their contents are pointwise related. If F is a *quotient* of a polynomial functor, the statements are the same up to a choice of representative.

▶ **Theorem 1.** *Let F be a QPF, let p be a predicate on α, and r be a binary relation between α and β.*

- *$\hat{p}(x)$ holds if and only if there are a and f such that $x = \mathsf{abs}(a, f)$ and, for every i, $p(f(i))$.*
- *$\hat{r}(x, y)$ holds if and only if there are a, f, f' such that $x = \mathsf{abs}(a, f)$, $y = \mathsf{abs}(a, f')$, and for every i, $r(f(i), f'(i))$.*

Lifting extends straightforwardly to multivariate QPFs: if $F(\vec{\alpha})$ is an n-ary QPF, we can lift n-ary tuples of predicates and n-ary tuples of relations analogously, and the corresponding version of Theorem 1 holds.

We can use these notions to clarify the additional structure that comes with the Isabelle formulation of a BNF. If F is a QPF and x is an element of $F(\alpha)$, intuitively, $\hat{p}(x)$ says that p holds of the contents of x. When F is a polynomial functor, this is literally true, but the possibility of multiple representations in a QPF muddies the waters. We would like to have a function $\mathsf{supp}(x)$, the "support" of x, such that for every predicate p on α and x in $F(\alpha)$, we have $\hat{p}(x) \leftrightarrow \forall u \in \mathsf{supp}(x)\ p(u)$. Call this condition $(*)$.

▶ **Theorem 2.** *Let F be any set-valued functor. If* supp *satisfies* $(*)$*, then for any* $x \in F(\alpha)$ *we have* $\mathsf{supp}(x) = \{u \mid \forall p\ (\hat{p}(x) \to p(u))\} = \bigcap\{\beta \mid \beta \subseteq \alpha \wedge x \in \mathsf{lm}\ F(\iota_{\beta\to\alpha})\}$*, where* $\iota_{\beta\to\alpha}$ *is the inclusion map from* β *to* α*.*

Proof. We check the first equation, and leave it to the reader to verify the second. Suppose $u \in \mathsf{supp}(x)$ and $\hat{p}(x)$. Then $(*)$ implies that $p(u)$ holds. For the converse, note that taking $p(u)$ to be $u \in \mathsf{supp}(x)$ in $(*)$, it is immediate that $\hat{p}(x)$ holds. So any u satisfying $\forall p\ (\hat{p}(x) \to p(u))$ is an element of $\mathsf{supp}(x)$. ◀

▶ **Theorem 3.** *Let F be a QPF, and let* $\mathsf{supp}(x) = \{u \mid \forall p\ (\hat{p}(x) \to p(u))\}$.

- *For every* x, $\mathsf{supp}(x) = \bigcap\{\mathsf{lm}\ f \mid \mathsf{abs}(a, f) = x\}$.
- *Condition* $(*)$ *holds at* x *for every* p *if and only if there are* a, f *such that* $\mathsf{abs}(a, f) = x$ *and for every* a', f' *such that* $\mathsf{abs}(a', f') = x$, $\mathsf{lm}\ f \subseteq \mathsf{lm}\ f'$.

Proof. For the first clause, let x be arbitrary, and suppose $\hat{p}(x)$ implies $p(u)$ for every p. If $x = \mathsf{abs}(a, f)$, let p be the predicate $u \in \mathsf{lm}\ f$. Then it is easy to check that $\hat{p}(x)$ holds, and hence $p(u)$. Conversely, suppose u is an element of the right-hand side and p is a predicate such that $\hat{p}(x)$ holds. Then there are a and f such that $\mathsf{abs}(a, f) = x$ and such that $p(f(i))$ holds for every i. Hence $p(u)$.

For the forward direction of the second clause, note that if $p(u)$ is the predicate $u \in \mathsf{supp}(x)$, then, by $(*)$, we have $\hat{p}(x)$. The conclusion follows from Theorem 2 and the first clause. Using the first clause, the converse direction of the second clause is also straightforward. ◀

Theorem 3 says that condition $(*)$ holds for a QPF F if and only if every element x of $F(\alpha)$ has a representation (a, f) whose contents are minimal, and these contents determine which lifted predicates hold. Unfortunately, there is nothing in the definition of a QPF that rules out representations having superfluous elements, but the next theorem shows that adding this as an additional assumption has pleasant consequences.

▶ **Theorem 4.** *Let F be a QPF satisfing the additional property that for every a, f, a', and f', if* $\mathsf{abs}(a, f) = \mathsf{abs}(a', f')$*, then* $\mathsf{lm}\ f = \mathsf{lm}\ f'$*. Then:*

- supp *satisfies* $(*)$*, and whenever* $x = \mathsf{abs}(a, f)$, $\mathsf{supp}(x) = \mathsf{lm}\ f$.
- *For every* x *in* $F(\alpha)$ *and* $g : \alpha \to \beta$, $\mathsf{supp}(F(g)(x)) = g[\mathsf{supp}(x)]$.

In other words, with the additional assumption, our function supp has the same properties as the function set associated to Isabelle's BNFs.

BNFs have one additional property, which can also conveniently be expressed in terms of lifts. If r is a relation between α and β and s is a relation between β and γ, the composition $r \circ s$ is defined by $(r \circ s)(u, w) \equiv \exists v\ (r(u, v) \wedge s(v, w))$. It is straightforward to show from the definition of lifting that for every x in $F(\alpha)$ and z in $F(\gamma)$, $\widehat{r \circ s}(x, z)$ implies $(\hat{r} \circ \hat{s})(x, z)$. But the converse does not necessarily hold, and the special case where F is a QPF gives an inkling of what can go wrong: the fact that there is a shape that relates x to y by $\hat{r}$ and another shape that relates y to z by $\hat{s}$ does not necessarily mean there is a single shape that does both, and hence relates x and z by $\widehat{r \circ s}$.

When the converse does hold for every r and s, F is said to *preserve weak pullbacks*. This is a useful property: it implies that the composition of bisimulation relations relative to F is again a bisimulation relation. There are, however, interesting examples of QPFs that do not preserve weak pullbacks, such as a bounded finite powerset, which for some fixed k returns the collection of finite subsets with at most k elements. For details and alternative characterizations of preservation of weak pullbacks, see [6, 33, 36, 38], and for more instances of QPFs that do not preserve them, see [4, Section 6] and [28, Section 6.4].

6 Implementation

We are currently writing a data type compiler for Lean that builds on the formal constructions just described. The compiler, which is implemented entirely in Lean's metaprogramming framework [22], introduces the keywords `data` and `codata` into Lean's normal syntax and translates each data type specification into a number of definitions. Whereas the Isabelle implementation has to construct n-ary instances of the constructions for each fixed n, the uniform theory of multivariate constructions described in Section 4 simplifies the expressions we need to construct, and therefore reduces the likelihood of failure at compile time.

The commands `data` and `codata`, respectively, declare the initial algebra and final coalgebra of a multivariate QPF $F(\vec{\alpha}, \beta)$. In our implementation, we refer to F as the *shape* of the declaration. The key insight is that both the functor and its representation as a QPF can be synthesized from the syntactic specification. Consider the following input:

```
data tree (α β : Type) : Type
| leaf : tree
| node : α → (β → tree) → tree
```

This describes the type of trees in which every internal node has a label from α and a sequence of children indexed by β. Since β occurs in a negative position, our compiler interprets that as a dead parameter. It then replaces `tree` with a parameter X and interprets the resulting shape as a binary functor $F_\beta(\alpha, X)$.

```
inductive tree.shape (α : Type) (β : Type) (X : Type) : Type
| nil : tree.shape
| cons : α → (β → X) → tree.shape

def tree.shape.internal (β : Type) : typevec 2 → Type
| ⟨α, X⟩ := shape α β X
```

Note that the internal version bundles α and X together into a vector of length 2. The next task is to synthesize a QPF instance. In general, the arguments to each constructor are compositions of QPFs, so the entire shape, a sum of products of QPFs, is again a QPF.

```
instance (β : Type) : mvfunctor (tree.shape.internal β) := ...
instance (β : Type) : mvqpf (tree.shape.internal β) := ...
```

We then use the generic QPF fix construction to define the initial fixed point.

```
def tree.internal (β : Type) (v : typevec 1) : Type :=
fix (list.shape.internal β) v

def tree (α β : Type) : Type := tree.internal β ⟦α⟧

instance (β : Type) : mvfunctor (tree.internal β) := ...
instance (β : Type) : mvqpf (tree.internal β) := ...
```

We can then define the constructors, destructors, recursor, and so on:

```
def tree.nil (α β : Type) : tree α β := ...

def tree.cons (α β : Type) (x : α) (xs : β → tree α β) : tree α β := ...

def tree.cases_on {α β : Type} {C : tree α β → Sort u_1} (n : tree α β) :
  C (tree.nil α β) →
```

```
    (Π (a : α) (a_1 : β → tree α β), C (tree.cons α β a a_1)) →
    C n := ...

def tree.rec {α β X : Type} : X → (α → (β → X) → X) → tree α β → X := ...
```

If we replace `data` by `codata`, we get the corresponding coinductive type. It has same constructors and destructors, but, instead, the following corecursor and bisimulation principle:

```
def tree'.corec :
  Π (α β α_1 : Type), (α_1 → shape α β α_1) → α_1 → tree' α β := ...

def tree'.bisim : ∀ (α β : Type) (r : tree' α β → tree' α β → Prop),
  (∀ (x y : tree' α β),
      r x y → mvfunctor.liftr (typevec.rel_last ⌈α⌉ r) (mvqpf.cofix.dest x)
    (mvqpf.cofix.dest y)) →
  ∀ (x y : tree' α β), r x y → x = y := ...
```

Our work on the compiler is still in progress: we do not yet handle nested data types in the specification of the shape or present lifted predicates and relations in a user-friendly way. We also intend to write an equation compiler to support more natural ways to define functions.

7 Conclusions and related work

We have shown that the representation of data types as quotients of polynomial functors is natural, and facilitates important data type constructions. Surprisingly, the bulk of our formalization deals with constructions that are intuitively straightforward, like the representations of multivariate W and M types as polynomial functors as described in Section 4. It is notable that, with this infrastructure, our constructions of the initial algebras and final coalgebras require only a few hundred lines of code.

Other theorem provers such as Coq [26] and Agda[1] support coinductive types and corecursion by extending the trusted kernel. Here we have followed Isabelle's approach by constructing such data types explictly, without extending the axiomatic framework. We made use of a quotient construction that is given axiomatically in Lean, though other libraries, including Isabelle's, take a definitional approach to quotients as well [17, 27, 29]. Tassi has recently developed methods for generating induction principles and other theorems to support the use of inductive types in Coq [40]. In a sense, this serves to recover some of the benefits of the more modular approaches given by BNFs and QPFs.

Abbot et al. [2] have considered quotients of polynomial functors by equivalence with respect to sets of permutations of the indices associated to each shape, and they have shown that these have nice computational properties. Such quotients are special cases of QPFs. Polynomial functors are closely related to *species* [30], but, as noted by Yorgey [41, Section 8], the precise relationship between polynomial functors and species is not yet well understood.

There are a number of ways that our work can be extended. Our constructions currently yield nondependent types and nondependent recursion and corecursion principles, so an obvious task is to work out and formalize the semantics of indexed inductive and coinductive data types. To that end, work by Altenkirch et al. [7] on indexed polynomial functors provides a good starting point. We are grateful to an anonymous referee for pointing out that there is nothing special about $\mathsf{fin}(n)$ in the definition of mvfunctor in Section 4, and so replacing

[1] `https://agda.readthedocs.io/en/latest/language/coinduction.html`

$\mathsf{fin}(n)$ by an arbitrary index type I is an easy first step towards handling families of types. The latter would also require generalizing our constructions to handle functors on categories other than the category of types (in this case, categories of indexed families). The paper by Blanchette et al. [16] shows how to achieve nonuniform forms of recursion and corecursion with BNFs, which can be seen as a step towards handling such dependencies. Dependent families would provide us with a shortcut to defining mutual inductive and coinductive definitions, currently handled by Isabelle's BNF package but not ours. Blanchette et al. [14] have shown that restricting morphisms to permutations can be used to model data types with binders.

We have not dealt with the computational interpretation of corecursion or code extraction at all. Even though most of our formalization is constructive, the defining equations for corecursion do not correspond to computational reductions in our underlying definitions. Firsov and Stump [23] show how to model inductive types in a computational type theory extending the calculus of constructions with implicit products, heterogeneous equality, and intersection types. It would be interesting to know whether coinductive types can be modeled in a similar way. Blanchette et al. [13] provide a nice overview of various approaches to computational intepretation of corecursion, and Basold and Geuvers [10, 11] provide a computational analysis of dependent versions of a type theory with both recursion and corecursion. We are hopeful that quotients of polynomial functors can provide insight into the semantics of such a system.

References

1 Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of Containers. In Andrew D. Gordon, editor, *Foundations of Software Science and Computational Structures (FOSSACS) 2003*, pages 23–38. Springer, 2003. doi:10.1007/3-540-36576-1_2.

2 Michael Gordon Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Constructing Polymorphic Programs with Quotient Types. In Dexter Kozen and Carron Shankland, editors, *Mathematics of Program Construction (MPC) 2004*, pages 2–15. Springer, 2004. doi:10.1007/978-3-540-27764-4_2.

3 Peter Aczel. *Non-well-founded sets*. Stanford University, Center for the Study of Language and Information, Stanford, CA, 1988.

4 Peter Aczel and Nax Mendler. A final coalgebra theorem. In *Category theory and computer science*, pages 357–365. Springer, Berlin, 1989. doi:10.1007/BFb0018361.

5 Jiri Adámek and H.-E. Porst. On tree coalgebras and coalgebra presentations. *Theoret. Comput. Sci.*, 311(1-3):257–283, 2004. doi:10.1016/S0304-3975(03)00378-5.

6 Jiří Adámek, Stefan Milius, and Lawrence S. Moss. Fixed points of functors. *J. Log. Algebr. Methods Program.*, 95:41–81, 2018. doi:10.1016/j.jlamp.2017.11.003.

7 Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. Indexed containers. *J. Funct. Programming*, 25:e5, 41, 2015. doi:10.1017/S095679681500009X.

8 Jeremy Avigad, Soonho Kong, and Leonardo de Moura. Theorem Proving in Lean. Online documentation. URL: https://leanprover.github.io/theorem_proving_in_lean/.

9 Michael Barr. Terminal coalgebras in well-founded set theory. *Theor. Comput. Sci.*, 114(2):299–315, 1993. doi:10.1016/0304-3975(93)90076-6.

10 Henning Basold. *Mixed Inductive-Coinductive Reasoning Types, Programs and Logic*. PhD thesis, Radboud Universiteit Nijmegen, 2018.

11 Henning Basold and Herman Geuvers. Type Theory based on Dependent Inductive and Coinductive Types. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Logic in Computer Science (LICS) 2016*, pages 327–336. ACM, 2016. doi:10.1145/2933575.2934514.

12 Julian Biendarra, Jasmin Christian Blanchette, Aymeric Bouzy, Martin Desharnais, Mathias Fleury, Johannes Hölzl, Ondrej Kuncar, Andreas Lochbihler, Fabian Meier, Lorenz Panny, Andrei Popescu, Christian Sternagel, René Thiemann, and Dmitriy Traytel. Foundational (Co)datatypes and (Co)recursion for Higher-Order Logic. In Clare Dixon and Marcelo Finger, editors, *Frontiers of Combining Systems (FroCoS) 2017*, pages 3–21. Springer, 2017. doi:10.1007/978-3-319-66167-4_1.

13 Jasmin Christian Blanchette, Aymeric Bouzy, Andreas Lochbihler, Andrei Popescu, and Dmitriy Traytel. Friends with Benefits: Implementing Corecursion in Foundational Proof Assistants. In Hongseok Yang, editor, *Programming Languages and Systems (ESOP) 2017*, pages 111–140. Springer, 2017. doi:10.1007/978-3-662-54434-1_5.

14 Jasmin Christian Blanchette, Lorenzo Gheri, Andrei Popescu, and Dmitriy Traytel. Bindings as bounded natural functors. *PACMPL*, 3(POPL):22:1–22:34, 2019. doi:10.1145/3290335.

15 Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Truly Modular (Co)datatypes for Isabelle/HOL. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving (ITP) 2014*, pages 93–110. Springer, 2014. doi:10.1007/978-3-319-08970-6_7.

16 Jasmin Christian Blanchette, Fabian Meier, Andrei Popescu, and Dmitriy Traytel. Foundational nonuniform (Co)datatypes for higher-order logic. In *Logic in Computer Science (LICS) 2017*, pages 1–12. IEEE Computer Society, 2017. doi:10.1109/LICS.2017.8005071.

17 Cyril Cohen. Pragmatic Quotient Types in Coq. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving (ITP) 2013*, pages 213–228. Springer, 2013. doi:10.1007/978-3-642-39634-2_17.

18 Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, Todd B. Knoblock, N. P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice Hall, 1986. URL: http://dl.acm.org/citation.cfm?id=10510.

19 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, 2015. doi:10.1007/978-3-319-21401-6_26.

20 Richard Dedekind. *Was sind und was sollen die Zahlen?* Vieweg, Braunschweig, 1888. Translated by Wooster Beman as "The nature and meaning of numbers" in *Essays on the theory of numbers*, Open Court, Chicago, 1901; reprinted by Dover, New York, 1963.

21 Peter Dybjer. Inductive Families. *Formal Asp. Comput.*, 6(4):440–465, 1994. doi:10.1007/BF01211308.

22 Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *PACMPL*, 1(ICFP):34:1–34:29, 2017. doi:10.1145/3110278.

23 Denis Firsov and Aaron Stump. Generic derivation of induction for impredicative encodings in Cedille. In June Andronick and Amy P. Felty, editors, *Certified Programs and Proofs (CPP) 2018*, pages 215–227. ACM, 2018. doi:10.1145/3167087.

24 Gottlob Frege. *Grundgesetze der Arithmetik*. H. Pohle, Jena, Volume 1, 1893, Volume 2, 1903.

25 Nicola Gambino and Martin Hyland. Wellfounded Trees and Dependent Polynomial Functors. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs (TYPES) 2003*, pages 210–225. Springer, 2003. doi:10.1007/978-3-540-24849-1_14.

26 Eduardo Giménez. Codifying Guarded Definitions with Recursive Schemes. In Peter Dybjer, Bengt Nordström, and Jan M. Smith, editors, *Types for Proofs and Programs (TYPES) 1994*, pages 39–59. Springer, 1994. doi:10.1007/3-540-60579-7_3.

27 Martin Hofmann. A Simple Model for Quotient Types. In Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin, editors, *Typed Lambda Calculi and Applications (TLCA) 1995*, pages 216–234. Springer, 1995. doi:10.1007/BFb0014055.

28 Johannes Hölzl, Andreas Lochbihler, and Dmitriy Traytel. A Formalized Hierarchy of Probabilistic System Types (Proof Pearl). In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving (ITP) 2015*, pages 203–220. Springer, 2015. doi:10.1007/978-3-319-22102-1_13.

29 Peter V. Homeier. A Design Structure for Higher Order Quotients. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics (TPHOLs) 2005*, pages 130–146. Springer, 2005. doi:10.1007/11541868_9.

30 André Joyal. Une théorie combinatoire des séries formelles. *Adv. in Math.*, 42(1):1–82, 1981. doi:10.1016/0001-8708(81)90052-9.

31 Bronisław Knaster. Un théorème sur les fonctions d'ensembles. *Ann. Soc. Polon. Math.*, 6:133–134, 1928.

32 Georg Kreisel. Generalized Inductive Definitions. Stanford Report on the Foundations of Analysis (mimeographed), CH. III, Stanford, 1963.

33 Alexander Kurz and Jiri Velebil. Relation lifting, a survey. *J. Log. Algebr. Meth. Program.*, 85(4):475–499, 2016. doi:10.1016/j.jlamp.2015.08.002.

34 Per Martin-Löf. An intuitionistic theory of types: predicative part. In *Logic Colloquium 1973*, pages 73–118. North-Holland, Amsterdam, 1975.

35 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL. A proof assistant for higher-order logic*. Springer Verlag, Berlin, 2002.

36 Jan J. M. M. Rutten. Relators and Metric Bisimulations. *Electr. Notes Theor. Comput. Sci.*, 11:252–258, 1998. doi:10.1016/S1571-0661(04)00063-5.

37 Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoret. Comput. Sci.*, 249(1):3–80, 2000. doi:10.1016/S0304-3975(00)00056-6.

38 Sam Staton. Relating Coalgebraic Notions of Bisimulation. In Alexander Kurz, Marina Lenisa, and Andrzej Tarlecki, editors, *Algebra and Coalgebra in Computer Science (CALCO) 2009*, pages 191–205. Springer, 2009. doi:10.1007/978-3-642-03741-2_14.

39 Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955. URL: http://projecteuclid.org/euclid.pjm/1103044538.

40 Enrico Tassi. Deriving proved equality tests in Coq-elpi: Stronger induction principles for containers in Coq. This *Proceedings*, 2019.

41 Brent A. Yorgey. Species and functors and types, oh my! In Jeremy Gibbons, editor, *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell*, pages 147–158. ACM, 2010. doi:10.1145/1863523.1863542.

Primitive Floats in Coq

Guillaume Bertholon
École Normale Supérieure, Paris, France
guillaume.bertholon@ens.fr

Érik Martin-Dorel
Lab. IRIT, University of Toulouse, CNRS, France
https://www.irit.fr/~Erik.Martin-Dorel/
erik.martin-dorel@irit.fr

Pierre Roux
ONERA, Toulouse, France
https://www.onera.fr/staff/pierre-roux
pierre.roux@onera.fr

Abstract

Some mathematical proofs involve intensive computations, for instance: the four-color theorem, Hales' theorem on sphere packing (formerly known as the Kepler conjecture) or interval arithmetic. For numerical computations, floating-point arithmetic enjoys widespread usage thanks to its efficiency, despite the introduction of rounding errors.

Formal guarantees can be obtained on floating-point algorithms based on the IEEE 754 standard, which precisely specifies floating-point arithmetic and its rounding modes, and a proof assistant such as Coq, that enjoys efficient computation capabilities. Coq offers machine integers, however floating-point arithmetic still needed to be emulated using these integers.

A modified version of Coq is presented that enables using the machine floating-point operators. The main obstacles to such an implementation and its soundness are discussed. Benchmarks show potential performance gains of two orders of magnitude.

2012 ACM Subject Classification Theory of computation → Type theory; Mathematics of computing → Numerical analysis; General and reference → Performance

Keywords and phrases Coq formal proofs, floating-point arithmetic, reflexive tactics, Cholesky decomposition

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.7

Supplement Material https://github.com/coq/coq/pull/9867

Acknowledgements The authors would like to thank Maxime Dénès and Guillaume Melquiond for helpful discussions.

1 Motivation

The proof of some mathematical facts can involve a numerical computation in such a way that trusting the proof requires trusting the numerical computation itself. Thus, being able to efficiently perform this kind of proofs inside a proof assistant eventually means that the tool must offer efficient numerical computation capabilities.

Floating-point arithmetic is widely used in particular for its efficiency thanks to its hardware implementation. Although it does not generally give exact results, introducing rounding errors, rigorous proofs can still be obtained by bounding the accumulated errors. There is thus a clear interest in providing an efficient and sound access to the processor floating-point operators inside a proof assistant such as Coq.

$R := 0$;
for j **from** 1 **to** n **do**
  for i **from** 1 **to** $j-1$ **do**
    $R_{i,j} := \left(A_{i,j} - \Sigma_{k=1}^{i-1} R_{k,i} R_{k,j}\right) / R_{i,i}$;
  end for
  $R_{j,j} := \sqrt{M_{j,j} - \Sigma_{k=1}^{j-1} {R_{k,j}}^2}$;
end for

Figure 1 Cholesky decomposition: given $A \in \mathbb{R}^{n \times n}$, attempts to compute R such that $A = R^T R$.

1.1 Proofs Involving Numerical Computations

We give below a few examples of proofs involving floating-point computations.

As a first example, consider the proof that a given real number $a \in \mathbb{R}$ is nonnegative. One can exhibit another real number r such that $a = r^2$ and apply a lemma stating that all squares of real numbers are nonnegative. Typically, one could use the square root $\sqrt{a}$.

A similar method can be applied to prove that a matrix $A \in \mathbb{R}^{n \times n}$ is positive semidefinite[1] as one can exhibit R such that[2] $A = R^T R$. Such a matrix can be computed using an algorithm called Cholesky decomposition, given in Figure 1. The algorithm succeeds, taking neither square roots of negative numbers nor divisions by zero, whenever A is positive definite[3].

When executed with floating-point arithmetic, the exact equality $A = R^T R$ is lost but it remains possible to bound the accumulated rounding errors in the Cholesky decomposition such that the following theorem holds under mild conditions.

▶ **Theorem 1** (Corollary 2.4 in [34]). *For $A \in R^{n \times n}$, defining $c := \frac{(n+1)\epsilon}{1-2(n+1)\epsilon} \operatorname{tr}(A) + 4n\,(2(n+1) + \max_i A_{i,i})\,\eta$, if the floating-point Cholesky decomposition succeeds on $A - c\,I$, then A is positive definite. ϵ and η are tiny constants given by the floating-point format used.*

A formal proof in Coq of this theorem can be found in a previous work [33]. Thus, an efficient implementation of floating-point arithmetic inside the proof assistant leads to efficient proofs of matrix positive definiteness. This can have multiple applications, such as proving that polynomials are nonnegative by expressing them as sums of squares [26] which can be used in a proof of the Kepler conjecture [24].

Interval arithmetic constitutes another example of proofs involving numerical computations. Sound enclosing intervals can be easily computed in floating-point arithmetic using directed roundings, towards $\pm\infty$ for lower or upper bounds. The Coq.Interval library [25] implements interval arithmetic and could benefit from efficient floating-point arithmetic.

More generally, there are many results on rigorous numerical methods [35] that could see efficient formal implementations provided efficient floating-point arithmetic is available inside proof assistants.

1.2 Objectives

The Coq proof assistant has built-in support for computation, which can be used within proofs, and recent progress have been done to provide efficient integer computation (relying on 63-bit machine integers).

---

[1] A matrix $A \in \mathbb{R}^{n \times n}$ is said positive semidefinite when for all $x \in \mathbb{R}^n$, $x^T A\, x \geq 0$.
[2] Since, when $A = R^T R$, one gets $x^T A\, x = x^T \left(R^T R\right) x = (Rx)^T (Rx) = \|Rx\|^2 \geq 0$.
[3] A matrix $A \in \mathbb{R}^{n \times n}$ is said positive definite when for all $x \in \mathbb{R}^n \setminus \{0\}$, $x^T A\, x > 0$.

The overall goal of this work is to implement efficient floating-point computation in Coq, relying directly on machine `binary64` floats, instead of emulating floats with pairs of integers. Experimentally, that latter emulation in Coq incurs a slowdown of about three orders of magnitude with respect to an equivalent implementation written in OCaml.

1.3 Outline

The article is organized as follows: Section 2 provides the background required to position our approach, from proof-by-reflection to the IEEE 754 standard for floating-point arithmetic to interval arithmetic formalized in Coq. Section 3 is devoted to the implementation itself, with a special focus on the interface that its exposes. Section 4 gathers a discussion on several design choices or technicalities that have been important to carry out the implementation and avoid some pitfalls. Section 5 provides benchmarks to evaluate the performance of the implementation. Section 6 finally gives concluding remarks and perspectives for future work.

2 Prerequisites and Related Works

In this section, we start by reviewing the two main features that underlie and motivate our work in the Coq proof assistant: Poincaré's principle and the availability of efficient reduction tactics (in Section 2.1). We then give an overview of all notions of floating-point arithmetic that appear necessary to make this paper self-contained (in Section 2.2). We finally summarize the features of two related Coq libraries that are either a prerequisite for our developments (in Section 2.3), or an important building block for a possible extension of this work (in Section 2.4).

2.1 Proof by Reflection and Efficient Numerical Computation

In the family of formal proof assistants, the underlying logic of several systems – including Agda, Coq, Lego, and Nuprl [2] – provides a notion of definitional equality that allows one to automatically prove some equalities by a mere computation. This feature is called *Poincaré's principle* in reference to Poincaré's statement that "a reasoning proving that $2 + 2 = 4$ is not a proof in the strict sense, it is a verification" [32, chap. I]. Based upon this principle, the so-called *proof by reflection* methodology has been developed to take advantage of the computational capabilities of the provers and build efficient (semi)-decision procedures [7]: this approach has been successfully applied to various application domains, such as: graph theory, with the formal verification of the four-color theorem in Coq by Gonthier and Werner [14], discrete geometry, with the formal proof of the Kepler conjecture developed in the Flyspeck project [17], Boolean satisfiability, with the verification of SAT traces in Coq [1], satisfiability modulo theories, with the development of the SMTCoq library [13], or global optimization, with the development of the ValidSDP library [26].

To be able to address the verification of increasingly complex proofs relying on this approach, works have been carried out to increase the computational performance of proof assistants, relying on two complementary approaches: (*i*) implement alternative evaluation engines, such as evaluators based on compilation to bytecode or native code, and (*ii*) optimized data structures that might be based on machine values and hardware operators.

For example, the Isabelle proof assistant provides (*i*) several evaluators that can be used within proofs, and allows one to generate Standard ML, OCaml, Haskell, or Scala code, then (*ii*) libraries of fast machine words (for fixed size or unspecified size) have been developed while ensuring compatibility with all Isabelle's target languages and evaluators [23].

In this work, we specifically focus on the Coq proof assistant which offers in particular (*i*) the reduction tactics `vm_compute`, involving bytecode compilation and evaluation by a virtual machine [15] and `native_compute`, involving code generation and native OCaml compilation [3], as well as (*ii*) *machine integers*, upon which the `Bignums` library for multiple-precision arithmetic has been developed [16].

Regarding machine integers in Coq, the original implementation by Spiwack [1, 39] was based on the so-called *retro-knowledge* approach, which consisted in developing a reference implementation of 31-bit integer operators in Coq (using lists of bits), then optimizing their evaluation in `vm_compute` (and later `native_compute`) by replacing the considered Coq operator on-the-fly with the corresponding hardware operator. The implicit assumption here is that both implementations match. This implementation has been recently replaced with so-called *primitive integers*[4] [12]: this approach required adding a representation of 63-bit machine integers in the kernel, and has the two-fold benefit of offering efficient operators for all reduction strategies with a compact representation of integers, and making explicit the axioms that specify the primitive operators.

The overall aim of this work is to provide a similar facility for floating-point arithmetic, to be able to compute with *primitive floating-point numbers* in Coq, instead of emulating floating-point numbers with pairs of integers.

A facility to compute with floating-point numbers for prototyping purposes is available in the PVS proof assistant thanks to the PVSio package [31] but to the best of our knowledge, no proof assistant currently provides support for machine floating-point computations in the scope of proof by reflection.

2.2 Floating-point Arithmetic

This section reviews the main concepts of floating-point arithmetic used in the remainder of this paper. The reader interested in more details could find them in reference books [30].

Computing in floating-point arithmetic amounts to performing calculations in what is often called scientific notation with one digit before the dot, a fixed number of digits following it and a power of ten specifying the position of the dot, hence the name *floating-point* arithmetic. When results do not fit in the required precision, they have to be rounded, e.g., with a precision of five digits, $1.234 \cdot 10^2 + 5.678 \cdot 10^{-1} = 1.240 \cdot 10^2$.

2.2.1 IEEE 754 Standard

Implementations of floating-point arithmetic in hardware nowadays adhere to the IEEE 754 standard [19]. This standard prescribes sets of floating-point numbers, mostly as subsets of the real numbers field $\mathbb{R}$, binary representations for them, rounding modes and basic arithmetic operators $+$, $-$, $\times$, $\div$ and $\sqrt{\cdot}$ defined as functions giving the same result as the operator in the real field composed with a rounding.

A floating-point format $\mathbb{F}$ is a subset of $\mathbb{R}$ such that $x \in \mathbb{F}$ when

$$x = m\beta^e \tag{1}$$

for some $m, e \in \mathbb{Z}$, $|m| < \beta^p$ and $e_{\min} \leq e \leq e_{\max} - p$. The integer m is called the *mantissa* of x and e its *exponent*[5]. The constants β and p are called respectively the *radix* and *precision*

[4] See the pull request `https://github.com/coq/coq/pull/6914`.
[5] More precisely called *quantum exponent* [30, p. 14].

of the format $\mathbb{F}$ while the constants $e_{\min}$ and $e_{\max}$ define the exponent range of $\mathbb{F}$. Some floating-point values can have multiple representations, e.g., $1230 \cdot 10^2 = 123 \cdot 10^3$. To get a canonical representation, $|m| \geq \beta^{p-1}$ is enforced as soon as $|x| \geq \beta^{p-1+e_{\min}}$. In other words, all the space allowed by the precision is used for the mantissa. Mantissas smaller than β^{p-1} are only used for tiny values x such that $\beta^{e_{\min}} \leq |x| < \beta^{p-1+e_{\min}}$, called *denormalized numbers*. Finally, 0 can get a canonical representation by arbitrary choosing an exponent.

2.2.1.1 Binary64 Format

The IEEE 754 standard defines multiple formats in radix $\beta = 2$ and $\beta = 10$ and various precisions. In the remaining of this paper, `binary64` will be the only format considered[6]. This is a binary format, i.e. $\beta = 2$, offering a precision of $p = 53$ bits and its minimal and maximal exponents are respectively $e_{\min} = -1074$ and $e_{\max} = 1024$. As its name suggests, this format enjoys a binary representation on 64 bits as follows:

sign exponent (11 bits) mantissa (52 bits)

The exponent is encoded on 11 bits while the mantissa is encoded as its sign and its absolute value on 52 bits[7]. One can notice that, out the of the 2048 values enabled by the 11 bits of exponent, two are unused when encoding exponents in the range $[e_{\min}, e_{\max} - p] = [-1074, 971]$. One is used for denormalized numbers, and 0 when the mantissa is 0, the other for special values NaN, and infinities when the mantissa is 0.

The two infinities $-\infty$ and $+\infty$ are used to represent values that are too large to fit in the range of representable numbers. Similarly, it is worth noting that due to the sign bit, there are actually two representations of 0, namely -0 and $+0$. The standard states that these two values should behave as if they were equal for comparison operators $=, <$ and $\leq$. However, they can be distinguished since $1 \div (+0)$ returns $+\infty$ whereas $1 \div (-0)$ returns $-\infty$. Finally, NaN stands for "Not a Number" and is used when a computation does not have any mathematical meaning, e.g., $0 \div 0$ or $\sqrt{-2}$. NaNs propagate, i.e., any operator on a NaN returns a NaN. Moreover, comparison with a NaN always returns false, in particular both $x < y$ and $x \geq y$ are false when x is a NaN, as well as[8] $x = x$. Thanks to the mantissa and sign bits, there are actually $2^{53} - 2$ different NaN values. These payloads can be used to keep track of which error created the special value but they are only partially specified by the standard and are in practice hardware dependent.

2.2.1.2 Precise Specification of Rounding Modes

From a formal point of view, a key definition introduced by the IEEE 754 standard is the notion of rounding. For a given floating-point format $\mathbb{F}$, a rounding is an increasing function $\bigcirc : \mathbb{R} \to \mathbb{F} \cup \{\pm\infty\}$ whose restriction to $\mathbb{F}$ is identity, that is:

$$\begin{cases} \forall x, y \in \mathbb{R}, & x \leq y \implies \bigcirc(x) \leq \bigcirc(y) \\ \forall x \in \mathbb{R}, & x \in \mathbb{F} \implies \bigcirc(x) = x. \end{cases}$$

The IEEE 754-2008 standard [19] defines five standard rounding modes:

[6] It is the usual implementation of the type `double` in the C language.
[7] It actually fits in 53 bits but, except for denormalized numbers, the most significant one is always 1 and doesn't need to be explicitly encoded.
[8] This is a simple way to test for NaN as otherwise $x = x$ is always true.

toward $-\infty$: $\mathrm{RD}(x)$ is the largest floating-point number $\leq x$;
toward $+\infty$: $\mathrm{RU}(x)$ is the smallest floating-point number $\geq x$;
toward zero: $\mathrm{RZ}(x)$ is equal to $\mathrm{RD}(x)$ if $x \geq 0$, and to $\mathrm{RU}(x)$ if $x \leq 0$;
to nearest even: $\mathrm{RNE}(x)$ is the floating-point number closest to x.
In case of a tie: the one with an even mantissa;
to nearest away from zero: $\mathrm{RNA}(x)$ is the floating-point number closest to x.
In case of a tie: the one with the largest mantissa in absolute value.
In this work, we will only rely on the RNE rounding, which is the default rounding mode in most floating-point programming environments. See Section 4.1 for a more in depth discussion of this point.

Then, all floating-point operators are required to be correctly rounded, that is to say, they should behave as if they were computed with an infinitely precise mantissa, then rounded according to the specified rounding mode. To be more precise, for a given floating-point format $\mathbb{F}$, operator $* : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, and rounding mode $\bigcirc : \mathbb{R} \to \mathbb{F}$, a correctly-rounded implementation $\circledast$ of $*$ should verify:

$$\forall x, y \in \mathbb{F}, \quad x \circledast y = \bigcirc(x * y).$$

The benefits of this definition are two-fold:
- all floating-point operators that are correctly-rounded (the 2008 revision of the standard requiring this for $+$, $-$, $\times$, $\div$, $\sqrt{\cdot}$) are fully-specified, which straightforwardly ensures the reproducibility of the results;
- it allows one to devise floating-point algorithms that directly rely upon this specification, as exemplified in the upcoming Section 2.2.2.

2.2.2 Error Free Transformations

Noticing that the rounding error of a floating-point addition is itself a floating-point number, algorithms such as Fast2Sum [11] and 2Sum [21, 28] can compute that exact error, taking advantage of correct rounding.

These two "compensated summation algorithms" fall into the larger class of error-free transformations [22, 37] which constitute an essential building block in the development of extended precision floating-point algorithms.

2.2.3 Standard Model

Although precise specifications are known for roundings, hence for basic arithmetic operators, a simpler model is commonly used to prove compound bounds of rounding errors on larger expressions [18]. Despite being weaker, this model is more amenable to algebraic proofs, whether pen and paper or mechanized. Called standard model of floating-point arithmetic, it states the following main properties in the absence of overflow[9]

$$\forall x, y \in \mathbb{F}, \quad \exists \delta, \quad |\delta| \leq \epsilon \ \wedge \ \bigcirc(x + y) = (1 + \delta)(x + y) \tag{2}$$

$$\forall x, y \in \mathbb{F}, \quad \exists \delta, \varphi, \quad |\delta| \leq \epsilon \ \wedge \ |\varphi| \leq \eta \ \wedge \ \bigcirc(x \times y) = (1 + \delta)(x \times y) + \varphi \tag{3}$$

where ϵ and η are tiny constants depending on the floating-point format[10]. As a recent example, the following result is proved in a slightly refined standard model [20].

[9] Overflow can often be handled separately.
[10] For `binary64` and $\bigcirc$ a rounding to nearest, $\epsilon = 2^{-53}$ and $\eta = 2^{-1075}$.

▶ **Theorem 2** (Theorem 4.1 in [20]). *For $x \in \mathbb{F}^n$, denoting $\hat{s}$ the sum $\sum_{i=1}^n x_i$ computed with floating-point arithmetic in any order*[11]*, assuming no overflow occurs, it satisfies*

$$\left|\hat{s} - \sum_{i=1}^{n} x_i\right| \leq \frac{(n-1)\epsilon}{1+\epsilon} \left(\sum_{i=1}^{n} |x_i|\right).$$

Coq proofs of such results can be performed, and are at the core of the proof of Theorem 1 [33].

2.3 The Flocq Library

Flocq [5, 6] is a Coq library offering a very generic formalization of floating-point arithmetic. Radix and precision can be fully parameterized and floating-point values are defined, similarly to (1), as a subset of the real numbers $\mathbb{R}$ provided in the Coq standard library [27, Chapter 1].

More specifically, multiple models are available:

- With an unbounded exponent range, i.e., without underflow nor overflow. Although unrealistic, this model is attractive for its simplicity and commonly used for error bounds [18].
- With an exponent range only lower bounded, i.e., with underflow but without overflow. This may still seem unrealistic but overflows can often be studied separately which usually proves much harder for underflows [33].
- A binary model of the `binary32` and `binary64` formats defined in the IEEE 754 standard, with underflows, overflows to infinities, signed zeros and NaNs with payloads. This model is used in the verified C compiler CompCert [4].

Along with these models and links between them, the library contains many classical results about roundings, about some error-free transformations as presented in Section 2.2.2, and basic properties of the standard model described in Section 2.2.3.

The library is mainly developed by Sylvie Boldo and Guillaume Melquiond and is available at URL `http://flocq.gforge.inria.fr/`.

2.4 The Coq.Interval Library

Another Coq library could benefit from efficient floating-point arithmetic: Coq.Interval [25], which offers a modular formalization of interval arithmetic. First, module types (a.k.a. signatures) are defined for floating-point and interval operators. Then, several implementations of the floating-point signature are provided, relying on the Flocq library and specifically its model with unbounded exponent range. A generic implementation is provided, as well as a specialized implementation assuming radix 2 and representing mantissa and exponent as pairs of integers from `Bignums`. Next, a parameterized module implements interval operators where intervals are pairs of floating-point numbers, and related computations are performed using directed roundings, towards $-\infty$ or $+\infty$. Elementary functions such as `exp`, `ln` or `atan` are provided among these interval operators, but correct rounding is not guaranteed (namely, the computed intervals can be overestimated, albeit the containment property always holds and has been formally proved). Finally, tactics `interval` (decision procedure) and `interval_intro` (for forward reasoning) are provided to automatically and formally prove inequalities on real-valued expressions.

The library is mainly developed by Guillaume Melquiond and is available at URL `http://coq-interval.gforge.inria.fr`.

[11] Floating-point addition is not associative.

3 Contributions

In order to provide access to efficient floating-point arithmetic inside proofs, the following steps have been performed:

1. Define a minimal working interface for the IEEE 754 `binary64` format. See Section 3.1.
2. Devise a specification of this interface that enables using `binary64` computations in proofs. This specification should be compatible with Flocq, so that all previously proved results, both in Flocq and based upon it, can be straightforwardly reused, using a simple compatibility layer. Details are in Section 3.2.
3. Implement the chosen interface in Coq's various computation mechanisms, i.e., `compute`, `vm_compute` and `native_compute` at the OCaml and C levels. A brief summary of the implementation is given in Section 3.3 and salient points are discussed in Section 4.
4. Assess the performance by running some benchmarks. Results are given in Section 5.

3.1 Interface

In our modified version of Coq, after typing

```
Require Import Floats.
```

the user gets access to the following interface[12]:

```
Parameter float : Set.
```

A type for primitive floating-point values. Inside the kernel, this is mapped to the `float` type of OCaml[13] that matches `binary64`.

```
Parameters add sub mul div : float -> float -> float.
Parameters sqrt opp abs : float -> float.
```

The basic arithmetic operators $+$, $-$, $\times$, $\div$, $\sqrt{\cdot}$, opposite and absolute value.

```
Variant float_comparison : Set := FEq | FLt | FGt | FNotComparable.
Parameter compare : float -> float -> float_comparison.
```

A comparison function that behaves as specified by the IEEE 754 standard. In particular $+0$ and -0 are considered equal and NaNs are not comparable to any value, hence the `FNotComparable` answer.

A few functions are then given to examine or craft precise floating-point values by translating them from or to primitive integers.

```
Variant float_class : Set :=
  | PNormal | NNormal | PSubn | NSubn | PZero | NZero | PInf | NInf | NaN.
Parameter classify : float -> float_class.
```

A function testing whether a given value is a NaN, an infinity (`NInf` and `PInf` for $-\infty$ and $+\infty$ respectively), -0 (`NZero`), $+0$ (`PZero`), a denormalized value (`NSubn` and `PSubn`) or a regular one (`NNormal` and `PNormal`).

[12] Defined in file `theories/Floats/PrimFloat.v` in the implementation.
[13] The implementation language of Coq.

```
Definition shift := 2101%int63.  (* = 2 × emax + prec *)
Parameter frshiftexp : float -> float * Int63.int.
```

`frshiftexp` f returns a pair (m, e) such that[14] $|m| \in [0.5, 1)$ and $f = m \times 2^{e-\texttt{shift}}$. Primitive integers are unsigned so `shift` is used to ensure that e is nonnegative.

```
Parameter ldshiftexp : float -> Int63.int -> float.
```

`ldshiftexp` f e returns $f \times 2^{e-\texttt{shift}}$. This is the reverse of `frshiftexp` and it is exact except when underflow or overflow occurs, in which case the result is rounded using RNE.

```
Parameter normfr_mantissa : float -> Int63.int.
```

When f, typically obtained from `frshiftexp`, satisfies $|f| \in [0.5, 1)$, `normfr_mantissa` f returns the primitive integer $|f| \times 2^p$, that is the integer encoding the mantissa of f.

```
Parameter of_int63 : Int63.int -> float.
```

Converts a primitive integer to a floating-point value. Since primitive integers are unsigned 63-bit integers, they do not all fit into the 53-bit mantissas of the `binary64` format. Values that do not fit are rounded using RNE.

Finally, two functions compute the successor and predecessor of a floating-point value. They can be used to implement interval arithmetic for instance.

```
Parameters next_up, next_down : float -> float.
```

Equipped with this interface, the Coq user can now perform floating-point computations using the processor operators and any of the evaluation mechanisms provided by Coq.

```
Coq < Require Import Floats. Open Scope float_scope.
Coq < Eval compute in 1 + 0.5.
     = 1.5 : float
Coq < Eval vm_compute in 1 / -0.
     = neg_infinity : float
Coq < Eval native_compute in 0 / 0.
     = nan : float
```

3.2 Specification

Although floating-point computations are possible, they remain entirely useless in proofs at this point, since there is no specification of their behavior. We thus need a Coq specification of floating-point arithmetic.

First of all, the set of floating-point values itself has to be specified[15].

```
Variant spec_float :=
  | S754_zero (sign : bool)  (* true for -0, false for +0 *)
  | S754_infinity (sign : bool)
  | S754_nan
  | S754_finite (sign : bool) (mantissa : positive) (exponent : Z).
```

[14] When f is finite and non zero, otherwise $(m, e) = (f, 0)$.
[15] See file `theories/Floats/SpecFloat.v` in the implementation.

This is similar to the `full_float` type in the `IEEE754.Binary` module of the Flocq library except for one point: the sign and payload of NaNs are not modeled here. It is also worth noting that this models much more values than the `binary64` format[16] since no bounds on mantissas nor exponents are enforced. This makes for a simple specification.

Then, each of the above operators must be specified on this `spec_float` type. This specification is mostly borrowed[17] from the `IEEE754.Binary` module of the Flocq library and totals 398 lines in our implementation[18]. We thus only detail the multiplication operator. We first need to define a few characteristics of the `binary64` format as seen in Section 2.2.1.1

```
Definition prec := 53%Z.
Definition emax := 1024%Z.
Definition emin := (3 - emax - prec)%Z. (* = -1074 *)
Definition fexp e := Z.max (e - prec) emin.
```

When $|x| \in \left[2^{e-1}, 2^e\right)$, then `fexp` e is the exponent used to encode x in the `binary64` format.

As seen in Section 2.2.1.2, the floating point multiplication is defined by $x \otimes y = \bigcirc(x \times y)$. When $x = m_x 2^{e_x}$ and $y = m_y 2^{e_y}$, then $x \times y = (m_x \times m_y)\, 2^{e_x+e_y}$ and the rounding operator $\bigcirc$ has to remove the extra bits in the mantissa to make this value fit in the format. To this end, we first abstract the bits to remove as two booleans, the *rounding* bit remembers the first forgotten bit whereas the *sticky* bit is `true` when any of the remaining forgotten bits is 1 and `false` when they are all 0. The function `shr_1` then shifts a mantissa one bit to the right, updating the rounding and sticky bits accordingly

```
Record shr_record := { shr_m : Z ; shr_r : bool ; shr_s : bool }.
Definition shr_1 mrs :=
  let s := orb (shr_r mrs) (shr_s mrs) in match shr_m mrs with
  | Z0 (* 0 *) => Build_shr_record Z0 false s
  | Zpos xH (* 1 *) => Build_shr_record Z0 true s
  | Zpos (xO p) (* 2p *) => Build_shr_record (Zpos p) false s
  | Zpos (xI p) (* 2p+1 *) => Build_shr_record (Zpos p) true s
  | ... (* same for Zneg _ *) end.
```

Eventually, `shr` can iterate n shifts and `shr_fexp` removes the required number of bits using the above function `fexp` (`Zdigits2 m` is the number of bits of `m`)

```
Definition shr mrs e n := match n with
  | Zpos p => (iter_pos shr_1 p mrs, (e + n)%Z) | _ => (mrs, e) end.
Definition shr_fexp m e :=
  shr (Build_shr_record m false false) e (fexp (Zdigits2 m + e) - e).
```

It now remains to round the mantissa according to the values of the rounding and sticky bits

```
Definition round_nearest_even mrs := match mrs with
  | Build_shr_record mx false _ => mx
  | Build_shr_record mx true false => if Z.even mx then mx else (mx + 1)%Z
  | Build_shr_record mx true true => (mx + 1)%Z end.
```

[16] `spec_float` gathers an infinite number of values, whereas `binary64` only contains finitely many values.
[17] Except for the specifications of `frexp`, `ldexp`, `normfr_mantissa`, `succ` and `pred` which were not yet present in Flocq and which we took the opportunity to add `https://gitlab.inria.fr/flocq/flocq/merge_requests/3`.
[18] See file `theories/Floats/SpecFloat.v` in the implementation.

Finally, the rounding function first shifts the mantissa, rounds it, shifts the result one bit to the right in case the rounding added an extra bit and handles potential overflows

```
Definition binary_round_aux sx mx ex :=
  let '(mrs', e') := shr_fexp mx ex in
  let '(mrs'', e'') := shr_fexp (round_nearest_even mrs') e'
  in match shr_m mrs'' with Z0 => S754_zero sx | Zneg _ => S754_nan
  | Zpos m => if Zle_bool e'' (emax - prec) then S754_finite sx m e''
    else S754_infinity sx end.
```

Thus, it remains to the multiplication to handle all particular cases

```
Definition SFmul x y := match x, y with
  | S754_nan, _ | _, S754_nan => S754_nan
  | S754_infinity sx, S754_infinity sy => S754_infinity (xorb sx sy)
  | S754_infinity sx, S754_finite sy _ _ => S754_infinity (xorb sx sy)
  | S754_finite sx _ _, S754_infinity sy => S754_infinity (xorb sx sy)
  | S754_infinity _, S754_zero _ => S754_nan
  | S754_zero _, S754_infinity _ => S754_nan
  | S754_finite sx _ _, S754_zero sy => S754_zero (xorb sx sy)
  | S754_zero sx, S754_finite sy _ _ => S754_zero (xorb sx sy)
  | S754_zero sx, S754_zero sy => S754_zero (xorb sx sy)
  | S754_finite sx mx ex, S754_finite sy my ey =>
    binary_round_aux (xorb sx sy) (Zpos (mx * my)) (ex + ey) end.
```

In addition to the usual operators, two functions are defined going back and forth from primitive floats to specification floats.

```
Definition Prim2SF : float -> spec_float.
Definition SF2Prim : spec_float -> float.
```

Finally, one needs to establish a link between the primitive operators and the specification. This is done by adding axioms to the system.[19] First, to specify the two functions `Prim2SF` and `SF2Prim` above, one needs to characterize those values of type `spec_float` that actually represent a `binary64` floating-point number, i.e., values with appropriately bounded mantissa and exponent.

```
Definition canonical_mantissa m e := Zeq_bool (fexp (Zdigits2 m + e)) e.
Definition bounded m e :=
  andb (canonical_mantissa m e) (Zle_bool e (emax - prec)).
Definition valid_binary x := match x with
  | SF754_finite _ m e => bounded m e | _ => true end.
```

Again, this code comes from the Flocq library [5]. So equipped, the following three axioms can be stated:

```
Axiom Prim2SF_valid : forall x, valid_binary (Prim2SF x) = true.
Axiom SF2Prim_Prim2SF : forall x, SF2Prim (Prim2SF x) = x.
Axiom Prim2SF_SF2Prim :
  forall x, valid_binary x = true -> Prim2SF (SF2Prim x) = x.
```

[19] See file `theories/Floats/FloatAxioms.v` in the implementation.

These properties allow one to prove that both `Prim2SF` and `SF2Prim` are injective and thereby form a bijection between primitive floats and the subset of valid specification floats.

```
Theorem Prim2SF_inj : forall x y, Prim2SF x = Prim2SF y -> x = y.
Theorem SF2Prim_inj : forall x y, SF2Prim x = SF2Prim y ->
  valid_binary x = true -> valid_binary y = true -> x = y.
```

Thus, all of the fifteen operators given in Section 3.1 are linked to their specification by an axiom such as, for the multiplication:

```
Axiom mul_spec :
  forall x y, Prim2SF (x * y)%float = SFmul (Prim2SF x) (Prim2SF y).
```

Since the specification is almost identical to the `IEEE754.Binary` module of Flocq, a link with Flocq is straightforwardly built[20], establishing a bridge towards real numbers and giving access to all the results already proved in the library. This plays a key role in enabling actual proofs using primitive floating-point computations. Moreover, this enables to gain additional confidence in the above non trivial specification, since Flocq contains correctness theorems basically stating for instance[21] that, except when overflow occurs, `SFmul x y` is indeed the rounding of the real number $\mathtt{x} \times \mathtt{y}$.

3.3 Implementation

The implementation was submitted to be integrated in Coq through the GitHub pull request `https://github.com/coq/coq/pull/9867`.

Below is an overview of the size of the development at the time of writing, summarized by sub-components (over the ≈ 3.7 kLoC added).

- OCaml and C: 1815 LoC
  (floats $\hookrightarrow$ kernel : 1070) (`vm_compute` support: 255) (`native_compute` support: 355) (parsing and pretty-printing: 85) (Coq checker: 50)
- Coq specifications: 620 LoC [mostly borrowed from Flocq]
- Coq proofs: 340 LoC
- Tests: 800 LoC
- Sphinx documentation: 115 LoC

This implementation required the addition of some code in the kernel of Coq. Most of it only consists in wrapping the floating-point operators into the different evaluation mechanisms of Coq and its core, actually dealing with floating-point arithmetic, can be found in the files `kernel/float64.ml`, `kernel/byterun/coq_interp.c` and `kernel/byterun/coq_float64.h`. Most operators are implemented in C, as required by the `vm_compute` mechanism, and boil down to calls to the appropriate functions of the C standard library. Thus, no involved algorithmic happens in this added code itself.

[20] See `https://gitlab.inria.fr/flocq/flocq/merge_requests/6`.
[21] See theorem `Bmult_correct` in module `Flocq.IEEE754.Binary`.

4 Discussion

4.1 Rounding Modes

We implement only one of the five rounding modes defined in the IEEE 754-2008 standard, namely rounding to nearest even (RNE). We argue here that implementing other rounding modes would not only easily be seriously harmful in terms of performance, notwithstanding the potential threat to soundness of the implementation, but also not very useful.

Unfortunately on most common processors, operators with different rounding modes are not implemented using different opcodes but a status flag. Once the flag is set to a particular rounding mode, all subsequent computations are performed with this rounding mode. Changing the rounding mode is then costly as it requires flushing pipelines.

Interval arithmetic constitutes the main use of rounding modes other than RNE we can foresee in a proof assistant. A common solution to the aforementioned performance issue is to set the rounding mode once to $+\infty$ (RU), used to compute upper bounds, and emulate rounding toward $-\infty$ (RD), used to compute lower bounds, by relying on properties like[22] $\mathrm{RD}(x+y) = -\mathrm{RU}((-x)+(-y))$. Although a monadic interface could be a reasonable implementation, this remains an imperative programming feature and doesn't integrate well within the functional paradigm offered by Coq. Moreover, if no particular care is taken to avoid or disable them, wild compiler optimizations – assuming that only RNE is used – could easily break the previous property, thus ruining the soundness of the whole approach.

However, interval arithmetic doesn't require precise directed roundings but only over- and under-approximations thereof. We thus offer the `next_up` and `next_down` functions, computing the successor and predecessor of a floating-point value. Together with rounding to nearest operators, they satisfy the following property, ensuring soundness of interval arithmetic while providing a reasonably precise approximation of directed roundings:

$$\forall x \in \mathbb{R}, \quad \mathrm{RU}(x) \leq \texttt{next\_up}(\mathrm{RNE}(x))$$
$$\forall x \in \mathbb{R}, \quad \texttt{next\_down}(\mathrm{RNE}(x)) \leq \mathrm{RD}(x).$$

4.2 Parsing and Pretty-Printing

Parsing and pretty-printing floating-point values is a non trivial question. We expect the following main property: printing a floating-point value and then reparsing the output of the printing function should give the initial value, i.e., $parse \circ print$ should be the identity over `binary64`. It is worth noting that this necessarily implies the injectivity of the printing function. However, we don't require the parsing function to be injective, i.e., we do accept that multiple strings are parsed as the same floating-point value.

A simple solution would be to print an exact hexadecimal representation of the floating-point values, with a binary exponent, e.g., "0xcp-3". This fulfills the above requirement. Unfortunately, this is not very user-friendly. A decimal output would be much more human readable, e.g., "1.5" instead of "0xcp-3".

It is known that printing `binary64` values using at least 17 significant digits and implementing parsing as a rounding to nearest guarantees the above requirements [30, Table 2.3, p. 44]. This is thus the adopted solution. The current version of Coq only offers support for parsing and printing integer constants, so we extended this support[23] to decimal constants using the ubiquitous format ⟨*integer_part*⟩.⟨*fractional_part*⟩e⟨*decimal_exponent*⟩, e.g., "1.23e-4".

[22] The opposite $x \mapsto -x$ being exact in floating-point arithmetic (the sign bit is simply flipped).
[23] See the pull request `https://github.com/coq/coq/pull/8764`.

4.3 Soundness

During our development, we identified three main potential threats to soundness:

Specification Issues due to a mismatch w.r.t. the implementation would break the soundness. We hope that taking in extenso our specification from the Flocq library, resulting from a few decades of experience in the field and proving links with other models, mitigates this risk. Moreover, such an error in the specification can only be harmful when the corresponding axiom is used. It is worth noting that all the axioms used in a proved theorem explicitly appear in the result of the Coq command `Print Assumptions`.

Incompatible Implementations in different evaluation mechanisms (`compute`, `vm_compute` or `native_compute`) or even on different machines could lead to a proof of `False` by evaluating a same term to different results. For instance, the payload of NaNs is not fully specified by the IEEE 754 standard and different hardwares can produce different NaNs for a same computation. That's why we chose to consider all NaNs as equal and not distinguish them. Thus incompatible implementations at the bit level remain compatible at the logical level. Double roundings due to the x87 on old 32 bits architectures [29] could also be harmful. The OCaml[24] compiler systematically relies on it, forcing us to implement all floating-point operators in C and to use the appropriate compiler flags. A runtime test[25] is eventually added to prevent Coq from running in case of miscompilation. Another extreme example of implementation discrepancy would be a hardware bug such as the one encountered in the division of the early Pentium processors.

Incorrect Convertibility Test that distinguish two values that shouldn't or vice versa is also a threat. For instance, implementing this test using the equality test on floating-point values (as defined in the IEEE 754 standard) would be wrong as it equates -0 and $+0$ which should be distinguished since $1 \div (-0) = -\infty \neq 1 \div (+0) = +\infty$. Fortunately enough, this keeps a very simple implementation, with the following OCaml code:

```
let equal f1 f2 =
  let is_nan f = f <> f in
  match classify_float f1 with
  | FP_normal | FP_subnormal | FP_infinite -> f1 = f2
  | FP_nan -> is_nan f2 | FP_zero -> f1 = f2 && 1. /. f1 = 1. /. f2
```

A few other, more minor, points appeared during the development. Among them, the fact that primitive integers in Coq are unsigned did require some care[26]. Finally, the way OCaml optimizes arrays[27] of floating-point values[28] did cause a few nasty bugs, although it is unlikely that such bugs could lead to a proof of `False` as they often yield a mere segmentation fault.

[24] The implementation language of Coq.
[25] See file `kernel/float64.ml` in the implementation.
[26] We indeed fixed a few soundness bugs in primitive integers, pertaining with unsigned integers, before they were merged in Coq master development branch (`https://github.com/coq/coq/pull/6914`).
[27] Arrays are used to communicate environments between the OCaml implementation of the kernel and the C implementation of the `vm_compute` virtual machine.
[28] This causes other issues in OCaml itself and seems to be a hot topic currently in the OCaml community [9].

5 Benchmarks

The overall objective of this work is to increase the performance of reflexive tactics involving floating-point arithmetic in Coq. Thus we first measure the performance gain on such a tactic, then evaluate it on its individual floating-point operators. We first present the reference problems under study (Section 5.1), then recap the hardware and software setup for these benchmarks (Section 5.2), and finally give the experimental results (Section 5.3).

5.1 Reference Test-suite

We developed a reflexive tactic `posdef_check`, performing some matrix positive definiteness check along the lines of Theorem 1 introduced in Section 1.1. Its implementation was adapted by reusing building blocks from our previous work on the `validsdp` tactic for multivariate polynomial positivity [26].

This tactic is available in four flavors using `vm_compute` or `native_compute` and emulated floats or primitive floats. Emulated floats are a state of the art implementation of floating point arithmetic, based on primitive integers, from the Coq.Interval library whereas primitive floats are our new implementation.

Regarding the test-suite, we generated a set of random positive definite matrices (after fixing a given seed to make the random data reproducible) of size 50×50 up to 400×400.

We perform two kinds of benchmarks on this test-suite: the overall speedup between the versions of `posdef_check` using emulated vs. primitive floats; and the individual speedup in floating-point operators involved in this tactic.

5.2 Hardware/Software Setup

The formalization of the `posdef_check` tactic relies on a large set of dependencies that takes around one hour to compile. For greater convenience, we devised some Docker images containing the benchmark environment, based on Debian Stretch, opam 2 (the OCaml package manager) and OCaml 4.07.0+flambda. The source code of all benchmarks as well as guidelines to install Docker and run the benchmarks are gathered on GitHub at this URL: `https://github.com/validsdp/benchs-primitive-floats/tree/1.0`

The use of Docker (a so-called *OS-level virtualization system*) for these benchmarks yields a number of interesting features, beyond the facility to download and run a pre-built image on different machines: it runs containers in an isolated environment from the host machine, it ensures portability (across OSes such as GNU/Linux, macOS and Windows) and reproducibility, while being more lightweight than traditional virtual machines (VMs).

The experimental results of the upcoming Section 5.3 have been obtained using a Debian GNU/Linux workstation based on a Intel Core i7-7700 CPU clocked at 3.60 GHz, with 16 GB of RAM. All benchmarks have been executed sequentially (namely, without the `-j` option of `make`), with a total elapsed time of about 3h35', using the following image: `"docker pull registry.gitlab.com/erikmd/docker-coq-primitive-floats/master_compiler-edge:9_coq-2ac1f46532264bacf2b1d8f5b6ee3659fe0cde67"`.

5.3 Experimental Results

We first measure the execution time of the whole tactic on the test-suite and compare it between emulated floats and primitive floats. The results are displayed in Table 1 for `vm_compute` and `native_compute`. Each timing is measured 5 times. The tables indicate the corresponding average and relative error among the 5 samples.

Table 1 Proof time for the reflexive tactic posdef_check.

Source	vm_compute			native_compute		
	Emulated	Primitive	Diff.	Emulated	Primitive	Diff.
mat050	0.16s ±2.0%	0.01s ±0.0%	20x	0.05s ±4.0%	0.02s ±5.1%	3x
mat100	1.16s ±1.3%	0.06s ±5.8%	21x	0.28s ±2.5%	0.03s ±2.5%	9x
mat150	3.61s ±1.2%	0.18s ±2.2%	21x	0.75s ±3.0%	0.08s ±3.5%	9x
mat200	8.68s ±0.2%	0.41s ±1.0%	21x	1.71s ±1.0%	0.18s ±3.4%	10x
mat250	17.14s ±1.3%	0.80s ±0.3%	21x	3.34s ±1.4%	0.33s ±2.1%	10x
mat300	30.01s ±1.2%	1.37s ±0.7%	22x	5.77s ±2.4%	0.56s ±1.0%	11x
mat350	48.31s ±1.3%	2.15s ±0.1%	23x	9.09s ±3.0%	0.81s ±1.2%	11x
mat400	70.19s ±1.4%	3.18s ±0.5%	22x	13.56s ±4.0%	1.12s ±0.7%	12x

One can notice that the obtained speedups are far from the three order of magnitudes separating emulated floats from equivalent OCaml implementations. From the above results, it appears that arithmetic operators constitute most of the computation time with emulated floats (at least 95% with `vm_compute`) but nothing tells us this is still the case with primitive floats. In fact, with primitive floats, most of the computation time is dedicated to list manipulating functions as our matrices are implemented using lists[29] [8]. Thus, we would like to get an idea of the time actually devoted to floating-point arithmetic in the total proof time of our reflexive tactic. We use the following simple methodology: replace each arithmetic operator with a version, uselessly, performing the computation twice[30], then subtract the execution time of the original program ("Op" in the tables) to the one of this modified program ("Op×2" in the tables). The obtained time ("Op time" in the tables) corresponds to the time devoted to the considered arithmetic operator. Note that the redundant computations involved in the modified program ("Op×2") could not be implemented with a mere additional let-in such as `...let m1 := mul a b in let m2 := mul a b in m2` because the virtual machine and the OCaml native compiler would optimize away the unused local definition; but doing so and adding an extra function call `...in select m1 m2` with `Definition select (a b : F.type) := a.` made it possible to use this doubling trick. The results are given in Table 2 for `vm_compute` and Table 3 for `native_compute`, in each case both on addition and multiplication[31]. Again, each timing is measured 5 times. It is worth noting that those last results should be taken more as coarse orders of magnitude than precise results. In particular, due to the overhead stemming from the duplication itself of the operators[32], the speedups are – maybe seriously – underapproximated. Actual speedups could thus be higher than the ones suggested here.

6 Conclusion and Future Work

We developed a theory of floating-point arithmetic for the Coq proof assistant, composed of primitive implementation of basic arithmetic operators ($+$, $-$, $\times$, $\div$, $\sqrt{\cdot}$), using the processor floating-point operators in rounding-to-nearest even, as well as successor and predecessor operators that can be used to approximate directed roundings to $-\infty$ or $+\infty$.

[29] This could be improved using primitive "persistent arrays" once they will be integrated in Coq [1].
[30] Or thousand times for primitive floats to avoid getting a result of the same order of magnitude than the variability of computation times.
[31] Additions and multiplications constitute the vast majority of the arithmetic computations performed in a Cholesky decomposition, as seen in Figure 1.
[32] Like expensive function calls.

Table 2 Computation time for individual operators with `vm_compute`.

Op Source	Emulated floats CPU times (Op×2−Op)	Op	Primitive floats CPU times (Op×1001−Op)	Op	Diff.
add					
mat200	10.78±0.9% − 8.38±2.8%	2.40s	15.72±0.5% − 0.45±1.1%	0.02s	157x
mat250	21.46±1.7% − 16.41±1.5%	5.06s	30.62±0.6% − 0.82±0.6%	0.03s	170x
mat300	37.43±1.4% − 28.63±1.4%	8.80s	53.12±2.4% − 1.40±0.5%	0.05s	170x
mat350	59.42±0.8% − 45.95±2.9%	13.48s	84.19±0.8% − 2.19±0.5%	0.08s	164x
mat400	87.78±0.9% − 66.17±1.7%	21.61s	127.56±8.5% − 3.21±0.3%	0.12s	174x
mul					
mat200	12.21±1.4% − 8.38±2.8%	3.83s	16.10±3.0% − 0.45±1.1%	0.02s	245x
mat250	24.52±1.4% − 16.41±1.5%	8.11s	31.12±3.7% − 0.82±0.6%	0.03s	268x
mat300	42.84±1.7% − 28.63±1.4%	14.21s	53.25±0.8% − 1.40±0.5%	0.05s	274x
mat350	68.23±1.5% − 45.95±2.9%	22.28s	84.33±0.7% − 2.19±0.5%	0.08s	271x
mat400	99.72±1.5% − 66.17±1.7%	33.55s	125.74±0.8% − 3.21±0.3%	0.12s	274x

Table 3 Computation time for individual operators with `native_compute`.

Op Source	Emulated floats CPU times (Op×2−Op)	Op	Primitive floats CPU times (Op×1001−Op)	Op	Diff.
add					
mat200	2.24±1.4% − 1.78±1.7%	0.46s	17.68±1.4% − 0.22±0.9%	0.02s	27x
mat250	4.49±4.2% − 3.41±3.1%	1.08s	34.29±0.7% − 0.37±1.5%	0.03s	32x
mat300	7.25±1.2% − 5.83±4.6%	1.42s	59.57±2.5% − 0.55±0.9%	0.06s	24x
mat350	11.66±3.8% − 9.28±3.5%	2.39s	93.82±1.1% − 0.82±0.8%	0.09s	26x
mat400	17.07±2.9% − 13.14±0.9%	3.93s	141.97±2.6% − 1.18±0.9%	0.14s	28x
mul					
mat200	2.48±1.5% − 1.78±1.7%	0.70s	17.81±1.1% − 0.22±0.9%	0.02s	40x
mat250	4.82±2.4% − 3.41±3.1%	1.41s	35.14±2.1% − 0.37±1.5%	0.04s	41x
mat300	8.41±2.4% − 5.83±4.6%	2.59s	60.66±2.2% − 0.55±0.9%	0.06s	43x
mat350	13.21±2.4% − 9.28±3.5%	3.94s	97.25±1.0% − 0.82±0.8%	0.10s	41x
mat400	19.27±1.5% − 13.14±0.9%	6.13s	138.61±2.3% − 1.18±0.9%	0.14s	45x

This implementation is axiomatized under the assumption that the processor complies with the IEEE 754 standard for floating-point arithmetic. Particular care has been taken to make the implementation compatible across the different reduction engines of Coq, and across different hardware, thereby avoiding soundness issues that could be caused, for example, by the semantics of NaN payloads that is under-specified in the IEEE 754 standard.

We evaluated the performance on an implementation – carried out in Gallina, the input language of Coq – of a Cholesky decomposition that underlies a reflexive tactic for matrix positive definiteness, and the experimental results indicate a speedup of two orders of magnitude for arithmetic operators using `vm_compute`. This is consistent with the performance factor of about three orders of magnitude observed between floating-point arithmetic emulated using primitive integers in Coq and equivalent implementations written in OCaml.

Now that primitive floats are available in a proof assistant, multiple future works can be envisioned. The most obvious one would be to adapt the Coq.Interval library to take advantage of primitive floats. Still in this direction, it is known that the successor and predecessor functions, used to approximate directed roundings, can be efficiently implemented using only arithmetic operators [36, 38]. Such an implementation could enable to remove these functions from the trusted code base. It would also be interesting to look at more elaborate elementary functions such as exp or arctan, relying for example on the CR-libm

implementation [10]. Finally, in an attempt to improve confidence in the consistency between specification and implementation, and while waiting for a fully formally specified hardware interface, it is worth noting that this consistency is amenable to some intensive automatic testing, although exhaustive testing is out of reach for even unary operators on `binary64`.

References

1 Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. Extending Coq with Imperative Features and Its Application to SAT Verification. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2010. doi:10.1007/978-3-642-14052-5_8.

2 Henk Barendregt and Herman Geuvers. Proof-Assistants Using Dependent Type Systems. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 1149–1238. Elsevier and MIT Press, 2001.

3 Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full Reduction at Full Throttle. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*, pages 362–377. Springer, 2011. doi:10.1007/978-3-642-25379-9_26.

4 Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. A Formally-Verified C Compiler Supporting Floating-Point Arithmetic. In Alberto Nannarelli, Peter-Michael Seidel, and Ping Tak Peter Tang, editors, *21st IEEE Symposium on Computer Arithmetic, ARITH 2013, Austin, TX, USA, April 7-10, 2013*, pages 107–115. IEEE Computer Society, 2013. doi:10.1109/ARITH.2013.30.

5 Sylvie Boldo and Guillaume Melquiond. Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In Elisardo Antelo, David Hough, and Paolo Ienne, editors, *20th IEEE Symposium on Computer Arithmetic, ARITH 2011, Tübingen, Germany, 25-27 July 2011*, pages 243–252. IEEE Computer Society, 2011. doi:10.1109/ARITH.2011.40.

6 Sylvie Boldo and Guillaume Melquiond. *Computer Arithmetic and Formal Proofs: Verifying Floating-point Algorithms with the Coq System*. Elsevier, 2017.

7 Samuel Boutin. Using Reflection to Build Efficient and Certified Decision Procedures. In Martín Abadi and Takayasu Ito, editors, *Theoretical Aspects of Computer Software, Third International Symposium, TACS '97, Sendai, Japan, September 23-26, 1997, Proceedings*, volume 1281 of *Lecture Notes in Computer Science*, pages 515–529. Springer, 1997. doi:10.1007/BFb0014565.

8 Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs*, volume 8307 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2013. URL: http://dx.doi.org/10.1007/978-3-319-03545-1_10.

9 Simon Colin, Rodolphe Lepigre, and Gabriel Scherer. Unboxing Mutually Recursive Type Definitions in OCaml. In *Proceedings of JFLA, Les Rousses, France, 30th January to 2nd February 2019.*, 2019.

10 Catherine Daramy-Loirat, David Defour, Florent De Dinechin, Matthieu Gallet, Nicolas Gast, Christoph Lauter, and Jean-Michel Muller. CR-libm. A library of correctly rounded elementary functions in double-precision. Technical report, LIP, ENS Lyon, 2006. URL: https://hal-ens-lyon.archives-ouvertes.fr/ensl-01529804/.

11 T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

12 Maxime Dénès. Towards primitive data types for COQ 63-bits integers and persistent arrays. In *Coq-5, the Coq Workshop 2013*, Rennes, France, July 2013. Extended abstract. URL: https://coq.inria.fr/files/coq5_submission_2.pdf.

13 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.

14 Georges Gonthier. Formal Proof—The Four-Color Theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008. URL: http://www.ams.org/notices/200811/tx081101382p.pdf.

15 Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In Mitchell Wand and Simon L. Peyton Jones, editors, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002.*, pages 235–246. ACM, 2002. doi:10.1145/581478.581501.

16 Benjamin Grégoire and Laurent Théry. A Purely Functional Library for Modular Arithmetic and Its Application to Certifying Large Prime Numbers. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 423–437. Springer, 2006. doi:10.1007/11814771_36.

17 Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Le Truong Hoang, Cezary Kaliszyk, Victor Magron, Sean Mclaughlin, Tat Thang Nguyen, and et al. a formal proof of the Kepler conjecture. *Forum of Mathematics, Pi*, 5:e2, 2017. 29 pages. doi:10.1017/fmp.2017.1.

18 Nicholas Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1996.

19 IEEE Computer Society. IEEE Standard for Floating-Point Arithmetic. *IEEE Standard 754-2008*, 2008.

20 Claude-Pierre Jeannerod and Siegfried M. Rump. On relative errors of floating-point operations: Optimal bounds and applications. *Math. Comput.*, 87(310):803–819, 2018. doi:10.1090/mcom/3234.

21 D. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, Reading, MA, 1973.

22 Peter Kornerup, Vincent Lefèvre, Nicolas Louvet, and Jean-Michel Muller. On the Computation of Correctly Rounded Sums. *IEEE Trans. Computers*, 61(3):289–298, 2012. doi:10.1109/TC.2011.27.

23 Andreas Lochbihler. Fast Machine Words in Isabelle/HOL. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer Science*, pages 388–410. Springer, 2018. doi:10.1007/978-3-319-94821-8_23.

24 Victor Magron, Xavier Allamigeon, Stéphane Gaubert, and Benjamin Werner. Formal Proofs for Nonlinear Optimization. *Journal of Formalized Reasoning*, 8(1):1–24, 2015. URL: http://jfr.unibo.it/article/view/4319.

25 Érik Martin-Dorel and Guillaume Melquiond. Proving Tight Bounds on Univariate Expressions with Elementary Functions in Coq. *Journal of Automated Reasoning*, 57(3):187–217, October 2016. doi:10.1007/s10817-015-9350-4.

26 Érik Martin-Dorel and Pierre Roux. A reflexive tactic for polynomial positivity using numerical solvers and floating-point computations. 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 90–99. ACM, 2017. doi:10.1145/3018610.3018622.

27 Micaela Mayero. *Formalisation et automatisation de preuves en analyses réelle et numérique*. PhD thesis, Université Paris VI, December 2001.

28 O. Møller. Quasi Double-Precision in Floating-Point Addition. *BIT*, 5:37–50, 1965.

29 David Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3):12:1–12:41, 2008. doi:10.1145/1353445.1353446.

30 Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010. doi:10.1007/978-0-8176-4705-6.

31 César Muñoz. Rapid prototyping in PVS. Technical Report CR-2003-212418, NASA, 2003.

32 Henri Poincaré. *La science et l'hypothèse*. Flammarion, Paris, 1902.

33 Pierre Roux. Formal Proofs of Rounding Error Bounds - With Application to an Automatic Positive Definiteness Check. *J. Autom. Reasoning*, 57(2):135–156, 2016. doi:10.1007/s10817-015-9339-z.

34 Siegfried M. Rump. Verification of positive definiteness. *BIT Numerical Mathematics*, 46:433–452, 2006.

35 Siegfried M. Rump. Verification methods: Rigorous results using floating-point arithmetic. *Acta Numerica*, 19:287–449, 2010.

36 Siegfried M Rump, Takeshi Ogita, Yusuke Morikura, and Shin'ichi Oishi. Interval arithmetic with fixed rounding mode. *Nonlinear Theory and Its Applications, IEICE*, 7(3):362–373, 2016.

37 Siegfried M. Rump, Takeshi Ogita, and Shin'ichi Oishi. Accurate Floating-Point Summation Part I: Faithful Rounding. *SIAM J. Scientific Computing*, 31(1):189–224, 2008. doi:10.1137/050645671.

38 Siegfried M. Rump, Paul Zimmermann, Sylvie Boldo, and Guillaume Melquiond. Computing predecessor and successor in rounding to nearest. *BIT Numerical Mathematics*, 49(2):419–431, June 2009. doi:10.1007/s10543-009-0218-z.

39 Arnaud Spiwack. *Verified Computing in Homological Algebra. (Calculs vérifiés en algèbre homologique)*. PhD thesis, École Polytechnique, Palaiseau, France, 2011. URL: https://tel.archives-ouvertes.fr/pastel-00605836.

A Certificate-Based Approach to Formally Verified Approximations

Florent Bréhard
Plume and AriC teams, LIP, ENS de Lyon, Université de Lyon, Lyon, France
MAC team, LAAS-CNRS, Université de Toulouse, CNRS, Toulouse, France

Assia Mahboubi
Gallinette team, LS2N, INRIA, Université de Nantes, Nantes, France

Damien Pous
Plume team, LIP, CNRS, ENS de Lyon, Université de Lyon, Lyon, France

Abstract

We present a library to verify rigorous approximations of univariate functions on real numbers, with the Coq proof assistant. Based on interval arithmetic, this library also implements a technique of validation *a posteriori* based on the Banach fixed-point theorem. We illustrate this technique on the case of operations of division and square root. This library features a collection of abstract structures that organise the specfication of rigorous approximations, and modularise the related proofs. Finally, we provide an implementation of verified Chebyshev approximations, and we discuss a few examples of computations.

2012 ACM Subject Classification Theory of computation → Logic

Keywords and phrases approximation theory, Chebyshev polynomials, Banach fixed-point theorem, interval arithmetic, Coq

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.8

Related Version Appendix available on HAL at `https://hal.archives-ouvertes.fr/hal-02088529`.

Supplement Material `https://gitlab.inria.fr/amahboub/approx-models`

Funding This work has been funded by the European Research Council (ERC) under the European Union's Horizon 2020 programme (CoVeCe, grant agreement No 678157), and was supported by the LABEX MILYON (ANR-10-LABX-0070) of Université de Lyon, within the program "Investissements d'Avenir" (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR).
This work was supported in part by the project FastRelax ANR-14-CE25-0018-01.

1 Introduction

While numerical analysis offers sophisticated computational methods to solve various function space problems, the numerical errors caused by floating-point computations, discretisations or finite iterations, are a major concern in domains like safety-critical engineering or computer assisted proofs in mathematics. To address these issues, *rigorous numerics* [38] provides algorithms to compute validated enclosures of the exact solution. However, their correctness is ensured by pen-and-paper mathematical proofs. In particular, there is no guarantee concerning their *implementations*.

In this regard, formal proof offers the highest level of confidence. Several noteworthy works use formally proved rigorous numerics to completely formalise highly nontrivial mathematical results, like the Flyspeck project [19] for the Kepler conjecture or the formal verification [23] of the computer-aided proof of the Lorenz attractor [37]. However, those methods often require intensive computations, which rapidly becomes restrictive inside proof

assistants. In the context of formal verification, certificate-based methods is an appealing strategy [20, 13, 1]. It consists in discharging part of the computation work load to external oracles, while correctness remains guaranteed via *a posteriori* validation steps performed inside the proof assistant. This approach has mostly been used for the purpose of verifying symbolic computations, e.g. primality proofs [18], but we illustrate here how it can also by used in the context of rigorous numerical analysis.

Interval arithmetic. Invented in the 60s by Moore [33] and significantly developed in the 80s by Kulisch *et al.*, interval arithmetic is an essential building block of rigorous numerics. The key idea consists in using real intervals with representable endpoints (e.g., floating-point numbers) as rigorous enclosures of real numbers, and providing operations preserving correctness. For example, from $\pi \in [3.1415, 3.1416]$ and $e \in [2.7182, 2.7183]$, one obtains $\pi + e \in [3.1415, 3.1416] \oplus [2.7182, 2.7183] = [5.8597, 5.8599]$. Efficient implementations are available, as MPFI [34], IntLab [35], C-XSC [28], ARB [24]. The CoqInterval library [32] moreover provides a fully verified implementation inside the Coq proof assistant.

Rigorous Chebyshev approximations. Interval arithmetic is however not a panacea, and replacing all operations on real numbers by interval ones should always be considered with caution: the *dependency phenomenon* may lead to disastrous over-approximations. In such cases, *higher order methods* such as rigorous polynomial approximations (RPAs) are preferable. A pioneer work is that of Berz and Makino on *Taylor models* [4]. Those provide not only a polynomial, but also a *remainder* s.t. the latter contains the difference between the former and the represented function. Since then, efforts were made to clarify the definition of RPAs and extend them to other bases, in particular the Chebyshev basis [10, 26], due to their far better approximation properties than Taylor expansions [36].

On the formal proof side, the CoqInterval library includes an implementation of Taylor models called CoqApprox [31], allowing in particular for an automated rigorous evaluation procedure of definite integrals inside Coq [30]. Unfortunately, an equally accomplished equivalent with Chebyshev approximations does not exist now. Our contribution is a first step towards a formally proved counterpart of the popular Chebfun package [15] for Matlab.

Fixed-point based a posteriori validation. Some operations in function spaces admit straightforward *self-validating* algorithms by replacing all operations in $\mathbb{R}$ by interval ones. Unfortunately, more complicated operations (e.g., division, square root, differential equations) face several obstructions: the intervals may fail to give sufficiently tight enclosures, bounds for the remainders may be unknown, or only asymptotic, or depend on noneffective quantities.

In such cases, *a posteriori* validation techniques are an attractive alternative, widely used in rigorous numerics. They consist in reconstructing afterwards an error bound for a candidate approximation. Dating back from the works of Kantorovich about Newton's method, they gained prominence with the rise of modern computers and were applied to numerous functional analysis problems [27, 40, 39, 29]. Even more recently, those methods were used to compute RPAs for solutions of linear ODEs [2, 8]. Broadly speaking, the function of interest is characterised as a fixed-point of a contracting operator, from which an error bound is recovered thanks to the Banach fixed-point theorem [3, Thm. 2.1]. Such techniques are of special interest for formal verification, for they allow one to rely on efficient but untrusted external tools while keeping the trusted codebase small: it suffices to formalise the theory about contracting operators and provide means of computing with those operators.

Contributions and outline. We present a Coq library that makes it possible to compute rigorous Chebyshev approximations of functions on reals. We support basic operations like multiplication or integration in the standard way. For more complex operations like division and square root, we resort to *a posteriori* validation techniques, thus making a first step towards a potential cooperation between external numerical tools and Coq.

We use the interval arithmetic provided by CoqInterval, but we design our abstractions for RPAs from scratch: this allows us to experiment with different design choices, with more flexibility. We first describe the main lines of the hierarchy (Section 2): we rely on canonical structures to abstract over the concrete implementation details of interval arithmetic, and we use them to denote both real valued functions and their rigorous approximations. We also abstract away from the concrete basis for approximations, to work in the future with different bases, even non polynomial ones (e.g., Bessel functions). We provide instances for the monomial and Chebyshev bases, the latter being described in Section 3.

The main theorem we need to perform a posteriori validation is the Banach fixed-point theorem, whose formalisation is described in Section 4. We show in Section 5 how to apply this theorem to compute rigorous approximations for division and square root using Newton-like operators. We finally discuss the benefits of our approach on two examples (Section 6): RPAs for the absolute value function, and verified computation of integrals related to the second part of Hilbert's 16th problem.

2 Approximating real numbers and functions

Numerical errors come from the estimation of both *real numbers*, e.g. using floating-point numbers, and *real functions*, e.g. using polynomials. Rigorous estimations must take all these uncertainties into account. For this purpose, *interval arithmetic* provides an explicit enclosure and *rigorous polynomial approximations* attach an interval to a polynomial approximant, which bounds the method error on a given domain. Note that the coefficients of polynomial approximations are usually themselves obtained from evaluations of the function or of its derivatives, and therefore also subject to numerical errors. A formal library about rigorous approximation thus implements several variants of each operation, on real numbers, floats, intervals, mathematical functions, approximants, etc., whose relationships are made precise in the various layers of specifications. Our library features a small hierarchy of structures which formalises and organises the dependencies between these variants.

2.1 Reals and Intervals

At the bottom of the hierarchy, structure `Ops0` collects the operations available on reals, floats, intervals, but also on polynomials and rigorous approximations. It provides the signature of a ring structure, with symbols $+$, $-$, $*$, 1 and 0 shared by all instances thanks to Coq's system of canonical structures. Yet the ring equational theory is a priori only available for real numbers. These operations are also those trivially self-validating. A super-structure `Ops1` collects other operations required on data-structures used for scalars: reals, floats, interval endpoints, intervals, etc. They are not meant to be implemented on polynomial approximations.

```
Record Ops0 := {
 car:> Type;
 add: car → car → car;
 sub: car → car → car;
 mul: car → car → car;
 zer, one: car }.
```

```
Record Ops1 := {
 ops0:> Ops0;
 fromZ: Z → ops0;
 div: ops0 → ops0 → ops0;
 sqrt, cos, abs: ops0 → ops0;
 pi: ops0 }.
```

Structure `Rel0` specifies the relationship between the operations of `Ops0` on reals and those on intervals. The field `rel` is a relation between the two instances `C` and `D`, which share overloaded notations. The relation will eventually be instantiated with the containment relation between intervals and reals. When doing so, the requirements on the relation precisely correspond to the fact that interval operations properly approximate real operations. A record `Rel1` is defined in the very same way for `Ops1`.

```
Record Rel0 (C D: Ops0) := {
 rel:> C → D → Prop;
 radd: ∀ x y, rel x y → ∀ x' y', rel x' y' → rel (x+x') (y+y');
 rsub: ∀ x y, rel x y → ∀ x' y', rel x' y' → rel (x-x') (y-y');
 rmul: ∀ x y, rel x y → ∀ x' y', rel x' y' → rel (x*x') (y*y');
 rzer: rel 0 0;
 rone: rel 1 1 }.
```

As much as possible, we will work with polymorphic functions like the following one:

```
Definition f (C: Ops1) (x: C): C := 1 / (1 + sqrt x).
```

First of all, this allows us to define at once a function on real numbers (here, $x \mapsto \frac{1}{1+\sqrt{x}}$) and a function on intervals, whatever the implementation of intervals. Second, and even more importantly, the corresponding approximation correctness theorem will always hold—by a parametricity meta-result, such a function f will always satisfy the following lemma:

```
Lemma rf: ∀ C D (T: Rel1 C D), ∀ x y, T x y → T (f x) (f y).
```

This is only a meta-result: we need to provide a proof for each function f; but the proof is always trivial, and we automatise it.

There are however operations which cannot be implemented at this level of abstraction, even if we were to add some operations to the record `Ops1`. This is typically the case for division and square root of rigorous approximations, which require operations on intervals that do not make sense on real numbers (e.g., computing the range of a function and checking that it is bounded). In order to define those operations while remaining rather agnostic about the choice of interval implementation, we setup an intermediate layer of abstraction using the structure `NBH` (for neighbourhood):

```
Record NBH := {
 II:> Ops1;                (* abstract intervals *)
 contains: Rel1 II ROps1;  (* containment relation; ROps1 is the Ops1 instance on R *)
 convex: ∀ Z x y, contains Z x → contains Z y → ∀ z, x≤z≤y → contains Z z };
 (* additional operations on intervals *)
 bnd: II → II → II;        (* directed convex hull *)
 is_lt: II → II → bool;    (* strict above test *)
 min,max: II → option II; (* min, max, if any *)
 bot: II;                  (* uninformative, contains all reals *)
 (* specification of the above operations *)
 bndE: ∀ X x, contains X x → ∀ Y y, contains Y y → ∀ z, x≤z≤y →contains (bnd X Y) z;
 is_ltE: ∀ X Y, wreflect (∀ x y, contains X x → contains Y y → x<y) (is_lt X Y);
 minE, maxE, botE: ... }.
```

We will also make use of the two following derived operations:

```
Definition mag (N: NBH) (X: II): option II := max (abs X).
Definition sym (N: NBH) (X: II): II := let X := abs X in bnd (-X) X.
```

The first one approximates the magnitude as an interval, if possible; the second one returns an interval centered in 0 that contains the argument. Note that we assume that intervals are convex. We provide an instance of this structure using the CoqInterval library, using intervals of floating point numbers from the Flocq library [6]. It is actually a family of instances indexed by the desired precision.

2.2 Abstract functions

The structure `FunOps` describes inductively the catalogue of expressions that the library can approximate.

```
Record FunOps (C: Type) := {
 funcar:> Ops0;  (* abstract type for functions, and pointwise basic operations *)
 id: funcar;
 cst: C → funcar;
 eval: funcar → C → C;
 integrate: funcar → C → C → C;
 div': nat → funcar → funcar → funcar;
 sqrt': nat → funcar → funcar }.
```

It is parameterised by a type `C` of ground values (typically, reals or intervals); it packages a set of basic operations on some abstract type for functions (pointwise addition, multiplication…), together with operations specific to functions: identity, constant function, evaluation, integration. It also asks for division and square root operations; those have an additional argument which is used to pass parameters to the oracles used in the implementation of those operations (for now, the degree of the interpolants).

When `C` is `R`, the type of real numbers, this structure is instantiated with the standard operations on `R→R` (ignoring the extra parameters for division and square root); our main goal is to provide instances with intervals for `C`, with which it is possible to perform computations.

Like for ground values, the structure `FunOps` makes it possible to write polymorphic functions like:

```
Definition g (C: Ops1) (F: FunOps C): F :=
  let f: F := div' 33 1 (1 + sqrt' 33 id ) in
  let a: C := integrate f 0 1 in
  pi + id * cst a
```

Such a declaration defines at the same time a function on reals ($x \mapsto \pi + x \int_0^1 \frac{\mathrm{d}t}{1+\sqrt{t}}$) and approximations of it, which will be obvious to prove correct whenever the chosen instance `F` satisfies appropriate properties. Those instances are obtained using *rigorous approximations*.

2.3 Rigorous Approximations

Approximating a function usually consists in projecting this function onto a finite dimension vector space, by expansion on a basis with appropriate properties. For instance, so-called Taylor models [4], are an instance of *rigorous polynomial approximation*. They attach an interval bounding the remainder to a certain polynomial, in this case represented in monomial basis, so as to describe a set of functions containing the one to be approximated. More generally in this section, a *rigorous approximation* refers to a linear combination of elements in a suitable basis, packaged with an interval remainder. In the code, we will also use the shorter term *model*, by analogy with Taylor models.

A basis is described by a family of functions, non necessarily polynomials, indexed by natural numbers, that is a term `T: nat→R→R`. The structure `BasisOps_on` below describes the signature required on a basis `T`. It is parameterised by the type `C` of coefficients; sequences of such coefficients (`seq C`) represent linear combinations of elements of `T`. Linear operations $(+, -, 0)$ need not be provided since they can be implemented independently from the basis. The range operation is important: its role is to bound the range on the given domain; it should be as accurate as possible since it is used at many places to compute error bounds in rigorous approximations (e.g., for multiplication and a posteriori validation). We define `BasisOps` to be a polymorphic function so that we capture with a single object the idealised operations on reals and their concrete implementation with intervals.

```
Record BasisOps_on (C: Type) := {
  lo, hi: C;                      (* bounds for the domain *)
  beval: seq C → C → C;           (* (efficient) evaluation *)
  bmul: seq C → seq C → seq C;    (* multiplication *)
  bone, bid: seq C;               (* constant to 1, identity *)
  bprim: seq C → seq C; }.        (* primitive *)
  brange: seq C → C*C; }.         (* range *)
Definition BasisOps := ∀ C: Ops1, BasisOps_on C.
```

Given such operations, we equip type `seq C` with the basic operations in `Ops0`. Then we can define rigorous approximations:

```
Record Model C := { pol: seq C; rem: C }.
```

Like with `seq C`, we equip `Model C` with the basic operations in `Ops0`, and then with those from `FunOps`. For instance, addition, evaluation and integration are defined as follows:

```
Definition madd (C: Ops1) (M N: Model C): Model C :=
  {| pol := pol M + pol N; rem := rem M + rem N |}.
Definition meval (C: Ops1) (M: Model C) (X: C): C := beval (pol M) X + rem M.
Definition mintegrate (C: Ops1) (M: Model C) (a b: C): C :=
   let N := bprim (pol M) in beval N b - beval N a + (b-a)*rem M.
```

For those relatively simple operations, it suffices to have the basic operations (`Ops1`) on `C`. For other operations like the range of a model, we actually need the additional operations on intervals provided by the structure `NBH`:

```
Definition mrange (N: NBH) (M: Model II) :=
  let (a,b) := brange (pol M) in bnd a b + rem M.
```

This is also the case for division and square root, which we will discuss in Section 5. All in all, we obtain instances `FunOps` through a construction of the following type:

```
Canonical Structure MFunOps (N: NBH) (B: BasisOps): FunOps II.
  (* with carrier [Model II] *)
```

It finally remains to show that those operations defined on rigorous approximations properly match the idealised operations on functions over reals. We fix in the sequel an instance `N: NBH` of neighbourhood and basis operations `B: BasisOps`, and we write `Model` for `Model II`). The central definition to establish this correspondence is the following one, where the function `eval` is the obvious evaluation function for linear combinations of elements of `T`.

```
Definition mcontains (F: Model) (f: R → R) :=
 ∃ p: seq R, scontains (pol F) p /\ ∀ x, lo≤x≤hi → contains (rem F) (f x - eval T p x)
```

Intuitively, a model contains a real-valued function f if it contains a generalised polynomial which is close enough to f on the domain of the basis. (The binary predicate `scontains` denotes the pointwise extension of the relation `contains` to sequences: in the definition, the real coefficients of `p` should be pointwise contained in the interval coefficients of `pol F`.)

Equipped with this definition, we prove lemmas like:

```
Lemma rmmul: ∀ F f G g,
   mcontains F f → mcontains G g → mcontains (F*G) (f*g).
Lemma rmdiv: ∀ n F f G g,
   mcontains F f → mcontains G g → mcontains (div' n F G) (div' n f g).
Lemma rmintegrate: ∀ F f A a B b,
   (∀ x, lo≤x≤hi → continuous_at f x) → lo≤a≤hi → lo≤b≤hi →
   mcontains F f → contains A a → contains B b →
   contains (integrate F A B) (integrate f a b).
```

Of course, we need assumptions on the basis operations in order to do so. Those assumptions are summarised in the following structure. Recall that a `B: BasisOps` provides us with operations `B ROps1` on reals and operations `B II` on intervals. The structure assumes: 1) the expected properties on the operations on reals, i.e, efficient evaluation corresponds to evaluation with `T`, multiplication indeed corresponds to pointwise multiplication under evaluation, etc.; and 2) a relationship between the operations on reals and on intervals. This separation of concerns is very convenient: the latter containment lemmas are always proved in a trivial way (i.e., automatically), and the former properties do not involve intervals at all, but only real numbers and functions, for which usual mathematical intuitions apply.

```
Record ValidBasisOps (N: NBH) (B: BasisOps) := {
 (* properties of operations on reals (B ROps1) *)
 lohi: lo < hi;
 bevalE: ∀ p x, beval p x = eval T p x;
 eval_cont: ∀ p x, continuity_pt (eval T p) x;
 eval_mul: ∀ p q x, eval T (bmul p q) x = eval T p x * eval T q x;
 eval_prim: ∀ p a b, eval T (bprim p) b - eval T (bprim p) a = RInt (eval T p) a b;
 ...
 (* relationship between operations on intervals (B II) and on reals (B ROps1) *)
 rbeval: ∀ P p X x, scontains P p → contains X x → contains (beval P X) (beval p x);
 rbmul: ∀ P p Q q, scontains P p → scontains Q q → scontains (bmul P Q) (bmul p q);
 rbprim: ∀ P p, scontains P p → scontains (bprim P) (bprim p);
 ... }.
```

3 Arithmetic on Chebyshev polynomials

In order to use the previously described rigorous approximations, it remains to provide implementation of operations (`BasisOps`) for certain families `T` of functions. We provide two instances of them: one for the standard monomial basis, where `T n x = x^n`, and one described in this section for Chebyshev basis, where `T n` is the n-th Chebyshev polynomial.

Chebyshev polynomials are defined by the following recurrence, which immediately translates to a recursive definition in Coq.

$$T_0 = 1 \qquad T_1 = X \qquad T_{n+2} = 2XT_{n+1} - T_n$$

We can then prove simple properties of those polynomials, for instance:

$$T_nT_m = (T_{n+m} + T_{m-n})/2 \qquad (n \leq m) \tag{1}$$

$$T_0 = T_1' \qquad T_1 = \frac{T_2'}{4} \qquad T_{n+3} = \frac{T_{n+3}'}{2(n+3)} - \frac{T_{n+1}'}{2(n+1)} \tag{2}$$

$$T_n(\cos t) = \cos(nt) \tag{3}$$

Those are proved in a few lines using existing lemmas about derivation and cosine.

3.1 Clenshaw's evaluation algorithm

The first operation we must implement for `BasisOps` is the evaluation function (`beval`). This operation should be polymorphic and as efficient as possible: it will be executed repeatedly when constructing and using rigorous approximations. We use Horner evaluation scheme for the monomial basis, and Clenshaw's algorithm [16] for Chebyshev, which are both linear in the number of elementary operations. The latter is usually presented as a dynamic programming routine. We fix abstract operations `C: Ops1` for the remaining Coq snippets in this section, and we translate this routine into a recursive function with two accumulators:

```
Fixpoint Clenshaw b c (p: seq C) x :=
  match p with
  | [] => c - x*b
  | a::q => Clenshaw c (a + 2*x*c - b) q x
  end.
Definition beval (p: seq C) x := Clenshaw 0 0 (rev p) x.
```

This code might look mysterious; it is justified by the following invariant on real numbers:

```
Lemma ClenshawR b c p x: Clenshaw b c p x = eval T (catrev p [c - 2*x*b; b]) x.
```

In the right-hand side, `catrev` is the function that reverses its first argument and catenate it with the second one. The proof is done by induction in just three lines, using the Coq tactic for ring equations [17]. Correctness (i.e., field `bevalE` from structure `ValidBasisOps`) follows.

Note that while the definition of `beval` can be used with any `Ops1` structure, its correctness is proved only on reals: the lemma `ClenshawR` does not hold in every `Ops1` structure. The behaviour of `beval` on those structures is specified only through the fact that it respects containments (field `rbeval` from structure `ValidBasisOps`, which is proved automatically.)

3.2 Multiplication

Another important operation is multiplication. Again, this operation should be polymorphic, and efficient. A difficulty here is that due to Equation (1), the n-th coefficient of a multiplication potentially depends on all coefficients of its arguments, not only on the coefficient of smaller rank. We use the following definition, with two auxiliary recursive functions:

```
Fixpoint mul_pls (p q: seq C): seq C :=
  match p,q with
  | [],_ | _,[] => []
  | a::p', b::q' => sadd (a*b::(sadd (sscal a q') (sscal b p'))) (0::0::mul_pls p' q')
  end.
Fixpoint mul_mns (p q: seq C): seq C :=
  match p,q with
  | [],_ | _,[] => []
  | a::p', b::q' => sadd (a*b::(sadd (sscal a q') (sscal b p'))) (mul_mns p' q')
  end.
Definition smul (p q: seq C): seq C := sscal (1/2) (sadd (mul_mns p q) (mul_pls p q))
```

where `sscal` is the scalar multiplication for polynomials – we cannot yet use the standard notation for this operation since we are in the process of defining an `Ops0` structure on `seq C`. The function `mul_pls` actually corresponds to multiplication in the monomial basis, it covers the first summand in the right-hand side of (1). The function `mul_mns` differs only in the fact that the recursive call is not pushed away using two "cons" operations; it covers the second summand in the right-hand side of (1). Like previously, that `smul` preserves containments (field `rbmul` of structure `ValidBasisOps`) is obvious: this operation only performs a finite sequence of operations preserving containments. Proving that it behaves correctly on reals numbers is more interesting; the key invariant is the following one:

```
Lemma eval_mul_: ∀ (p q: seq R) n x,
  eval_ n p x * eval_ n q x = (eval (mul_mns p q) x + eval_ (n+n) (mul_pls p q) x)/2.
```

Here, `eval_ n p` evaluates `P` padded with n zeros in front of it. Again, the difficulty is to find the lemma: it is proved in six lines using (1), and correctness of `smul` on reals immediately follows. Taking primitives in Chebyshev basis follows the same pattern (see [12, Appendix A]).

3.3 Range

As mentioned above, we need accurate estimations of the range of a given polynomial in order to be able to compute precise rigorous approximations. This range can always be estimated by evaluating the polynomial on the interval representing the domain (i.e., given `p: seq C`, compute `beval p (bnd lo hi)`). This technique is however not sufficient in practice: this tends to produce largely over-estimated bounds. With Chebyshev basis we can proceed differently: indeed, thanks to Equation (3), T_n ranges over $[-1;1]$ on $[-1;1]$. Therefore, the range of a polynomial on $[-1;1]$ can be estimated by using the sum of the absolute values of the coefficients in Chebyshev basis (and actually, we do not need to take the absolute value of the first coefficient since $T_0 = 1$).

```
Definition range_: seq C → C := foldr (fun A X => abs A + X) 0.
Definition range (P: seq C): C*C :=
  match p with
  | [] => (0,0)
  | A::Q => let R := range_ Q in (A-R,A+R)
  end.
```

3.4 Rescaling

Putting everything together, we obtain the polymorphic operations `chebyshev.basis: BasisOps`, which can readily be used to construct rigorous approximations, with the instance `MFunOps` from Section 2.3. This basis however requires to work on the domain $[-1;1]$ (for estimating the range as explained in the previous section, but also to perform interpolation, see Section 5.1). In order to use it on other domains, we provide a rescaling function that takes a `B: BasisOps` and rescales it to a given interval $[a;b]$ using the obvious affine function. We show that this operation preserves validity of basis operations, so that we can use it whenever needed.

4 Formalisation of Banach fixed-point theorem

Banach fixed-point theorem is the cornerstone of the method discussed here.

▶ **Theorem 1** (Banach fixed-point). *Let $(X, \|\cdot\|)$ be a Banach space, an operator $\boldsymbol{F} : X \to X$, $h^\circ \in X$, and $\mu, b, r \in \mathbb{R}_+$, satisfying the following conditions:*

(i) $\|h^\circ - \boldsymbol{F} \cdot h^\circ\| \;\leq\; b$;
(ii) *$\boldsymbol{F}$ is μ-Lipschitz over the closed ball $\overline{B}(h^\circ, r) := \{h \in X \mid \|h - h^\circ\| \leq r\}$:*

$$\forall h_1, h_2 \in X, \quad h_1 \in \overline{B}(h^\circ, r) \wedge h_2 \in \overline{B}(h^\circ, r) \;\Rightarrow\; \|\boldsymbol{F} \cdot h_1 - \boldsymbol{F} \cdot h_2\| \;\leq\; \mu \|h_1 - h_2\|;$$

(iii) $\mu < 1$ *: $\boldsymbol{F}$ is contracting over $\overline{B}(h^\circ, r)$;*
(iv) $b + \mu r \;\leq\; r$.

Then $\boldsymbol{F}$ admits a unique fixed-point h^* in $\overline{B}(h^\circ, r)$.

This classic result has been formalised in various flavours of logic and proof assistants. In particular, Boldo et al. have provided a formal proof of a version of this fixed-point theorem, based on the Coquelicot library, for the purpose of the formalisation of the Lax-Milgram theorem [5]. Using the same backbone library, we formalise an alternative version of the theorem: our version is significantly more concise, and closer to the computational content of the result. We describe below this formalisation.

The Coquelicot library formalises topological concepts using *filters* [7, 21], which we briefly recall here. A filter on a type T is a collection of collections of inhabitants of T which is non-empty, upward closed and stable under finite intersections:

```
Record Filter (T : Type) (F : (T → Prop) → Prop) := {
  filter_true : F (fun _ => True) ;
  filter_and : ∀ P Q : T → Prop, F P → F Q → F (fun x => P x /\ Q x) ;
  filter_imp : ∀ P Q : T → Prop, (∀ x, P x → Q x) → F P → F Q }.
```

While filters are used to formalise neighbourhoods, *balls* allow for expressing the relative closeness of points in the space. Balls are formalised using a ternary relation between two points in the carrier type, and a real number, with the following axioms:

```
  ball : M → R → M → Prop ;
  ax1 : ∀ x (e > 0), ball x e x ;
  ax2 : ∀ x y e, ball x e y → ball y e x ;
  ax3 : ∀ x y z e1 e2, ball x e1 y → ball y e2 z → ball x (e1 + e2) z
```

Two points are called *close* when they cannot be separated by balls:

```
Definition close (x y : M) : Prop := ∀ eps > 0, ball x eps y.
```

A filter is called a *Cauchy filter* when it contains balls of arbitrary (small) radius:

```
Definition cauchy (T : UniformSpace) (F : (T → Prop) → Prop) :=
  ∀ eps > 0, ∃ x, F (ball x eps).
```

Finally, a *uniform space* is a type equipped with a ball relation and a *complete space* moreover has a limit operation on filters, which ensures the convergence of Cauchy sequences via the following axioms (where `ProperFilter F` is equivalent to `Filter F /\ ∀ P, F P → ∃ x, P x`):

```
  lim : ((T → Prop) → Prop) → T ;
  ax1 : ∀ F, ProperFilter F → cauchy F → ∀ eps > 0, F (ball (lim F) eps) ;
  ax2 : ∀ F1 F2, F1 ⊆  F2 → F2 ⊆ F1 → close (lim F1) (lim F2)
```

The above formal definition of balls does not enforce closedness nor openness. We thus introduced the relation associated with the *closure* of balls, so as to model *closed* neighbourhoods:

```
Definition cball x r y := ∀ e > 0, ball x (r+e) y.
```

Equipped with this definition, hypothesis (ii) of Theorem 1 is formalised as follows:

```
Definition lipschitz_on (F : U → U) (mu : R) (P : U → Prop) :=
∀ x y : U, ∀ r ≥ 0, P x → P y → cball x r y → cball (F x) (mu*r) (F y).
```

We now sketch our formalised proof, using mathematical notations. We consider a complete space X and we write $y \in B(x,r)$ for the formal `(ball x r y)`, and $y \in \overline{B}(x,r)$ for `(cball x r y)`. The key notion is that of *strongly stable ball* (see Figure 1):

▶ **Definition 2** (Strongly stable ball). *A ball $\overline{B}(u,r)$ is μ-*strongly stable *for $\boldsymbol{F}$ if $\boldsymbol{F}$ is μ-Lipschitz on $\overline{B}(u,r)$ and if there is a non-negative real number s, called the* offset*, s.t.:*

$$\boldsymbol{F} \cdot u \in \overline{B}(u,r) \qquad \textit{and} \qquad s + \mu r \ \leq \ r.$$

▶ Remark 3 (Stability). For any x in $\overline{B}(u,r)$, a strongly stable ball for $\boldsymbol{F}$, $\boldsymbol{F} \cdot x \in \overline{B}(u,r)$.

▶ Remark 4 (Contracting case). When $0 \leq \mu < 1$, for any μ-strongly stable ball $\overline{B}(v,\rho)$, with offset σ, $\overline{B}(\boldsymbol{F} \cdot v, \mu\rho)$ is also a strongly stable ball, with offset $\mu\sigma$. Moreover, $\overline{B}(\boldsymbol{F} \cdot v, \mu\rho)$ is included in $\overline{B}(v,\rho)$.

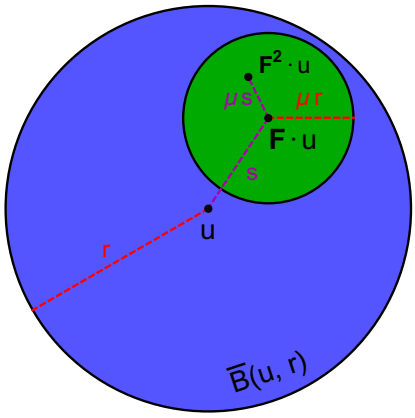


Figure 1 Balls $\overline{B}_0$ and $\overline{B}_1$.

Assume that $\boldsymbol{F}$ has a μ-strongly stable ball $\overline{B}(u,r)$ of offset s, with $\mu < 1$. In particular, $\boldsymbol{F}$ is contracting on $\overline{B}(u,r)$. Consider the sequence of balls defined as follows:

$$\overline{B}_n = \overline{B}(u_n, r_n) \quad \text{with} \quad u_n = \boldsymbol{F}^n \cdot u \quad \text{and} \quad r_n = r\mu^n$$

where $\boldsymbol{F}^n \cdot u$ denotes the iterated images of u under $\boldsymbol{F}$. By Remark 4, $(\overline{B}_n)_{n\in\mathbb{N}}$ is a nested sequence of μ-strongly stable ball for $\boldsymbol{F}$, with offset $s\mu^n$. Let $\mathcal{F}$ be the family of collections of points in U defined as:

$$\mathcal{F} = \{P \subseteq U \mid \exists\, n, \overline{B}_n \subseteq P\}.$$

It is a proper filter: $\mathcal{F}$ contains U, it is obviously upward closed, and for $P, Q \in \mathcal{F}$, $P \cap Q$ is also in $\mathcal{F}$ because $(\overline{B}_n)_{n\in\mathbb{N}}$ is decreasing for inclusion. Thus $\mathcal{F}$ has a limit w, such that for any $\varepsilon > 0$, balls $\overline{B}_n$ are eventually included in $B(w,\varepsilon)$. We provide a formal proof of Theorem 5, a reformulation of Theorem 1 using the vocabulary of the Coquelicot library:

▶ **Theorem 5.** *The limit w of the filter $\mathcal{F}$ is in $\overline{B}_0$, and w is a fixed point of $\boldsymbol{F}$. Moreover, w is close to every other fixed point of $\boldsymbol{F}$ in $\overline{B}_0$.*

Proof. In this statement "w is a fixed point of $\boldsymbol{F}$" means "w is close to $\boldsymbol{F}\cdot w$". First, $w \in \overline{B}_n$ for all n. Indeed, for any $\varepsilon > 0$, there is an $m \geq n$ s.t. $\overline{B}_m \subseteq B(w,\varepsilon)$, and since $\overline{B}_m \subseteq \overline{B}_n$, $u_m \in \overline{B}_n \cap B(w,\varepsilon)$. In particular, $w \in \overline{B}_0$. It is also clear by stability that $\boldsymbol{F}\cdot w \in \overline{B}_n$ for all n. Moreover, w is close to any point v s.t. $v \in \overline{B}_n$ for all n (for any $\varepsilon > 0$, choose n s.t. $2\mu r^n < \varepsilon$). Taking $v := \boldsymbol{F}\cdot w$ proves that w is a fixed point of $\boldsymbol{F}$.

Finally, if $w' \in \overline{B}_0$ is another fixed point of $\boldsymbol{F}$, then it follows from an easy induction that $w' \in \overline{B}_n$ for all n. Hence, the foregoing shows that w is close to w'. ◀

Strongly stable balls model the requirements set on the untrusted data to be formally verified. They can also be seen as balls centered at the initial point, and large enough to include all its successive iterates, i.e. as instances of the locus at stake in classical presentations of the proof. The version proved by Boldo et al. has a slightly more technical wording, which seems to be made necessary by its further usage in the verification of the Lax-Milgram theorem. Our proof script is significantly shorter, partly because we automate proofs of positivity conditions (for radii of balls) using canonical structures for manifestly positive expressions. But the key ingredient for concision is to make most of the filter device in the proof, and to refrain from resorting to low-level properties of geometric sequences. To

the best of our knowledge, the other libraries of formalised analysis featuring a proof of this result, notably Isabelle/HOL and HOL-Light, are based on variant of proof strategy closer to the approach of Boldo et al. than to ours.

5 Newton-like validation operators

The purpose of this section is twofold. We first present the general principle of fixed-point based *a posteriori* validation methods, and more particularly, the use of Newton-like validation operators. Then we apply it to the division and square root of models.

Throughout this section, let $(X, \|\cdot\|)$ denote a Banach space, and h^* the exact solution of an equation in X. In this article, X stands for the space $\mathcal{C}(I)$ of real-valued continuous functions defined over a compact segment $I = [a, b]$, with the uniform norm $\|h\| := \sup_{x\in I} |h(x)|$. The division and square root of functions are simple examples of solutions of equations in $\mathcal{C}(I)$, but there are also differential equations, integral equations, delay equation, etc. The general scheme for Banach fixed-point based *a posteriori* validation methods follows two steps:

1. **Approximation step.** A numerical approximation $h^\circ \in X$ of h^* is obtained by an oracle, which may resort to any approximation method. In particular, this step requires no mathematical assumption and can be executed purely numerically outside the proof assistant, good approximation properties being only desirable for efficiency. In our setting with $X = \mathcal{C}(I)$, interpolation at Chebyshev nodes (Section 5.1) is an efficient and accurate oracle for a wide range of function space problems.
2. **Validation step.** The initial problem is rephrased in such a way that h^* is a fixed point of a (locally) contracting operator $\boldsymbol{F} : X \to X$. An *a posteriori* error bound on $\|h^\circ - h^*\|$ is deduced from the Banach fixed-point theorem (Theorem 1).

We thus need to find a contracting operator $\boldsymbol{F}$ of which h^* is a fixed point. To this end, we use Newton-like validation methods, which transform an equation $\boldsymbol{M} \cdot h = 0$ into an equivalent fixed-point equation $\boldsymbol{F} \cdot h = h$ with $\boldsymbol{F}$ contracting. More specifically, suppose that $\boldsymbol{M} : X \to Y$ is differentiable; we use a Newton-like operator $\boldsymbol{F} : X \to X$ defined as:

$$\boldsymbol{F} \cdot h = h - \boldsymbol{A} \cdot \boldsymbol{M} \cdot h, \qquad h \in X,$$

with $\boldsymbol{A} : Y \to X$ an *injective bounded linear* operator, intended to be close to $(\mathcal{D}\boldsymbol{M}_{h^\circ})^{-1}$. The operator $\boldsymbol{A}$ may be given by an oracle and does not need to be this exact inverse, which anyway might be non representable on computers exactly. The mean value theorem yields a Lipschitz ratio μ for $\boldsymbol{F}$ over any convex subset S of X:

$$\forall h_1, h_2 \in S,\ \|\boldsymbol{F}\cdot h_1 - \boldsymbol{F}\cdot h_2\| \le \mu \|h_1 - h_2\|, \quad \text{with } \mu = \sup_{h\in S} \|\mathcal{D}\boldsymbol{F}_h\| = \sup_{h\in S} \|\mathbf{1}_X - \boldsymbol{A}\cdot\mathcal{D}\boldsymbol{M}_h\|,$$

which is expected to be small over some neighbourhood of h°.

Concretely, in order to apply Theorem 1, one needs to compute the following quantities:

- a bound $b \ge \|\boldsymbol{A} \cdot \boldsymbol{M} \cdot h^\circ\| = \|h^\circ - \boldsymbol{F} \cdot h^\circ\|$;
- a bound $\mu_0 \ge \|\mathbf{1}_X - \boldsymbol{A} \cdot \mathcal{D}\boldsymbol{M}_{h^\circ}\| = \|\mathcal{D}\boldsymbol{F}_{h^\circ}\|$;
- a bound $\mu'(r) \ge \|\boldsymbol{A} \cdot (\mathcal{D}\boldsymbol{M}_h - \mathcal{D}\boldsymbol{M}_{h^\circ})\| = \|\mathcal{D}\boldsymbol{F}_h - \mathcal{D}\boldsymbol{F}_{h^\circ}\|$ valid for any $h \in B(h^\circ, r)$, and parameterised by a radius $r \in \mathbb{R}_+$.

If we are able to find a radius $r \in \mathbb{R}_+$ satisfying:

$$\mu(r) := \mu_0 + \mu'(r) < 1, \qquad \text{and} \qquad b + r\mu(r) \ \le\ r, \tag{4}$$

then Theorem 1 guarantees the existence and uniqueness of a root h^* of $\boldsymbol{M}$ in $B(h^\circ, r)$.

▶ Remark 6. Finding an r as small as possible while satisfying (4) may be an nontrivial task for automated validation procedures. For many problems, $\mu'(r)$ is polynomial, hence conditions (4) are polynomial inequalities over r: this is called the *radii polynomial approach* [22] in rigorous numerics. In our case, division (resp. square root) induces an affine (resp. quadratic) equation, which admits closed form solutions.

5.1 Approximation step: interpolation

Since they are certified a posteriori, (non-rigorous) approximations for division and square root of given models can be obtained using arbitrary numerical techniques. We use interpolation at Chebyshev nodes of the second kind for its efficiency and excellent approximation properties [36].

Ideally, we would implement this operation outside of the proof assistant, in order not to pay the price of an interpreted language. This would however require a lot of work in order to design a proper interface between Coq values and external values (e.g., converting Coq representation of floating points numbers into machine level floating points, and back). Instead, and for now, we implement the oracles inside Coq, as unspecified functions. To this end, we add a field to the structure `BasisOps_on`, to compute interpolants of a given degree:

```
interpolate: nat → (C → C) → seq C;
```

We implement this operation for Chebyshev basis using the discrete orthogonality relations on Chebyshev polynomials.

To reduce the price of staying inside Coq for those computations, we exploit the polymorphism built in our framework to perform those computations on floating-point numbers rather than intervals. To this end, we add the following fields to the structure `NBH`:

```
FF: Ops1;          (* abstract type for floating points and their operations *)
I2F: II → FF;      (* conversion from intervals to floating points to (midpoint) *)
F2I: FF → II;      (* conversion from floating points to intervals (singleton) *)
```

Equipped with these operations, we can define conversion operations between models (on intervals) and polynomials with floating point coefficients:

```
Definition mcf (M: Model): seq FF := map I2F (pol M).
Definition mfc (p: seq FF): Model := {| pol := map F2I p; rem := 0 |}.
```

The field `FF`, of type `Ops1`, will make it possible to call the functions `interpolate` and `beval` from the basis with `C` as `FF`, i.e., to let them operate on floating point numbers. By doing so we do not have to reimplement Clenshaw's evaluation scheme on floating point numbers.

5.2 Validation step for division

For $f, g \in \mathcal{C}(I)$ with g nonvanishing over I, the quotient f/g is the unique root of $\boldsymbol{M} : h \mapsto gh - f$. Let h° be a candidate approximation given by the approximation step. Constructing the Newton-like operator $\boldsymbol{F}$ requires an approximation $\boldsymbol{A}$ of $(\mathcal{D}\boldsymbol{M}_{h^\circ})^{-1} : k \mapsto k/g$. For that purpose, suppose $w \approx 1/g \in \mathcal{C}(I)$ is also given by an oracle, and define:

$$\boldsymbol{F} \cdot h = h - w(gh - f). \tag{5}$$

The next proposition computes an upper bound for $\|h^\circ - f/g\|$; it is implemented in `div.newton`.

▶ **Proposition 7.** *Let $f, g, h^\circ, w \in \mathcal{C}(I)$, and $\mu, b \in \mathbb{R}_+$ such that:*

(7 i) $\|w(gh^\circ - f)\| \le b$, (7 ii) $\|1 - wg\| \le \mu$, (7 iii) $\mu < 1$.

Then g does not vanish over I and $\|h^\circ - f/g\| \le b/(1-\mu)$.

Proof. Conditions (7 ii) and (7 iii) imply that $\boldsymbol{F}$ (Equation (5)) is contracting over $\mathcal{C}(I)$ with ratio μ. The radius $r := \frac{b}{1-\mu}$ makes the ball $\overline{B}(h^\circ, r)$ strongly stable with offset b (7 i), since $b + \mu r = r$. Therefore, h^* is the (global) unique root of $\boldsymbol{M}$, and $\|h^\circ - h^*\| \le r$.

Finally, w and g do not vanish because $\|1 - wg\| \le \mu < 1$. Hence, $h^* = f/g$ over I. ◀

The concrete division of models is implemented as follows:

```
Definition mdiv_aux (F G H W: Model): Model :=
 let K1 := 1-W*G in
 let K2 := W*(G*H - F) in
 match mag (mrange K1), mag (mrange K2) with
 | Some mu, Some b when is_lt mu 1 => {| pol := pol H; rem := rem H + sym (b/(1-mu)) |}
 | _ => mbot
 end.
Definition mdiv n (F G: Model): Model :=
  let p, q := mcf F, mcf G in
  mdiv_aux F G (mfc (interpolate n (fun x => beval p x / beval q x)))
               (mfc (interpolate n (fun x => 1 / beval q x))).
```

Note that we use the trivial model `mbot={|pol:=[];rem:=bot|}` as a default value, when the concrete computations fail to validate the guess of the oracle (either because this guess is just wrong, or because of over-approximations in the computations). The correctness lemmas use the properties of operations on models to prove the assumptions of `div.newton`.

```
Lemma rmdiv_aux F f G g H h W w:
  mcontains F f → mcontains G g → mcontains H h → mcontains W w →
  mcontains (mdiv_aux F G H W) (f/g).
Lemma rmdiv n F f G g: mcontains F f → mcontains G g → mcontains (mdiv' n F G) (f/g).
```

5.3 Validation step for square root

Let $f \in \mathcal{C}(I)$ be strictly positive over I. The square root $\sqrt{f}$ is one of the two roots of the quadratic equation $\boldsymbol{M} \cdot h := h^2 - f = 0$ (the other being $-\sqrt{f}$). Let h° be a candidate approximation. Since $\mathcal{D}\boldsymbol{M}_h : k \mapsto 2hk$, one also needs an approximation $w \approx 1/(2h^\circ) \approx 1/(2\sqrt{f}) \in \mathcal{C}(I)$ in order to define $\boldsymbol{A} : k \mapsto wk$, approximating $(\mathcal{D}\boldsymbol{M}_{h^\circ})^{-1}$. Then:

$$\boldsymbol{F} : \quad h \quad \mapsto \quad h - w(h^2 - f). \tag{6}$$

The next proposition (implemented by `sqrt.newton`), computes an upper bound for $\|h^\circ - \sqrt{f}\|$.

▶ **Proposition 8.** *Let $f, h^\circ, w \in \mathcal{C}(I)$, $\mu_0, \mu_1, b \in \mathbb{R}_+$ and $t_0 \in I$ such that:*

(8 i) $\left\|w\left(h^{\circ 2} - f\right)\right\| \le b$, (8 ii) $\|1 - 2wh^\circ\| \le \mu_0$, (8 iii) $\|w\| \le \mu_1$,

(8 iv) $\mu_0 < 1$, (8 v) $(1-\mu_0)^2 - 8b\mu_1 \ge 0$, (8 vi) $w(t_0) > 0$.

Then $f > 0$ over I and $\left\|h^\circ - \sqrt{f}\right\| \le r^*$, where:

$$r^* := \frac{1 - \mu_0 - \sqrt{(1-\mu_0)^2 - 8b\mu_1}}{4\mu_1}.$$

Proof. First, since $\|1 - 2wh^\circ\| \leq \mu_0 < 1$ (by (8 ii) and (8 iv)) and $w(t_0) > 0$ (8 vi), w and h° are strictly positive over I, by continuity. Using (8 iii), $\mu_1 > 0$.

If $b = 0$, then $r^* = 0$ and $h^\circ = \sqrt{f}$ over I, because $w(h^{\circ 2} - f) = 0$ (8 i) and $w, h^\circ > 0$. Hence the conclusion holds.

From now on, we assume $b > 0$. $\boldsymbol{F}$ is Lipschitz of ratio $\mu(r) := \mu_0 + 2\mu_1 r$ over $\overline{B}(h^\circ, r)$ for any $r \in \mathbb{R}_+$, because:

$$\boldsymbol{F}\cdot h_1 - \boldsymbol{F}\cdot h_2 = (h_1 - h_2) - w(h_1^2 - h_2^2) = [(1 - 2wh^\circ) + w(h^\circ - h_1) + w(h^\circ - h_2)]\,(h_1 - h_2).$$

Therefore, satisfying $b + \mu(r)r \leq r$ is equivalent to the quadratic inequality:

$$2\mu_1 r^2 + (\mu_0 - 1)r + b \leq 0. \tag{7}$$

Condition (8 v) implies that (7) admits solutions, and r^* is the smallest one. Moreover, since $b, \mu_1 > 0$, we get $r^* > 0$, so that $b + \mu(r^*)r^* = r^*$ also implies $\mu(r^*) < 1$.

Now, all the assumptions of Theorem 1 are fulfilled. Hence, $\boldsymbol{F}$ has a unique fixed point h^* in $\overline{B}(h^\circ, r^*)$. To obtain $h^* = \sqrt{f}$ over I, it remains to show that $h^* > 0$. This follows from $w > 0$ and:

$$\|1 - 2wh^*\| \leq \|1 - 2wh^\circ\| + \|2w(h^* - h^\circ)\| \leq \mu_0 + 2\mu_1 r^* = \mu(r^*) < 1. \qquad \blacktriangleleft$$

▶ Remark 9. Contrary to the case of division where continuity was not needed at all, it is here used for w. Therefore, `sqrt.newton` requires w to be continuous over I.

The Coq code for the corresponding operations on models `msqrt_aux` and `msqrt`, together with the statements of their correctness lemmas, are given in [12, Appendix B].

6 Examples

6.1 Playing with approximations of the absolute value function

Consider the function $f_\varepsilon : x \mapsto \sqrt{\varepsilon + x^2}$ over $[-1, 1]$, with $\varepsilon > 0$. When $\varepsilon \to 0$, f_ε converges uniformly to the absolute value function $x \mapsto |x|$ (which is not analytic at 0), with:

$$|f(x) - |x|| = \left|\sqrt{\varepsilon + x^2} - \sqrt{x^2}\right| = \left|\frac{\varepsilon}{\sqrt{\varepsilon + x^2} + \sqrt{x^2}}\right| \leq \sqrt{\varepsilon}. \tag{8}$$

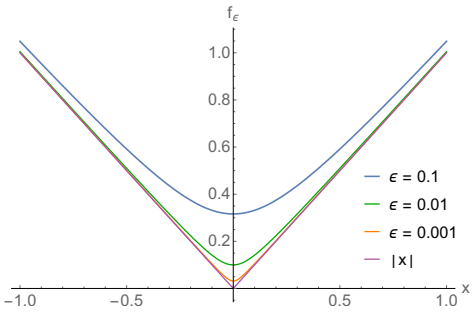
Rigorous uniform approximations. Approximating f_ε with polynomials becomes harder for small ε, due to the complex singularities $\pm i\sqrt{\varepsilon}$ getting closer to the interval $[-1, 1]$. Nevertheless, Chebyshev interpolation still works and our implementation computes rigorous approximations as accurate as desired (see Figure 2b), of exponential convergence with ratio determined by ε. Note that for too small degree, the computed approximation of the square root is too far from the solution, and the a posteriori validation returns an infinite remainder.

In order to provide a comparison with CoqApprox's Taylor models, we used the tactic `interval with (i_depth 1, i_bisect_taylor x N, i_prec p)` to build a Taylor model of degree N with precision p. Timings given in Table 2c reveal a significant advantage of our implementation (there we use $\varepsilon = 2$ to avoid convergence issues of Taylor models). Concerning accuracy, our experiments tend to show that when $\varepsilon \leq 1$, CoqApprox fails to compute *converging* Taylor models. Indeed, even with large L, a goal like:

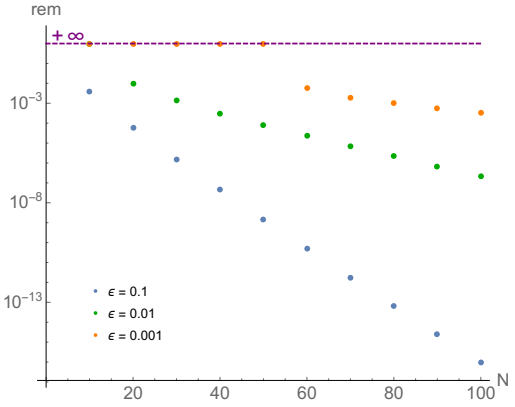
```
Goal Fail : ∀ x : R, -1 ≤ x ≤ 1 → sqrt (1/100+x*x) ≤ L
```

is not solved when the degree N becomes too large, probably indicating that the Taylor models *diverge* due to complex singularities inside the unit disk. (Note that the `interval` tactic can solve this goal, but only by resorting to subdivision techniques.)

Error bounding. We want to bound $|f_\varepsilon(x) - |x||$ for $x \in [-1, 1]$ without making use of any symbolic manipulation like (8). At first glance, one can choose to use the rigorous approximations over $[-1, 1]$ obtained previously, and evaluate $f_\varepsilon(x) - x$ (resp. $f_\varepsilon(x) + x$) over $[0, 1]$ (resp. $[-1, 0]$) using Clenshaw's algorithm. However, even if the approximations are quite good, this evaluation strategy gives huge overestimations because $[0, 1]$ and $[-1, 0]$ are not small intervals. Instead, we compute separately two approximations for f_ε: one over $[0, 1]$ and one over $[-1, 0]$, and we evaluate $f_\varepsilon(x) - x$ (resp. $f_\varepsilon(x) + x$) over $[1, 0]$ (resp. $[-1, 0]$) using the Chebyshev `range` function. This approach yields bounds that are rather close to the optimal $\sqrt{\varepsilon}$ (see Figure 2d). However, this does not allow for arbitrary accuracy: a subdivision procedure would be necessary here.



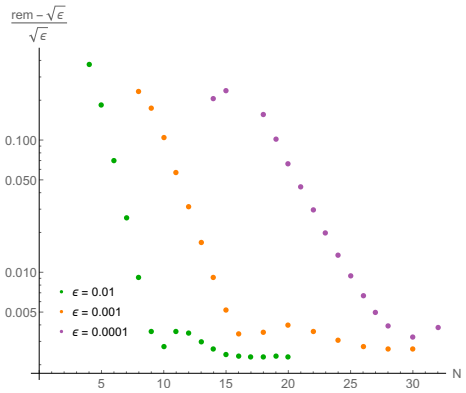
(a) Functions f_ε and $x \mapsto |x|$ over $[-1, 1]$.



(b) Magnitude of the remainders of degree-N Chebyshev models approximating f_ε over $[-1, 1]$.

N	time (in seconds)	
	Chebyshev	CoqApprox
10	0.11	0.10
20	0.16	0.12
30	0.22	0.14
40	0.31	0.20
50	0.42	0.29
60	0.56	0.44
70	0.71	0.64
80	0.89	0.93
90	1.08	1.33
100	1.31	1.84
120	1.80	3.34
140	2.43	5.60
160	2.98	8.89
180	3.86	13.47
200	4.75	19.71

(c) Timings of degree-N models for f_2.



(d) Overapproximation ratio of the remainder of the degree-N Chebyshev model for $f_\varepsilon - 1$ over $[0, 1]$, compared to the optimal bound $\sqrt{\varepsilon}$.

Figure 2 Approximating functions f_ε and $x \mapsto |x|$ with Chebyshev models.

6.2 Evaluating an Abelian integral

Abelian integrals naturally appear when computing the number of limit cycles bifurcating from a Hamiltonian polynomial vector field in the plane. Indeed, the number of sign alternations of those contour integrals (parameterised by the energy level of the potential function) gives a lower bound on the number of limit cycles of the perturbed system, which is a hard question related to Hilbert's 16th problem.

In [25], the author claims to prove the existence of 26 limit cycles for a well-constructed quartic system, whereas the previous record for degree 4 was 22 [14]. However, the implementation with which the Abelian integrals were "rigorously" computed was erroneous, which led to apparently more sign alternations than in reality. By tuning the coefficients and computing the integrals with another rigorous numerics library, the authors of the ongoing work [9] obtain 24 limit cycles, which, if not 26, is still greater than the current record 22.

To conclude this article, we rigorously evaluate some of these integrals inside Coq to show how our implementation behaves on non-crafted examples. Below are the formulas defining a family of integral $\mathfrak{I}_{ij}(r)$ which need to be computed precisely for several values of r. Table 1 summarises the results of our computational experiments. In each line, we chose parameters that were enough to obtain the desired precision. These encouraging results give us hope that it will be possible to fully verify the critical computations involved in recent work of the first author [9].

$$\mathfrak{I}_{ij}(r) = \int_{x_-}^{x_+} x^i (y^+(x)^{j-1} - y^-(x)^{j-1})\,\mathrm{d}x + \int_{y_-}^{y_+} (x^-(y)^{i-1} + x^+(y)^{i-1}) y^j \frac{y^2 - y_0}{\delta_x(y)}\,\mathrm{d}y.$$

$$x_0 = \frac{9}{10}, \quad x_\pm = \sqrt{x_0 \pm r/\sqrt{2}}\ , \quad \delta_y(x) = \sqrt{r^2 - (x^2 - x_0)^2}\ , \quad x^\pm(y) = \sqrt{x_0 \pm \delta_x(y)}\ ,$$

$$y_0 = \frac{11}{10}, \quad y_\pm = \sqrt{y_0 \pm r/\sqrt{2}}\ , \quad \delta_x(y) = \sqrt{r^2 - (y^2 - y_0)^2}\ , \quad y^\pm(x) = \sqrt{y_0 \pm \delta_y(x)}\ .$$

Table 1 Reached precision for $\mathfrak{I}_{ij}(r)$ for different values of r, computed with degree-N Chebyshev models and floating-point precision p (in each cell we display the width of the computed enclosure).

r	N	p	time (s)	$\mathfrak{I}_{00}$	$\mathfrak{I}_{20}$	$\mathfrak{I}_{22}$	$\mathfrak{I}_{40}$	$\mathfrak{I}_{04}$
0.5	13	32	0.38	2,4e-05	2,9e-05	4,1e-05	3,0e-05	4,8e-05
0.78	15	32	0.47	4,6e-05	2,0e-05	2,7e-05	2,4e-05	1,1e-04
0.88	65	128	17.34	2,5e-08	5,0e-11	8,5e-11	5,3e-11	6,3e-08
0.89	95	128	35.13	2,0e-08	1,8e-11	2,9e-11	2,0e-11	5,1e-08
0.895	135	300	173.23	2,6e-08	1,7e-11	1,8e-11	1,3e-11	6,7e-08

7 Conclusion and future work

The Coq development is available online [11]. It consists of around 1300 lines of specifications and 1500 lines of proofs. We leave several directions for future work: integrate it with CoqInterval to benefit from its automatic subdivision techniques; interface the library with external tools for the approximation steps; implement other bases; address higher-dimensional problems. Applying this approach to verify solutions of linear ODEs in a systematic way [2, 8] is also a longer-term perspective.

References

1 Henk Barendregt and Erik Barendsen. Autarkic Computations in Formal Proofs. *Journal of Automated Reasoning*, 28(3):321–336, April 2002.

2 Alexandre Benoit, Mioara Joldeş, and Marc Mezzarobba. Rigorous uniform approximation of D-finite functions using Chebyshev expansions. *Math. Comp.*, 86(305):1303–1341, 2017. doi:10.1090/mcom/3135.

3 Vasile Berinde. *Iterative approximation of fixed points*, volume 1912 of *Lecture Notes in Mathematics*. Springer, Berlin, 2007.

4 Martin Berz and Kyoko Makino. Verified integration of ODEs and flows using differential algebraic methods on high-order Taylor models. *Reliable Computing*, 4(4):361–369, 1998.

5 Sylvie Boldo, François Clément, Florian Faissole, Vincent Martin, and Micaela Mayero. A Coq formal proof of the Lax–Milgram theorem. In *6th ACM SIGPLAN Conference on Certified Programs and Proofs*, Paris, France, January 2017. doi:10.1145/3018610.3018625.

6 Sylvie Boldo and Guillaume Melquiond. *Verifying Floating-point Algorithms with the Coq System*. Elsevier, 2017.

7 Nicolas Bourbaki. *General Topology*. Springer, 1995. Original French edition published by MASSON, Paris, 1971. doi:10.1007/978-3-642-61701-0.

8 Florent Bréhard, Nicolas Brisebarre, and Mioara Joldes. Validated and numerically efficient Chebyshev spectral methods for linear ordinary differential equations. *ACM Transactions on Mathematical Software*, 2018.

9 Florent Bréhard, Nicolas Brisebarre, Mioara Joldes, and Warwick Tucker. A New Lower Bound on the Hilbert Number for Quartic Systems, 2019. URL: http://www.jncf2019.uvsq.fr/program/abs-brehard.pdf.

10 Nicolas Brisebarre and Mioara Joldeş. Chebyshev interpolation polynomial-based tools for rigorous computing. In *Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, pages 147–154. ACM, 2010.

11 Florent Bréhard, Assia Mahboubi, and Damien Pous. Web appendix to the present paper. URL: https://gitlab.inria.fr/amahboub/approx-models.

12 Florent Bréhard, Assia Mahboubi, and Damien Pous. A certificate-based approach to formally verified approximations, 2019. Version with appendix. URL: https://hal.archives-ouvertes.fr/hal-02088529.

13 Olga Caprotti and Martijn Oostdijk. Formal and Efficient Primality Proofs by Use of Computer Algebra Oracles. *J. Symb. Comput.*, 32(1/2):55–70, 2001.

14 Colin Christopher. Estimating limit cycle bifurcations from centers. In *Differential equations with symbolic computation*, Trends Math., pages 23–35. Birkhäuser, Basel, 2005. doi:10.1007/3-7643-7429-2_2.

15 Tobin A Driscoll, Nicholas Hale, and Lloyd N Trefethen. Chebfun guide, 2014.

16 L. Fox and I. B. Parker. *Chebyshev polynomials in numerical analysis*. Oxford University Press, London-New York-Toronto, Ont., 1968.

17 Benjamin Grégoire and Assia Mahboubi. Proving Equalities in a Commutative Ring Done Right in Coq. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2005. doi:10.1007/11541868_7.

18 Benjamin Grégoire and Laurent Théry. A Purely Functional Library for Modular Arithmetic and Its Application to Certifying Large Prime Numbers. In *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 423–437. Springer, 2006.

19 Thomas C. Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A formal proof of the Kepler conjecture. *CoRR*, abs/1501.02155, 2015. URL: http://arxiv.org/abs/1501.02155, arXiv:1501.02155.

20 John Harrison and Laurent Théry. A Skeptic's Approach to Combining HOL and Maple. *J. Autom. Reasoning*, 21(3):279–294, 1998.

21 Johannes Hölzl, Fabian Immler, and Brian Huffman. Type Classes and Filters for Mathematical Analysis in Isabelle/HOL. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, pages 279–294, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

22 Allan Hungria, Jean-Philippe Lessard, and Jason D. Mireles James. Rigorous numerics for analytic solutions of differential equations: the radii polynomial approach. *Math. Comp.*, 85(299):1427–1459, 2016.

23 Fabian Immler. A verified ODE solver and the Lorenz attractor. *Journal of automated reasoning*, pages 1–39, 2018.

24 Fredrik Johansson. Arb: efficient arbitrary-precision midpoint-radius interval arithmetic. *IEEE Transactions on Computers*, 66(8):1281–1292, 2017.

25 Tomas Johnson. A quartic system with twenty-six limit cycles. *Exp. Math.*, 20(3):323–328, 2011. doi:10.1080/10586458.2011.565252.

26 Mioara Joldeş. *Rigorous Polynomial Approximations and Applications*. PhD thesis, École normale supérieure de Lyon – Université de Lyon, Lyon, France, 2011. URL: https://tel.archives-ouvertes.fr/tel-00657843.

27 Edgar W Kaucher and Willard L Miranker. *Self-validating numerics for function space problems: Computation with guarantees for differential and integral equations*, volume 9. Elsevier, 1984.

28 Rudi Klatte, Ulrich Kulisch, Andreas Wiethoff, and Michael Rauch. *C-XSC: A C++ class library for extended scientific computing*. Springer Science & Business Media, 2012.

29 Jean-Philippe Lessard and Christian Reinhardt. Rigorous numerics for nonlinear differential equations using Chebyshev series. *SIAM J. Numer. Anal.*, 52(1):1–22, 2014.

30 Assia Mahboubi, Guillaume Melquiond, and Thomas Sibut-Pinote. Formally Verified Approximations of Definite Integrals. *Journal of Automated Reasoning*, 62(2):281–300, February 2019. doi:10.1007/s10817-018-9463-7.

31 Érik Martin-Dorel and Guillaume Melquiond. Proving Tight Bounds on Univariate Expressions with Elementary Functions in Coq. *Journal of Automated Reasoning*, 57(3):187–217, October 2016. doi:10.1007/s10817-015-9350-4.

32 Guillaume Melquiond. Proving bounds on real-valued functions with computations. In *International Joint Conference on Automated Reasoning*, pages 2–17. Springer, 2008.

33 Ramon E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

34 Nathalie Revol and Fabrice Rouillier. Motivations for an arbitrary precision interval arithmetic and the MPFI library. *Reliable computing*, 11(4):275–290, 2005.

35 Siegfried M Rump. INTLAB—interval laboratory. In *Developments in reliable computing*, pages 77–104. Springer, 1999.

36 Lloyd Nicholas Trefethen. *Approximation Theory and Approximation Practice*. SIAM, 2013. See http://www.chebfun.org/ATAP/. URL: http://www.chebfun.org/ATAP/.

37 Warwick Tucker. A rigorous ODE solver and Smale's 14th problem. *Found. Comput. Math.*, 2(1):53–117, 2002. URL: http://www.math.cornell.edu/~warwick/main/rodes/JFoCM.pdf.

38 Warwick Tucker. *Validated numerics: a short introduction to rigorous computations*. Princeton University Press, 2011.

39 Jan Bouwe Van Den Berg and Jean-Philippe Lessard. Chaotic braided solutions via rigorous numerics: Chaos in the Swift–Hohenberg equation. *SIAM Journal on Applied Dynamical Systems*, 7(3):988–1031, 2008.

40 Nobito Yamamoto. A numerical verification method for solutions of boundary value problems with local uniqueness by Banach's fixed-point theorem. *SIAM J. Numer. Anal.*, 35(5):2004–2013, 1998.

Higher-Order Tarski Grothendieck as a Foundation for Formal Proof

Chad E. Brown
Czech Technical University in Prague, Czech Republic

Cezary Kaliszyk
University of Innsbruck, Austria
University of Warsaw, Poland
cezary.kaliszyk@uibk.ac.at

Karol Pąk
University of Białystok, Poland
pakkarol@uwb.edu.pl

Abstract

We formally introduce a foundation for computer verified proofs based on higher-order Tarski-Grothendieck set theory. We show that this theory has a model if a 2-inaccessible cardinal exists. This assumption is the same as the one needed for a model of plain Tarski-Grothendieck set theory. The foundation allows the co-existence of proofs based on two major competing foundations for formal proofs: higher-order logic and TG set theory. We align two co-existing Isabelle libraries, Isabelle/HOL and Isabelle/Mizar, in a single foundation in the Isabelle logical framework. We do this by defining isomorphisms between the basic concepts, including integers, functions, lists, and algebraic structures that preserve the important operations. With this we can transfer theorems proved in higher-order logic to TG set theory and vice versa. We practically show this by formally transferring Lagrange's four-square theorem, Fermat 3-4, and other theorems between the foundations in the Isabelle framework.

2012 ACM Subject Classification Theory of computation → Interactive proof systems; Theory of computation → Logic and verification

Keywords and phrases model, higher-order, Tarski Grothendieck, proof foundation

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.9

Supplement Material The formalization is available at:
`http://cl-informatik.uibk.ac.at/cek/itp19merge/`

Funding *Chad E. Brown*: European Research Council (ERC) grant no. 649043 *AI4REASON*
Cezary Kaliszyk: European Research Council (ERC) grant no. 714034 *SMART*
Karol Pąk: Polish National Science Center granted by decision no. DEC-2015/19/D/ST6/01473

1 Introduction

Various formal proof foundations combine higher-order logic with set theory [10, 24, 34, 35]. Such a combination offers a familiar mathematical foundation, while at the same time offering more powerful automation present in HOL. All the combinations have been presented without a model, even though models for the two separate foundations are well known and studied. In this paper we will give a model of such a combination and show some consequences of the existence of the model for practical formalizations.

Today the libraries of proof assistants based on the two separate foundations are among the largest proof libraries available. The library of higher-order logic based Isabelle/HOL [44] together with the Archive of Formal Proofs consist of more than 100,000 theorems [9], while the Mizar Mathematical Library (MML) [6, 16] based on set theory contains 59,000 theorems. A number of results in the libraries are incomparable, for example among the theorems

present in Wiedjik's list of 100 important theorems to formalize Isabelle has 16 theorems not formalized in Mizar, while Mizar has 5 theorems absent in Isabelle (64 are formalized in both). The Mizar library includes results about lattice theory [7], topology, and manifolds [39] not present in the Isabelle library.

A model for the higher-order Tarski-Grothendieck allows merging the results in the two libraries. This merging will be performed mostly manually. The reason for this is that definitions for isomorphic concepts may be quite different in the usual approaches in these system. Consider the real numbers. In the MML their definition is performed in multiple steps. First, natural numbers are introduced using the set-theoretic successor. Next, positive rationals are created by adding fractions as pairs of irreducible naturals $\langle n, k\rangle$ (with $k > 1$). Finally, Dedekind cuts are used to obtain positive reals. The Isabelle approach is fundamentally different. Natural numbers are a subtype of the axiomatic type of individuals. Pairs of naturals are quotiented into integers and rationals. Finally, Cauchy sequences of rationals grant reals. The differences in the construction also imply differences in their behaviours. Every Mizar natural number is also an integer or real, while in Isabelle coercions are required. It is similar when it comes to mathematical structures (used by over 70% of the Mizar library). Their semantics [22] in Mizar is close to partial functions specified on named fields, which enables for example inheritance and this is used to realize the main algebraic structures. Isabelle records are quite similar, but it is type classes that are used to express algebra.

We will propose a way to lift the merged elementary concepts to the more involved ones. By associating the Isabelle number 0 and the empty set and the corresponding successor operations, we will show a homomorphism between the set theoretic and higher-order natural numbers and later integers. We will show that this homomorphism preserves the basic operations, which will allow transporting basic number theorems, including Lagrange, and Bertrand, and cases of Fermat's last theorem.

We will also show that it is possible to show a mapping between the Isabelle type classes and the set theoretic structures corresponding basic algebra. This will allow merging the formalizations of groups and rings in the two libraries. We again use some merged basic concepts, namely functions and binary operators. This brings us to Euclidean spaces where we transport the Brouwer theorem for n-dimensional case (the fixed point theorem [37], the topological invariance of degree, and the topological invariance of dimension [38]) that are essential to define and develop topological manifolds.

The rest of the paper is structured as follows. In Section 2 we review the higher-order logic foundations used later. Section 3 gives an axiomatization of higher-order Tarski-Grothendieck (HOTG). We first define it in a higher-order setting and then relate to the actual proof assistants based on this foundation. Section 4 presents our model of HOTG. Next, in Section 5 we show the implications of the existence of the model for practical formalization: we align the proof libraries of Isabelle/HOL and Isabelle/Mizar by building isomorphisms between the various concepts present in these libraries and by translating theorems via the isomorphism. Section 6 discusses related work.

2 Preliminaries

We begin by reviewing the syntax and semantics of higher-order logic. The original presentation of higher-order logic using simple type theory was due to Church [12] with a corresponding notion of semantics due to Henkin [19] (with an important correction by Andrews [2]). We largely follow the notation and presentation style used in [5].

Let $\mathcal{B}$ be a set of base types. We use β to range over the types in $\mathcal{B}$. We next define *types* and use σ, τ to range over types. The set $\mathcal{T}$ of types is given by inductively extending $\mathcal{B}$ to include the type o (of truth values) and the type $\sigma \Rightarrow \tau$ (of functions from σ to τ) for all $\sigma, \tau \in \mathcal{T}$. We assume $o \notin \mathcal{B}$ and that types are freely generated.

For each type σ let $\mathcal{V}_\sigma$ be a countable set of variables of type σ, where we assume $\mathcal{V}_\sigma \cap \mathcal{V}_\tau = \emptyset$ whenever $\sigma \neq \tau$. We use x, y, z to range over variables. For each type σ let $\mathcal{C}_\sigma$ be a set of constants of type σ, where again $\mathcal{C}_\sigma \cap \mathcal{C}_\tau = \emptyset$ whenever $\sigma \neq \tau$. Furthermore, we assume $\mathcal{V}_\sigma \cap \mathcal{C}_\tau = \emptyset$. We use c, d to range over constants. A *name* is either a variable or a constant. We use ν to range over names.

We now inductively define a family of sets Λ_σ of *terms*, using s, t, u to range over terms. For the base cases, $\mathcal{V}_\sigma \subseteq \Lambda_\sigma$ and $\mathcal{C}_\sigma \subseteq \Lambda_\sigma$. There are two inductive cases: application and abstraction. If $s \in \Lambda_{\sigma\Rightarrow\tau}$ and $t \in \Lambda_\sigma$, then $(st) \in \Lambda_\tau$. If $x \in \mathcal{V}_\sigma$ and $t \in \Lambda_\tau$, then $(\lambda x.t) \in \Lambda_{\sigma\Rightarrow\tau}$. We often omit parenthesis with the convention that application associates to the left, so that stu means $((st)u)$. Terms of type o are also called *formulas*.

We insist on the inclusion of certain constants called *logical constants* in the family $\mathcal{C}$ of constants. For simplicity of presentation, we take every logical constant we will use as a constant. In particular, we assume:

- $\neg$ is a logical constant in $\mathcal{C}_{o\Rightarrow o}$. We write $\neg(st)$ as $\neg st$.
- $\wedge$, $\vee$, $\longrightarrow$ and $\longleftrightarrow$ are logical constants in $\mathcal{C}_{o\Rightarrow o\Rightarrow o}$. We use infix notation for $\wedge$, $\vee$, $\longrightarrow$ and $\longleftrightarrow$, with priority in this order, and each one associating to the right.
- For each type σ Π_σ and Σ_σ are a logical constants in $\mathcal{C}_{(\sigma\Rightarrow o)\Rightarrow o}$. We write $\forall x_1 \cdots x_n : \sigma.t$ to mean $\Pi_\sigma(\lambda x_1.\cdots\Pi_\sigma(\lambda x_n.t))$ and write $\exists x_1 \cdots x_n : \sigma.t$ to mean $\Sigma_\sigma(\lambda x_1.\cdots\Sigma_\sigma(\lambda x_n.t))$.
- For each type σ $=_\sigma$ is a logical constant in $\mathcal{C}_{\sigma\Rightarrow\sigma\Rightarrow o}$. We write $=_\sigma \ s\ t$ in infix as $s = t$.
- For each type σ ε_σ is a logical constant in $\mathcal{C}_{(\sigma\Rightarrow o)\Rightarrow\sigma}$.

It is well-known that smaller sets of logical constants would be sufficient. For example, it is known that in (extensional) higher-order logic equality is sufficient to define the propositional constants and connectives as well as the existential and universal quantifiers at each type [1].

We next turn to a review of Henkin semantics for our language [19] closely following the presentation style in [5]. A *frame* is a family $\mathcal{D}_\sigma$ of nonempty sets such that $\mathcal{D}_o = \{0, 1\}$ and $\mathcal{D}_{\sigma\Rightarrow\tau} \subseteq (\mathcal{D}_\tau)^{\mathcal{D}_\sigma}$ for each $\sigma, \tau \in \mathcal{T}$. A frame is called *standard* if $\mathcal{D}_{\sigma\Rightarrow\tau} = (\mathcal{D}_\tau)^{\mathcal{D}_\sigma}$ for every $\sigma, \tau \in \mathcal{T}$. An *assignment* is a function $\mathcal{I}$ mapping every name of type σ to an element in $\mathcal{D}_\sigma$. Given a variable $x \in \mathcal{V}_\sigma$ and element $a \in \mathcal{D}_\sigma$ let $\mathcal{I}^x_a$ be the assignment agreeing with $\mathcal{I}$ except possibly on x where $\mathcal{I}^x_a(x) = a$. An assignment $\mathcal{I}$ is *logical* if for each $\sigma \in \mathcal{T}$ the following conditions hold:

- for $a \in \mathcal{D}_o$ $\mathcal{I}(\neg)(a) = 1$ if and only if $a = 0$,
- for $a, b \in \mathcal{D}_o$ $\mathcal{I}(\wedge)(a)(b) = 1$ if and only if $a = 1$ and $b = 1$,
- for $a, b \in \mathcal{D}_o$ $\mathcal{I}(\vee)(a)(b) = 1$ if and only if $a = 1$ or $b = 1$,
- for $a, b \in \mathcal{D}_o$ $\mathcal{I}(\longrightarrow)(a)(b) = 1$ if and only if $a = 0$ or $b = 1$,
- for $a, b \in \mathcal{D}_o$ $\mathcal{I}(\longleftrightarrow)(a)(b) = 1$ if and only if $a = b$,
- for $f \in \mathcal{D}_{\sigma\Rightarrow o}$ $\mathcal{I}(\Pi_\sigma)(f) = 1$ if and only if $f(a) = 1$ for all $a \in \mathcal{D}_\sigma$,
- for $f \in \mathcal{D}_{\sigma\Rightarrow o}$ $\mathcal{I}(\Sigma_\sigma)(f) = 1$ if and only if there is some $a \in \mathcal{D}_\sigma$ such that $f(a) = 1$,
- for $a, b \in \mathcal{D}_\sigma$ $\mathcal{I}(=_\sigma)(a)(b) = 1$ if and only if $a = b$, and
- for $f \in \mathcal{D}_{\sigma\Rightarrow o}$ $f(\mathcal{I}(\varepsilon_\sigma)(f)) = 1$ if and only if there is some $a \in \mathcal{D}_\sigma$ such that $f(a) = 1$.

In other words, $\mathcal{I}$ is logical if it interprets the logical constants appropriately.

We lift an assignment $\mathcal{I}$ to be a partial function $\hat{\mathcal{I}}$ on terms as follows:

- For names ν, $\hat{\mathcal{I}}(\nu) = \mathcal{I}(\nu)$.
- For $s \in \Lambda_{\sigma\Rightarrow\tau}$ and $t \in \Lambda_\sigma$, $\hat{\mathcal{I}}(st) = f(a)$ if $\hat{\mathcal{I}}(s) = f \in \mathcal{D}_{\sigma\Rightarrow\tau}$ and $\hat{\mathcal{I}}(t) = a \in \mathcal{D}_\sigma$.
- For $x \in \mathcal{V}_\sigma$ and $t \in \Lambda_\tau$, $\hat{\mathcal{I}}(\lambda x.t) = f$ if $f \in \mathcal{D}_{\sigma\Rightarrow\tau}$ and $\widehat{\mathcal{I}^x_a}(t) = f(a)$ for all $a \in \mathcal{D}_\sigma$.

Note that for all $s \in \Lambda_\sigma$ if $\hat{\mathcal{I}}(s)$ is defined, then $\hat{\mathcal{I}}(s) \in \mathcal{D}_\sigma$. If $\hat{\mathcal{I}}$ is a total function with domain $\bigcup_{\sigma\in\mathcal{T}}$, then $\mathcal{I}$ is called an *interpretation*.

A *(Henkin) model* is a pair $(\mathcal{D},\mathcal{I})$ where $\mathcal{D}$ is a frame and $\mathcal{I}$ is a logical interpretation. A model is called *standard* if the frame is standard. We say $(\mathcal{D},\mathcal{I})$ satisfies a formula s if $\hat{\mathcal{I}}(s) = 1$ and say $(\mathcal{D},\mathcal{I})$ is a model for a set $\mathcal{A}$ of formulas if $(\mathcal{D},\mathcal{I})$ satisfies every $s \in \mathcal{A}$.

To simplify the presentation above, some dependencies were left implicit. For each set $\mathcal{B}$ of base types (with $o \notin \mathcal{B}$), we obtain a set $\mathcal{T}^{\mathcal{B}}$ of types. Additionally, for each set $\mathcal{B}$ of base types and each family $\mathcal{C}$ of constants indexed by $\mathcal{T}^{\mathcal{B}}$, we obtain a family $\Lambda^{\mathcal{B},\mathcal{C}}$ of terms. The definition of a frame above technically depends on the set $\mathcal{B}$ of base types and we say $\mathcal{D}$ is a *frame over* $\mathcal{B}$ when this dependency needs to be explicit. Futhermore an assignment depends on both $\mathcal{B}$ and $\mathcal{C}$ and we say $\mathcal{I}$ is an *assignment over* $\mathcal{B}$ *for* $\mathcal{C}$ when these dependencies need to be explicit.

A *theory* is a triple $(\mathcal{B},\mathcal{C},\mathcal{A})$ where $\mathcal{B}$ is a set of base types, $\mathcal{C}$ is a family of sets of constants (which must include the logical constants) over the types $\mathcal{T}^{\mathcal{B}}$ and $\mathcal{A} \subseteq \Lambda_o^{\mathcal{B},\mathcal{C}}$ is a set of formulas called the *axioms* of the theory. A pair $(\mathcal{D},\mathcal{I})$ is a *model of a theory* $(\mathcal{B},\mathcal{C},\mathcal{A})$ if $\mathcal{D}$ is a frame over $\mathcal{B}$, $\mathcal{I}$ is a logical interpretation over $\mathcal{B}$ for $\mathcal{C}$ and $(\mathcal{D},\mathcal{I})$ is a model of the set $\mathcal{A}$ of formulas.

It is known that the notion of a Henkin model provides a sound and complete semantics for a variety of proof calculi [5, 8, 11]. Our concern in this article is not with proof calculi directly, but with consistency of certain axiom sets for higher-order set theory. In this paper we will only consider one axiomatization of higher-order Tarski Grothendieck set theory. Soundness implies it is sufficient to find models of these axiom sets to infer consistency, and for this purpose constructing a standard model is enough. In future work we plan to consider different axiomatizations of higher-order Tarski Grothendieck (e.g., the one in [24]) and plan to use soundness and completeness with respect to Henkin models to prove the two versions of Tarski Grothendieck are equivalent.

3 An Axiomatization of Higher-Order Tarski Grothendieck

In this section we give a formulation of higher-order Tarski Grothendieck (HOTG) set theory by giving a theory **HOTG**. The theory is identical to the one implemented by the first author in the Egal system [10]. In particular, the theory specifies an operator that explicitly gives the Grothenieck universe of a set [17]. In the presence of the axiom of choice, this is equivalent to specifying that such a universe exists for every set, which is the approach used in the Mizar system as specified by Trybulec [43]. In the below axiomatization and in the model in the next section, we will use the explicit universe operation, as it makes the presentation simpler, but our intention is to use it both for explicit universes and implicit ones, as specified in Isabelle/Mizar by Kaliszyk and Pąk [24] using Tarski's Axiom A [42] and used in Section 5.

We first describe the theory **HOTG** as given by the triple $(\mathcal{B},\mathcal{C},\mathcal{A})$. Here $\mathcal{B}$ be the singleton $\{\iota\}$ and the base type ι is intended to be the type of sets. The typed constants $\mathcal{C}$ consists precisely of the logical constants and the following additional constants:

- In in $\mathcal{C}_{\iota\Rightarrow\iota\Rightarrow o}$. We write $\mathsf{In}\ s\ t$ in infix as $s \in t$.
- Empty in $\mathcal{C}_{\iota}$.
- Un in $\mathcal{C}_{\iota\Rightarrow\iota}$.
- Pow in $\mathcal{C}_{\iota\Rightarrow\iota}$.
- Repl in $\mathcal{C}_{\iota\Rightarrow(\iota\Rightarrow\iota)\Rightarrow\iota}$.
- Univ in $\mathcal{C}_{\iota\Rightarrow\iota}$.

To state the axioms, we will use three abbreviations. Let Subq be the term

$$\lambda X.\lambda Y.\forall z : \iota.z \in X \longrightarrow z \in Y$$

of type $\iota \Rightarrow \iota \Rightarrow o$. We write Subq s t as $s \subseteq t$. Let TransSet be the term

$$\lambda U.\forall X : \iota.X \in U \longrightarrow X \subseteq U$$

of type $\iota \Rightarrow o$. Let ZFclosed be the term

$$\lambda U.\ (\forall X : \iota.X \in U \longrightarrow \mathsf{Un}\ X \in U) \wedge (\forall X : \iota.X \in U \longrightarrow \mathsf{Pow}\ X \in U)$$
$$\wedge(\forall X : \iota.\forall F : \iota \Rightarrow \iota.X \in U \longrightarrow (\forall x : \iota.x \in X \longrightarrow F\ x \in U) \longrightarrow \mathsf{Repl}\ X\ F \in U)$$

of type $\iota \Rightarrow o$.

The set $\mathcal{A}$ of axioms consists of the following formulas:

Extensionality: $\forall XY : \iota.X \subseteq Y \longrightarrow Y \subseteq X \longrightarrow X = Y$.
$\in$-Induction $\forall P : \iota \Rightarrow o.(\forall X : \iota.(\forall x : \iota.x \in X \longrightarrow Px) \longrightarrow PX) \longrightarrow \forall X : \iota.PX$.
Empty: $\neg\exists x : \iota.x \in \mathsf{Empty}$.
Union: $\forall X : \iota.\forall x : \iota.x \in \mathsf{Un}\ X \longleftrightarrow \exists Y : \iota.x \in Y \wedge Y \in X$.
Power: $\forall XY : \iota.Y \in \mathsf{Pow}\ X \longleftrightarrow Y \subseteq X$.
Replacement: $\forall X : \iota.\forall F : \iota \Rightarrow \iota.\forall y : \iota.y \in \mathsf{Repl}\ X\ F \longleftrightarrow \exists x : \iota.x \in X \wedge y = Fx$.
UnivIn: $\forall N : \iota.N \in \mathsf{Univ}N$
UnivTransSet: $\forall N : \iota.\mathsf{TransSet}\ (\mathsf{Univ}N)$.
UnivZF: $\forall N : \iota.\mathsf{ZFclosed}\ (\mathsf{Univ}N)$.
UnivMin: $\forall N\,U : \iota.N \in U \longrightarrow \mathsf{TransSet}\ U \longrightarrow \mathsf{ZFclosed}\ U \longrightarrow \mathsf{Univ}N \subseteq U$.

4 A Model of Higher-Order Set Theory

We will make heavy use of the von Neumann hierarchy (see for example [28]). By ordinal induction we define the set V_α for ordinals α as $V_\emptyset = \emptyset$, $V_{\alpha+1} = \wp(V_\alpha)$ and $V_\lambda = \bigcup_{\alpha<\lambda} V_\alpha$. Since we work in a well-founded set theory, for every set X there is some ordinal α such that $X \subseteq V_\alpha$ (and so $X \in V_{\alpha+1}$).

A cardinal κ is *inaccessible* if it is regular and $\lambda < \kappa$ implies $2^\lambda < \kappa$. A cardinal κ is *2-inaccessible* if it is a regular limit of inaccessible cardinals. Note that if κ is *2-inaccessible*, then for every $\lambda < \kappa$ there is some inaccessible κ' with $\lambda < \kappa' < \kappa$. It easily follows every 2-inaccessible is also inaccessible.

The following proposition can be found in Kanamori (see Proposition 2.1 in [27]).

▶ **Proposition 1.** *Let κ be inaccessible.*
1. *$x \subseteq V_\kappa$ implies $x \in V_\kappa$ iff $|x| < \kappa$.*
2. *$V_\kappa \models$ ZFC*

We define universes following Grothendieck [17].

▶ **Definition 2.** *Let U be a set. We say U is a* universe *if four conditions hold:*
- *U is transitive.*
- *If $x, y \in U$, then $\{x, y\} \in U$.*
- *If $X \in U$, then $\wp(X) \in U$.*
- *If $I \in U$ and $X_i \in U$ for each $i \in I$, then $\bigcup_{i\in I} X_i \in U$.*

The fact that every inaccessible yields a universe follows easily from Proposition 1.

▶ **Proposition 3.** *If κ is inaccessible, then V_κ is a universe.*

The following proposition will ensure that universes satisfy the properties in the definition of ZFclosed.

▶ **Proposition 4.** *Let U be a universe.*
1. *If $X \in U$, then $\bigcup X \in U$.*
2. *If $X \in U$ and $f : X \to U$, then $\{f(x)|x \in X\} \in U$.*

Proof. Suppose $X \in U$. We know $\bigcup X \in U$ since $\bigcup X = \bigcup_{x \in X}\{x\}$. Now suppose $X \in U$ and $f : X \to U$. We know $\{f(x)|x \in X\} \in U$ since $\{f(x)|x \in X\} = \bigcup_{x \in X}\{f(x)\}$. ◀

To interpret the constant Univ we will not only need universes, but a global function uniformly giving the least universe containing a given set.

▶ **Definition 5.** *Let $\alpha > 0$ be an ordinal. A* universe function for α *is a function $\mathcal{U} : V_\alpha \to V_\alpha$ such that for all $A \in V_\alpha$ we have $A \in \mathcal{U}(A)$, $\mathcal{U}(A)$ is a universe and $\mathcal{U}(A) \subseteq U$ for all universes $U \in V_\alpha$ with $A \in U$.*

▶ **Definition 6.** *Let $\alpha > 0$ be an ordinal and $\mathcal{U}$ be a universe function for α. Let $\mathcal{D}^\alpha_\iota$ be V_α, $\mathcal{D}^\alpha_o = \{0, 1\}$ and $\mathcal{D}^\alpha_{\sigma\Rightarrow\tau} = (\mathcal{D}^\alpha_\tau)^{\mathcal{D}^\alpha_\sigma}$ for each $\sigma, \tau \in \mathcal{T}^{\mathcal{B}}$. Note that $V_\alpha \neq \emptyset$ since $\alpha > 0$ and so $\mathcal{D}^\alpha$ is a standard frame over $\mathcal{B}$. We call $\mathcal{D}^\alpha$ the* standard set-theoretic frame for α. *An assignment $\mathcal{I}$ over $\mathcal{B}$ for $\mathcal{C}$ into $\mathcal{D}^\alpha$ is called a* standard set-theoretic interpretation for α and $\mathcal{U}$ *if $\mathcal{I}$ is a logical interpretation and the following properties hold:*
- $\mathcal{I}(\mathsf{In})(a)(A) = 1$ *if and only if $a \in A$ for $a, A \in \mathcal{D}^\alpha_\iota$.*
- $\mathcal{I}(\mathsf{Empty}) = \emptyset$
- $\mathcal{I}(\mathsf{Un})(A) = \bigcup A$ *for every $A \in \mathcal{D}^\alpha_\iota$.*
- $\mathcal{I}(\mathsf{Pow})(A) = \wp(A)$ *for every $A \in \mathcal{D}^\alpha_\iota$.*
- $\mathcal{I}(\mathsf{Repl})(A)(f) = \{f(a)|a \in A\}$ *for every $A \in \mathcal{D}^\alpha_\iota$ and $f \in \mathcal{D}^\alpha_{\iota\Rightarrow\iota}$.*
- $\mathcal{I}(\mathsf{Univ}) = \mathcal{U}$.

▶ **Theorem 7.** *Let $\alpha > 0$ be an ordinal, $\mathcal{U}$ be a universe function for α and $\mathcal{D}^\alpha$ be the standard set-theoretic frame for α. If $\mathcal{I}$ is a standard set-theoretic interpretation for α and $\mathcal{U}$, then $(\mathcal{D}^\alpha, \mathcal{I})$ is a model of the theory* **HOTG**.

Proof. Assume $\mathcal{I}$ is a standard set-theoretic interpretation for α and $\mathcal{U}$. We only need to prove $\mathcal{I}$ maps every formula in $\mathcal{A}$ to 1.
Extensionality: The fact that

$$\mathcal{I}(\forall XY : \iota.X \subseteq Y \longrightarrow Y \subseteq X \longrightarrow X = Y) = 1$$

follows easily from the fact that $A = B$ whenever $A \subseteq B$ and $B \subseteq A$ for $A, B \in V_\alpha$.
$\in$-Induction: In order to prove

$$\mathcal{I}(\forall P : \iota \Rightarrow o.(\forall X : \iota.(\forall x : \iota.x \in X \longrightarrow Px) \longrightarrow PX) \longrightarrow \forall X : \iota.PX) = 1$$

it suffices to prove that $C = V_\alpha$ for every $C \subseteq V_\alpha$ such that $A \in C$ for every $A \in V_\alpha$ with $A \subseteq C$. Let $C \subseteq V_\alpha$ be given and assume $A \in C$ for every $A \in V_\alpha$ with $A \subseteq C$. Consider $V_\alpha \setminus C$. Assume $V_\alpha \neq C$. In this case $V_\alpha \setminus C$ must be nonempty. By regularity there is an element $A \in V_\alpha \setminus C$ such that $A \cap (V_\alpha \setminus C) = \emptyset$. Since V_α is transitive $A \subseteq V_\alpha$ and so $A \cap (V_\alpha \setminus C) = \emptyset$ implies $A \subseteq C$. By our assumption about C, we must have $A \in C$, contradicting $A \in V_\alpha \setminus C$.
Empty: We know $\mathcal{I}(\neg\exists x : \iota.x \in \mathsf{Empty}) = 1$ since $\mathcal{I}(\mathsf{Empty}) = \emptyset$.
Union: We know $\mathcal{I}(\forall X : \iota.\forall x : \iota.x \in \mathsf{Un}\ X \longleftrightarrow \exists Y : \iota.x \in Y \wedge Y \in X) = 1$ since $\mathcal{I}(\mathsf{Un})(A) = \bigcup A$.

Power: We know $\mathcal{I}(\forall XY : \iota.Y \in \mathsf{Pow}\ X \longleftrightarrow Y \subseteq X) = 1$ since $\mathcal{I}(\mathsf{Pow})(A) = \wp A$.

Replacement: We can easily prove $\mathcal{I}(\forall X : \iota.\forall F : \iota \Rightarrow \iota.\forall y : \iota.y \in \mathsf{Repl}\ X\ F \longleftrightarrow \exists x : \iota.x \in X \wedge y = Fx) = 1$ using the fact that $\mathcal{I}(\mathsf{Repl})(A)(f) = \{f(a)|a \in A\}$ for every $A \in V_\alpha$ and every $f : V_\alpha \to V_\alpha$.

UnivIn: Since $\mathcal{U}$ is a universe function we know $A \in \mathcal{U}(A)$ for every $A \in V_\alpha$. Hence $\mathcal{I}(\forall N : \iota.N \in \mathsf{Univ}N) = 1$.

UnivTransSet: Since $\mathcal{U}$ is a universe function, $\mathcal{U}(A)$ is a universe (and hence transitive) for every $A \in V_\alpha$. Hence $\mathcal{I}(\forall N : \iota.\mathsf{TransSet}\ (\mathsf{Univ}N)) = 1$.

UnivZF: It is easy to see $\mathcal{I}(\forall N : \iota.\mathsf{ZFclosed}\ (\mathsf{Univ}N)) = 1$ using Definitions 2 and 5 and Proposition 4.

UnivMin: Suppose $A, U \in V_\alpha$ where $A \in U$, U is transitive and $\mathcal{I}(\mathsf{ZFclosed})(U) = 1$. We argue that U is a universe. We know U is transitive. The fact that $\wp(X) \in U$ whenever $X \in U$ follows directly from $\mathcal{I}(\mathsf{ZFclosed})(U) = 1$. In particular, since $A \in U$, we know $\wp(A) \in U$ and $\wp(\wp(A)) \in U$. Let $x, y \in U$ be given. Let $f : \wp(\wp(A)) \to U$ be the function

$$f(X) = \begin{cases} x & \text{if } A \in X \\ y & \text{otherwise} \end{cases}$$

Since $f(A) = x$ and $f(\emptyset) = y$, we know $\{x, y\} = \{f(X)|X \in \wp(\wp(A))\}$. Using $\mathcal{I}(\mathsf{ZFclosed})(U) = 1$ we conclude $\{x, y\} \in U$. Now let $I \in U$ and a family $X_i \in U$ for each $i \in I$ be given. Let $g : I \to U$ be the function $g(i) = X_i$. Using $\mathcal{I}(\mathsf{ZFclosed})(U) = 1$ we know $\{g(i)|i \in I\} \in U$ and then $\bigcup_{i \in I} X_i = \bigcup\{g(i)|i \in I\} \in U$. Hence U is a universe. Since U is a universe with $A \in U$, we conclude $\mathcal{U}(A) \subseteq U$ from Definition 5. ◀

For a general ordinal α there will be no universe function $\mathcal{U}$. For 2-inaccessible cardinals there is a universe function and a corresponding standard set-theoretic interpretation.

▶ **Theorem 8.** *Let κ be 2-inaccessible and $\mathcal{D}^\kappa$ be the standard set-theoretic frame for κ. There is a universe function $\mathcal{U}$ for κ and there is a standard set-theoretic interpretation $\mathcal{I}$ for κ and $\mathcal{U}$.*

Proof. We first construct the universe function. For each $A \in V_\kappa$, let A' be

$$\{U \in V_\kappa | U \text{ is a universe and } A \in U\}.$$

We argue A' is always nonempty. Since $A \in V_\kappa$ there must be some $\alpha < \kappa$ such that $A \in V_\alpha$. Since κ is 2-inaccessible there must be some inaccessible $\kappa' < \kappa$ with $\alpha < \kappa'$. By Proposition 3 $V_{\kappa'}$ is a universe and so $V_{\kappa'} \in A'$. Since A' is a nonempty set, $\bigcap A'$ is well-defined and we can take $\mathcal{U}(A)$ to be $\bigcap A'$. A simple inspection of Definition 2 reveals that the intersection of a nonempty set of universes is itself a universe. Thus $\mathcal{U}(A)$ is the least universe with A as a member and $\mathcal{U}$ is a universe function for κ.

Next we turn to the interpretation $\mathcal{I}$. The axiom of choice states that there is a function $\mathfrak{e} : \wp(V_{\kappa+\omega}) \setminus \{\emptyset\} \to V_{\kappa+\omega}$ such that $\mathfrak{e}(A) \in A$ for every $A \in \wp(V_{\kappa+\omega}) \setminus \{\emptyset\}$. An easy induction on types shows $\mathcal{D}^\kappa_\sigma \in V_{\kappa+\omega}$ for each $\sigma \in \mathcal{T}^\mathcal{B}$. Hence $\mathcal{D}^\kappa_\sigma \in \wp(V_{\kappa+\omega}) \setminus \{\emptyset\}$ for each $\sigma \in \mathcal{T}^\mathcal{B}$ since $V_{\kappa+\omega}$ is transitive. We can simply define $\mathcal{I}(x) = \mathfrak{e}(\mathcal{D}^\kappa_\sigma) \in \mathcal{D}^\kappa_\sigma$ for each variable $x \in \mathcal{V}_\sigma$. For the logical constants c other than ε_σ we take the obvious value $\mathcal{I}(c)$ so that $\mathcal{I}$ will be a logical interpretation. In each case this value is in $\mathcal{D}^\kappa_\sigma$ since $\mathcal{D}^\kappa$ is a standard frame. We take $\mathcal{I}(\varepsilon_\sigma)$ to be the function $g \in \mathcal{D}^\kappa_{(\sigma \Rightarrow o) \Rightarrow \sigma}$ such that for $f \in \mathcal{D}^\kappa_{\sigma \Rightarrow o}$ we have

$$g(f) = \begin{cases} \mathfrak{e}(\{a \in \mathcal{D}^\kappa_\sigma | f(a) = 1\}) & \text{if } f(a) = 1 \text{ for some } a \in \mathcal{D}^\kappa_\sigma \\ \mathfrak{e}(\mathcal{D}^\kappa_\sigma) & \text{otherwise.} \end{cases}$$

It only remains to give values $\mathcal{I}(c)$ for the nonlogical constants in $\mathcal{C}$. For In, Empty, Un, Pow and Repl there is at most one corresponding value that might possibly satisfy the conditions in Definition 6. Since we know $\mathcal{D}_\iota^\kappa = V_\kappa$ is a universe, each of these values is in $\mathcal{D}_\sigma^\kappa$ in each respective case. Finally we take $\mathcal{I}(\mathsf{Univ})$ to be the universe function $\mathcal{U}$ constructed above. By the choice of $\mathcal{I}$ it is easy to see that $\mathcal{I}$ is a standard set-theoretic interpretation for κ. ◀

As an easy corollary of Theorems 7 and 8 we have the following relative satisfiability result.

▶ **Theorem 9.** *If there is a 2-inaccessible cardinal, then* **HOTG** *is satisfiable.*

5 Proof Integration

The axiomatization together with the model defined in the previous section allows us to use the higher-order library and set theoretic library simultaneously. We will do this in the Isabelle logical framework, by importing various results from the two libraries in the same environment and define transfer methods between these results. This will allow us to use theorems proved in one of the foundations using the term language of the other.

All the definitions and theorems presented in this section have been formalized in Isabelle and will be presented close to the Isabelle notation. The Isabelle environment will import both Isabelle/HOL [33] and Isabelle/Mizar [24] object logics along with a number of results formalized in the standard libraries of the two. Isabelle distinguishes between meta-level implication ($\Longrightarrow$) and object-level implication ($\longrightarrow$) and our notation in examples below reflects this distinction. The remaining notations will follow first-order conventions. In particular the symbols $=_{\mathcal{H}}$ and $=_{\mathcal{S}}$ will refer to the HOL and set-theoretic equality operations respectively. Finally *be* is the Mizar infix operator for specifying the type of a set in the Mizar intersection type system [25].

To combine two types we will first define bijections between these types. We will next show that the bijection preserves various constants and operators. This will allow us to transfer results using higher-order rewriting, in the style of quotient packages for HOL [20,26] and the Isabelle transfer package [21]. In the MML set theory it is common to reason both about the type of the natural numbers and the members of the set of natural numbers. This is necessary, since the arguments of all operations must be sets, while the reasoning engine allows more advanced reasoning steps for types [6]. We therefore define two operators, one that specifies a bijection between a HOL type and a set theoretic set and one that specified a bijection between a HOL type and a set theoretic type. The definitions are analogous and we show only the latter one here. We will define an isomorphism between a type σ and a set $d \in \Lambda_\iota$ to be a pair (f, g) of functions (at the type theory level) where f maps sets to objects of type σ and g maps objects of type σ to sets in such a way that objects of type σ (in the type theory) correspond uniquely to elements of d (in the set theory).

▶ **Definition 10.** *Let σ be a type, $d \in \Lambda_\iota$ be a set and $\mathfrak{s}2\mathfrak{h} \in \Lambda_{\iota\Rightarrow\sigma}$ and $\mathfrak{h}2\mathfrak{s} \in \Lambda_{\sigma\Rightarrow\iota}$ be functions. The predicate $beIso_S\langle\mathfrak{h}2\mathfrak{s}, \mathfrak{s}2\mathfrak{h}, d\rangle$ holds whenever all of the following hold:*

- $\forall x : \sigma.\mathfrak{s}2\mathfrak{h}(\mathfrak{h}2\mathfrak{s}(x)) =_{\mathcal{H}} x$,
- $\forall x : \iota.x \in d \longrightarrow \mathfrak{h}2\mathfrak{s}(\mathfrak{s}2\mathfrak{h}(x)) =_{\mathcal{S}} x$,
- $\forall x : \sigma.\mathfrak{s}2\mathfrak{h}(x) \in d$.

In Isabelle the definition appears as follows:

definition *beIsoS*($\mathfrak{h}2\mathfrak{s}$,$\mathfrak{s}2\mathfrak{h}$,$d$) $\longleftrightarrow$ (($\forall_L y.\ \mathfrak{s}2\mathfrak{h}(\mathfrak{h}2\mathfrak{s}(y)) =_{\mathcal{H}} y$) $\wedge$
($\forall\, x$:*Element-of* $d.\ \mathfrak{h}2\mathfrak{s}(\mathfrak{s}2\mathfrak{h}(x)) =_{\mathcal{S}} x$)$\wedge$ ($\forall_L y.\ \mathfrak{h}2\mathfrak{s}(y)$ *in* d))

The existence of a bijection does not immediately imply the inhabitation of the type/set. However, as types need to be non-empty in both formalisms, we can derive this result as below. For space reasons we only present the statements, all the theorems have proofs in our formalization.

theorem *beIsoS_d*:
beIsoS($\mathfrak{h}2\mathfrak{s}$,$\mathfrak{s}2\mathfrak{h}$,*d*) $\Longrightarrow$ *d is non empty*

5.1 Natural numbers and integers

The Isabelle/Mizar natural numbers are defined as the smallest limit ordinal. The existence of this set is a consequence of the Tarski universe property. The formal definition is as follows:

mdef *ordinal1_def_11* (*omega*) **where**
func omega $\rightarrow$ *set means* (λ*it*.
$0_{\mathcal{S}}$ *in it* $\wedge$ *it be limit_ordinal* $\wedge$ *it be Ordinal* $\wedge$
($\forall$ *A*:*Ordinal*. $0_{\mathcal{S}}$ *in A* $\wedge$ *A is limit_ordinal* $\longrightarrow$ *it* $\subseteq$ *A*))

On the other hand, the Isabelle natural numbers are a subtype of the type of individuals. In order to merge these two different approaches we specified a functor that preserves zero and the successor. Note that the functor is specified only for the type of the natural numbers which in Isabelle/HOL is implicit, but in the softly-typed set theory needs to be written and checked explicitly. This is the reason for having an undefined case, which as we will see later, still gives an isomorphism.

$$\mathfrak{h}2\mathfrak{s}_{\mathbb{N}}(n) =_{\mathcal{S}} \begin{cases} 0_{\mathcal{S}} & \text{if } n =_{\mathcal{H}} 0_{\mathcal{H}}, \\ \mathrm{S}_{\mathcal{S}}(\mathfrak{h}2\mathfrak{s}_{\mathbb{N}}(k)) & \text{if } n =_{\mathcal{H}} \mathrm{S}_{\mathcal{H}}(k) \text{ for some } \mathcal{H}\text{-natural } k. \end{cases}$$

$$\mathfrak{s}2\mathfrak{h}_{\mathbb{N}}(n) =_{\mathcal{H}} \begin{cases} 0_{\mathcal{H}} & \text{if } n =_{\mathcal{S}} 0_{\mathcal{S}}, \\ \mathrm{S}_{\mathcal{H}}(\mathfrak{s}2\mathfrak{h}_{\mathbb{N}}(k)) & \text{if } n =_{\mathcal{S}} \mathrm{S}_{\mathcal{S}}(k) \text{ for some } \mathcal{S}\text{-natural } k, \\ \mathit{undefined} & \text{otherwise.} \end{cases}$$

The functor and its inverse are formally defined in Isabelle as follows

fun *h2sn* :: *nat* $\Rightarrow$ *Set* ($\mathfrak{h}2\mathfrak{s}_{\mathbb{N}}$(_)) **where**
$\mathfrak{h}2\mathfrak{s}_{\mathbb{N}}$(*0*::*nat*) $=_{\mathcal{S}}$ $0_{\mathcal{S}}$ | $\mathfrak{h}2\mathfrak{s}_{\mathbb{N}}$(*Suc*(*x*)) $=_{\mathcal{S}}$ *succ* $\mathfrak{h}2\mathfrak{s}_{\mathbb{N}}$(*x*)

function *s2hn* :: *Set* $\Rightarrow$ *nat* ($\mathfrak{s}2\mathfrak{h}_{\mathbb{N}}$(_)) **where**
¬*x be Nat* $\Longrightarrow$ $\mathfrak{s}2\mathfrak{h}_{\mathbb{N}}$(*x*) $=_{\mathcal{H}}$ *undefined*
| $\mathfrak{s}2\mathfrak{h}_{\mathbb{N}}$($0_{\mathcal{S}}$) $=_{\mathcal{H}}$ *0*
| *x be Nat* $\Longrightarrow$ $\mathfrak{s}2\mathfrak{h}_{\mathbb{N}}$(*succ*(*x*)) $=_{\mathcal{H}}$ *Suc*($\mathfrak{s}2\mathfrak{h}_{\mathbb{N}}$(*x*))

Note that $\mathfrak{h}2\mathfrak{s}_{\mathbb{N}}$ is defined only on the HOL natural numbers (*nat*), while $\mathfrak{s}2\mathfrak{h}_{\mathbb{N}}$ is defined on all sets and its definition is only meaningful for arguments that are of the type *Nat*. The soft-type system of Mizar requires us to give this assumption explicitly here, but it can normally be hidden in the contexts where the argument type is restricted appropriately. Isabelle requires us to prove the termination of the definition, which can be done using the proper subset relation defined on natural numbers in the Peano sense.

Using the two induction principles for natural numbers present in both libraries, we can show that *beIsoS*($\mathfrak{h}2\mathfrak{s}_{\mathbb{N}}$, $\mathfrak{s}2\mathfrak{h}_{\mathbb{N}}$,*NAT*), where *NAT* is the set of all *Nat*. In particular it gives a bijection (note the hidden type restriction to sets of type *nat*). We show also that the functors preserve the basic operations on the natural numbers including addition, multiplication, comparison operators, division, primality, etc. The formalized statement is as follows:

theorem *Nat_to_Nat*:
fixes x::*nat* **and** y::*nat*
assumes n *be Nat* **and** m *be Nat*
shows $\mathfrak{h}2\mathfrak{s}_{\mathbb{N}}(x +_{\mathcal{H}} y) =_{\mathcal{S}} \mathfrak{h}2\mathfrak{s}_{\mathbb{N}}(x) +_{\mathcal{S}}{}^{\mathbb{N}} \mathfrak{h}2\mathfrak{s}_{\mathbb{N}}(y)$
$\mathfrak{s}2\mathfrak{h}_{\mathbb{N}}(n +_{\mathcal{S}}{}^{\mathbb{N}} m) =_{\mathcal{H}} \mathfrak{s}2\mathfrak{h}_{\mathbb{N}}(n) +_{\mathcal{H}} \mathfrak{s}2\mathfrak{h}_{\mathbb{N}}(m)$
$\mathfrak{h}2\mathfrak{s}_{\mathbb{N}}(x *_{\mathcal{H}} y) =_{\mathcal{S}} \mathfrak{h}2\mathfrak{s}_{\mathbb{N}}(x) *_{\mathcal{S}}{}^{\mathbb{N}} \mathfrak{h}2\mathfrak{s}_{\mathbb{N}}(y)$
$\mathfrak{s}2\mathfrak{h}_{\mathbb{N}}(n *_{\mathcal{S}}{}^{\mathbb{N}} m) =_{\mathcal{H}} \mathfrak{s}2\mathfrak{h}_{\mathbb{N}}(n) *_{\mathcal{H}} \mathfrak{s}2\mathfrak{h}_{\mathbb{N}}(m)$
$x < y \longleftrightarrow \mathfrak{h}2\mathfrak{s}_{\mathbb{N}}(x) \subset \mathfrak{h}2\mathfrak{s}_{\mathbb{N}}(y)$
$n \subset m \longleftrightarrow \mathfrak{s}2\mathfrak{h}_{\mathbb{N}}(n) < \mathfrak{s}2\mathfrak{h}_{\mathbb{N}}(m)$
x *dvd* $y \longleftrightarrow \mathfrak{h}2\mathfrak{s}_{\mathbb{N}}(x)$ *divides* $\mathfrak{h}2\mathfrak{s}_{\mathbb{N}}(y)$
n *divides* $m \longleftrightarrow \mathfrak{s}2\mathfrak{h}_{\mathbb{N}}(n)$ *dvd* $\mathfrak{s}2\mathfrak{h}_{\mathbb{N}}(m)$
$prime(x) \longleftrightarrow \mathfrak{h}2\mathfrak{s}_{\mathbb{N}}(x)$ *is* $prime_{\mathcal{S}}$
n *is* $prime_{\mathcal{S}} \longleftrightarrow prime(\mathfrak{s}2\mathfrak{h}_{\mathbb{N}}(n))$

It is now possible to translate the Lagrange's Four Squares theorem and Bertrand's postulate between the libraries. We can prove the Isabelle/Mizar counterpart of the Isabelle/HOL theorem only using higher-order rewriting and the above properties.

theorem *LagrangeFourSquares*:
$\forall\, n$:*Nat*. $\exists\, a,b,c,d$:*Nat*.
$a *_{\mathcal{S}}{}^{\mathbb{N}} a \; +_{\mathcal{S}}{}^{\mathbb{N}} \; b *_{\mathcal{S}}{}^{\mathbb{N}} b \; +_{\mathcal{S}}{}^{\mathbb{N}} \; c *_{\mathcal{S}}{}^{\mathbb{N}} c \; +_{\mathcal{S}}{}^{\mathbb{N}} \; d *_{\mathcal{S}}{}^{\mathbb{N}} d =_{\mathcal{S}} \; n$

theorem *Bertrand*:
$\forall\, n$:*Nat*. $1_{\mathcal{S}} \subset n \longrightarrow$
$(\exists\, p$:*Nat*. p *be* $prime_{\mathcal{S}} \wedge n \subset p \wedge p \subset (2_{\mathcal{S}} *_{\mathcal{S}}{}^{\mathbb{N}} n))$

Integers can be handled in an analogous way: the definitions are again different but it is straightforward to define a bijection between the two, and show that is preserves all the basic operators. For operators that are missing in one of the libraries, it is possible to actually lift their definitions. For example the exponentiation operation, which has not been considered in the Isabelle/Mizar library so far, can be defined as $TransformHS(\mathfrak{s}2\mathfrak{h}_{\mathbb{Z}},\mathfrak{s}2\mathfrak{h}_{\mathbb{N}},\mathfrak{h}2\mathfrak{s}_{\mathbb{Z}},(\widehat{\ }))$, where

definition *TransformHS* **where**
func TransformHS($\mathfrak{s}2\mathfrak{h}X1$,$\mathfrak{s}2\mathfrak{h}X2$,$\mathfrak{h}2\mathfrak{s}Y$,*HFun*,*x1*,*x2*) $\rightarrow$ *set equals*
$\mathfrak{h}2\mathfrak{s}Y(HFun(\mathfrak{s}2\mathfrak{h}X1(x1),\mathfrak{s}2\mathfrak{h}X2(x2)))$

This allows translating the proved Fermat's last theorem for powers divisible by 3 and 4 from Isabelle/HOL to Isabelle/Mizar. The proof involved quite some computation and therefore has not been attempted in Mizar so far.

theorem *Fermat_divides_3_4*:
$\forall\, x,y,z$:*Integer*. $\forall\, n$:*Nat*.
$(3_{\mathcal{S}}$ *divides* $n \vee 4_{\mathcal{S}}$ *divides* $n) \wedge x|\widehat{\ }n +_{\mathcal{S}}{}^{\mathbf{Z}} y|\widehat{\ }n =_{\mathcal{S}} z|\widehat{\ }n$
$\longrightarrow x *_{\mathcal{S}}{}^{\mathbf{Z}} y *_{\mathcal{S}}{}^{\mathbf{Z}} z =_{\mathcal{S}} 0_{\mathcal{S}}$

5.2 Polymorphic types and lists

Isabelle/HOL lists are realized as a polymorphic algebraic datatype, corresponding to functional programming language lists. MML lists (called finite sequences, `FinSequence`) are functions from an initial segment of the natural numbers. Higher-order lists behave like stacks, with access to the top of the stack, whereas for the set theoretic ones the natural operations are the restriction or extension of the domain.

To build a bijection between these types, we note that the *Cons* operator corresponds to the concatenation of a singleton list and the second argument. Since the list type is polymorphic (in the shallow polymorphism sense used in HOL), in order to build this bijection,

we also need to map the actual elements of the list. Therefore the bijection on lists will be parametric on a bijection on elements:

```
fun h2sfs :: (a ⇒ Set) ⇒ a List.list ⇒ Set (𝔥2𝔰_L(_,_))where
  𝔥2𝔰_L(𝔥2𝔰, Nil) =_𝒮 <*>
| 𝔥2𝔰_L(𝔥2𝔰, Cons(h, t)) =_𝒮 ((<*𝔥2𝔰(h)*>) ^ (𝔥2𝔰_L(𝔥2𝔰, t)))
```

The converse operation needs to separate the first element of a sequence from the rest and shift it by one. We define this operation in Isabelle/Mizar and complete the definition. Isabelle will again require us to show the termination of the function, which can be done by induction on the length of the list/sequence:

```
function s2hl :: (Set ⇒ a) ⇒ Set ⇒a List.list (𝔰2𝔥_L(_,_)) where
  ¬ x be FinSequence ⟹ 𝔰2𝔥_L(𝔰2𝔥,x) =_ℋ undefined
| 𝔰2𝔥_L(𝔰2𝔥,<*>) =_ℋ  Nil
| x be FinSequence ⟹ x ≠ <*> ⟹
    𝔰2𝔥_L(𝔰2𝔥,x) =_ℋ Cons (𝔰2𝔥(x.1_𝒮), 𝔰2𝔥_L(𝔰2𝔥,x/^1_𝒮 ))
```

For the transformation introduced above, we can show that if we have a good homomorphism between the elements of the lists, then lists over this type are homomorphic with finite sequences.

We can again show that this homomorphism preserves various basic operations, such as concatenation, the selection of n-th element, length, etc.

```
theorem s2hL_Prop:
  assumes p be FinSequence and q be FinSequence
    and n be Nat and n in len p
  shows size(𝔰2𝔥_L(𝔰2𝔥,p)) =_ℋ 𝔰2𝔥_ℕ(len p)
          𝔰2𝔥_L(𝔰2𝔥,p^q) =_ℋ 𝔰2𝔥_L(𝔰2𝔥,p) @ 𝔰2𝔥_L(𝔰2𝔥,q)
    𝔰2𝔥_L(𝔰2𝔥,p) ! 𝔰2𝔥_ℕ(n) =_ℋ 𝔰2𝔥(p. (succ n))
```

Another polymorphic type that we need to map are functions. Set theoretic functions (sets of pairs) correspond to higher-order functions and this homomorphism preserves function application.

```
theorem HtoSappl:
  assumes beIsoS(𝔥2𝔰d,𝔰2𝔥d,d) and beIsoS(𝔥2𝔰r,𝔰2𝔥r,r)
  shows 𝔥2𝔰_f(𝔰2𝔥d,𝔥2𝔰r,d,f).𝔥2𝔰d(x) =_𝒮 𝔥2𝔰r(f(x))
```

5.3 Algebra

The structure representations used in higher-order logic and set theories are usually different. This will be particularly visible when it comes to algebraic structures. In the Isabelle/HOL formalization algebraic structures are type-classes while in set theory a common approach would be partial functions. We will illustrate the difference on the example of groups. A type α forms a group when we can indicate a binary function on this type that will serve as the the group operation satisfying the group axioms. On the other hand, in the usual set-theoretic approach a group in set theory would consist of an explicitly given set (the carrier), and the group operation. With an intersection type system, the fact that the given set with an operation is a group is specified by intersecting the type of structures with the types that specify their individual properties (i.e. a group is a non-empty associative Group-like multMagma).

There are two more differences in the particular formalizations we consider, that we will not focus on, but we will only hint them in this paragraph and consider them only in the formalization. First, the existence and uniqueness of the neutral element can be either assumed in the group specification or derived from the axioms. Will not focus on that, as this is only the choice of a group axiomatization. Second, in the Mizar library there are two theories of groups: additive groups and multiplicative groups. Rings and fields inherit the latter, while some group-theoretic results are derived only for the former. Even if the Isabelle/HOL group includes a field for the unit, we will ignore it in the morphism, since the set theoretic definition does not use one. The neutral element along with the other properties is however necessary to justify that the result of the morphism is a group in the set theoretic sense.

definition *h2sg* ($\mathfrak{h}2\mathfrak{s}_G(\_,\_,\_,\_)$) **where**
$\mathfrak{h}2\mathfrak{s}_G(\mathfrak{s}2\mathfrak{h}c,\mathfrak{h}2\mathfrak{s}c,c,g) =_{\mathcal{S}}$ [#
carrier $\mapsto$ c;
multF $\mapsto$ $\mathfrak{h}2\mathfrak{s}_{BinOp}(\mathfrak{s}2\mathfrak{h}c,\mathfrak{h}2\mathfrak{s}c,c,mult(g))$ #]

definition *s2hg* ($\mathfrak{s}2\mathfrak{h}_G(\_,\_,\_)$) **where**
$\mathfrak{s}2\mathfrak{h}_G(\mathfrak{s}2\mathfrak{h}c,\mathfrak{h}2\mathfrak{s}c,g) =_{\mathcal{H}}$ *Igroup*(
Collect(λx. $\mathfrak{h}2\mathfrak{s}c(x)$ *in the carrier of* g),
$\mathfrak{s}2\mathfrak{h}_{BinOp}(\mathfrak{s}2\mathfrak{h}c,\mathfrak{h}2\mathfrak{s}c,$*the multF of* g),
$\mathfrak{s}2\mathfrak{h}c(1_g)$)

For the dual morphism, we indicate the result of the operation selecting the neutral element (1_g) as the element needed in the construction of the type-class element. With its help, we can justify that the fields of the translated structure are translation of the fields.

theorem *s2hg_Prop*:
assumes *beIsoS*($\mathfrak{h}2\mathfrak{s}c,\mathfrak{s}2\mathfrak{h}c,c$) **and** g *be Group*
and *the carrier of* $g =_{\mathcal{S}} c$
and $x \in$ *carrierI*($\mathfrak{s}2\mathfrak{h}_G(\mathfrak{s}2\mathfrak{h}c,\ \mathfrak{h}2\mathfrak{s}c,\ g)$)
$y \in$ *carrierI*($\mathfrak{s}2\mathfrak{h}_G(\mathfrak{s}2\mathfrak{h}c,\ \mathfrak{h}2\mathfrak{s}c,\ g)$)
shows *one*($\mathfrak{s}2\mathfrak{h}_G(\mathfrak{s}2\mathfrak{h}c,\mathfrak{h}2\mathfrak{s}c,g)$) $=_{\mathcal{H}}$ $\mathfrak{s}2\mathfrak{h}c(1_g)$
$x \otimes_{\mathfrak{s}2\mathfrak{h}_G(\mathfrak{s}2\mathfrak{h}c,\mathfrak{h}2\mathfrak{s}c,g)} y =_{\mathcal{H}} \mathfrak{s}2\mathfrak{h}c(\mathfrak{h}2\mathfrak{s}c(x) \otimes_g \mathfrak{h}2\mathfrak{s}c(y))$
group ($\mathfrak{s}2\mathfrak{h}_G(\mathfrak{s}2\mathfrak{h}c,\mathfrak{h}2\mathfrak{s}c,g)$)

A number of proof assistant systems based both on higher-order logic (including Isabelle/HOL) and set theory (including Mizar) support inheritance between their algebraic structures. As part of our work aligning the libraries we also want to verify that such inheritance is supported in the combined library. For this, we align the ring structures present in the two libraries. The isomorphism between the structures is defined in a similar way to the one for groups, we refer the interested reader to our formalization.

We can show that the morphisms form an isomorphism and derive some basic preservation properties. The most basic one is the fact that the isomorphism preserves being a ring.

theorem *s2hr_Prop*:
assumes *beIsoS*($\mathfrak{h}2\mathfrak{s}c,\mathfrak{s}2\mathfrak{h}c,c$) **and** r *be Ring*
and *the carrier of* $r =_{\mathcal{S}} c$
and $x \in$ *carrierI*($\mathfrak{s}2\mathfrak{h}_R(\mathfrak{s}2\mathfrak{h}c,\mathfrak{h}2\mathfrak{s}c,r)$)
$y \in$ *carrierI*($\mathfrak{s}2\mathfrak{h}_R(\mathfrak{s}2\mathfrak{h}c,\mathfrak{h}2\mathfrak{s}c,r)$)
shows *zero*($\mathfrak{s}2\mathfrak{h}_R(\mathfrak{s}2\mathfrak{h}c,\mathfrak{h}2\mathfrak{s}c,r)$) $=_{\mathcal{H}}$ $\mathfrak{s}2\mathfrak{h}c(0_r)$
one($\mathfrak{s}2\mathfrak{h}_R(\mathfrak{s}2\mathfrak{h}c,\mathfrak{h}2\mathfrak{s}c,r)$) $=_{\mathcal{H}}$ $\mathfrak{s}2\mathfrak{h}c(1_r)$
$x \oplus_{\mathfrak{s}2\mathfrak{h}_R(\mathfrak{s}2\mathfrak{h}c,\mathfrak{h}2\mathfrak{s}c,r)} y =_{\mathcal{H}} \mathfrak{s}2\mathfrak{h}c(\mathfrak{h}2\mathfrak{s}c(x) \oplus_r \mathfrak{h}2\mathfrak{s}c(y))$
$x \otimes_{\mathfrak{s}2\mathfrak{h}_R(\mathfrak{s}2\mathfrak{h}c,\mathfrak{h}2\mathfrak{s}c,r)} y =_{\mathcal{H}} \mathfrak{s}2\mathfrak{h}c(\mathfrak{h}2\mathfrak{s}c(x) \otimes_r \mathfrak{h}2\mathfrak{s}c(y))$
ring ($\mathfrak{s}2\mathfrak{h}_R(\mathfrak{s}2\mathfrak{h}c,\mathfrak{h}2\mathfrak{s}c,r)$)

Finally, we introduce the equivalent of the definition of the integer ring introduced in the MML in [41]. We show that $\mathfrak{s}2\mathfrak{h}_R$ and $\mathfrak{h}2\mathfrak{i}_R$ determine an isomorphism between the fields of the rings developed in Isabelle/HOL and the Mizar Mathematical Library.

mdef *int_3_def_3* (*$\mathbb{Z}$-ring*) **where**
func $\mathbb{Z}$*-ring* $\rightarrow$ *strict*(*doubleLoopStr*) *equals* [#
carrier $\mapsto$ *INT*;
addF $\mapsto$ *addint*;
ZeroF $\mapsto$ $0_{\mathcal{S}}$;
multF $\mapsto$ *multint*;
OneF $\mapsto$ $1_{\mathcal{S}}$#]

theorem *H_Zring_to_S_Zring*:
$\mathfrak{h}2\mathfrak{s}_R(\mathfrak{s}2\mathfrak{h}_{\mathbf{Z}}, \mathfrak{h}2\mathfrak{s}_{\mathbf{Z}}, INT, \mathcal{Z}) =_{\mathcal{S}} \mathbb{Z}$*-ring*
$\mathfrak{s}2\mathfrak{h}_R(\mathfrak{s}2\mathfrak{h}_{\mathbf{Z}}, \mathfrak{h}2\mathfrak{s}_{\mathbf{Z}}, \mathbb{Z}$*-ring*$) =_{\mathcal{H}} \mathcal{Z}$

6 Related Work

As proof assistants based on plain higher-order logic lack the full expressivity of set theory, the idea of adding set theory axioms on top of HOL (without a model) has been tried multiple times. Gordon [15] discusses approaches to combine the power of HOL and set theory. Obua has proposed HOLZF [34], where Zermelo-Fraenkel axioms are added on top of Isabelle/HOL. With this, he was able to show results on partisan games, that would be hard to show in plain higher-order logic. Later, as part of the ProofPeer project [35], the combination of HOL with ZF became the basis for an LCF system, reducing the proofs in higher-order logic part to a minimum (again, since there was no guarantee, that combining the results is safe). Kunčar [30] attempted to import the Tarski-Grothendieck-based library into HOL Light. Here, the set-theoretic concepts were immediately mapped to their HOL counterparts, but it soon came out that without adding the axioms of set theory they system was not strong enough. The first author, Brown [10] proposed the Egal system which again combines a specification of higher-order logic with the axioms of set theory. The system uses explicit universes, which is in fact the same presentation as given in this work. This work therefore also gives a model for the Egal system. Finally, second and third authors [24] have specified and imported [23] significant parts of the Mizar library into Isabelle. In this work we only use the specification of Mizar in Isabelle and the re-formalized parts of the MML.

The idea to combine proof assistant libraries across different foundations also arose in the Flyspeck project [18] formalizing the proof of the Kepler conjecture. There, the dependency on Coq has been eliminated and an ad-hoc justification for the concepts moved between Isabelle and HOL was specified. Logical frameworks allow importing multiple libraries at the same time, again without a model. In the Dedukti framework, Assaf and Cauderlier [3, 4] have combined properties originating from the Coq library and the HOL library. Both were imported in the same system, based on the $\lambda\Pi$ calculus modulo, however the two parts of the library relied on different rewrite rules. Krauss and Schropp [29] specified and implemented a translation from Isabelle/HOL proof terms to set theoretic proved theorems. The translation is sound and only relies on the Isabelle/ZF logic, however it is too slow to be useful in practice, in fact it is not possible to translate the basic Main library of Isabelle/HOL into set theory in reasonable time. It also possible to deep embed multiple libraries in a single meta-theory. Rabe [40] does this practically in the MMT framework deep embedding various proof assistant foundations and providing category-theoretic mappings between some foundations.

Most implementation of set theory in logical frameworks could implicitly use some higher-order features of the framework, as this is already used for the definition of the object logic. The definition of the Zermelo-Fraenkel object logic [36] in Isabelle uses lambda abstractions and higher-order applications for example to specify the quantifiers. This is also the case in Isabelle/TLA [31]. These object logics are normally careful to restrict the use of higher-order features to a minimum, however the system itself does not restrict this usage.

The second author together with Gauthier [14] has previously proposed heuristics for automatically finding alignments across proof assistant libraries. Such alignments, even without merging the libraries can be useful for conjecturing new properties [32] as well as to improve proof assistant automation [13].

7 Conclusion

We have defined a model of higher-order Tarski-Grothendieck. The model relies on a 2-inaccessible cardinal, which is the same assumption as the one required for a model of a TG set theory. This model shows that it is safe to combine higher-order features with the axioms of set theory, which has already been done by a number of developments [10, 24, 34, 35].

Moreover, thanks to the model we can safely combine results proved in TG set theory with ones proved in plain higher-order logic. We benefit from this, by combining two of the largest proof assistant libraries: the Mizar Mathematical library and the Isabelle/HOL library. Above the theorems and proofs coming from both, we define a number of isomorphisms that allow us to translate theorems proved in of these part of the library and use them in the other part.

As part of the library merging we have formally defined and proved in Isabelle the necessary concepts. This involved 18 definitions and 135 theorems, which amounts to 2667 lines of proofs.

Apart from higher-order and set-theoretic foundations, the third most commonly used foundation is dependent type theory. The most important future work would be to investigate the consistency of a theory that imports such foundations as well.

References

1 P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Kluwer Academic Publishers, 2nd edition, 2002.

2 Peter B. Andrews. General Models and Extensionality. *J. Symb. Log.*, 37:395–397, 1972.

3 Ali Assaf. *A framework for defining computational higher-order logics. (Un cadre de définition de logiques calculatoires d'ordre supérieur)*. PhD thesis, École Polytechnique, Palaiseau, France, 2015. URL: `https://tel.archives-ouvertes.fr/tel-01235303`.

4 Ali Assaf and Raphaël Cauderlier. Mixing HOL and Coq in Dedukti. In Cezary Kaliszyk and Andrei Paskevich, editors, *Proof eXchange for Theorem Proving (PxTP 2015)*, volume 186 of *EPTCS*, pages 89–96, 2015.

5 Julian Backes and Chad E. Brown. Analytic Tableaux for Higher-Order Logic with Choice. *Journal of Automated Reasoning*, 47(4):451–479, 2011.

6 Grzegorz Bancerek, Czesław Byliński, Adam Grabowski, Artur Korniłowicz, Roman Matuszewski, Adam Naumowicz, and Karol Pąk. The Role of the Mizar Mathematical Library for Interactive Proof Development in Mizar. *Journal of Automated Reasoning*, 2017. `doi:10.1007/s10817-017-9440-6`.

7 Grzegorz Bancerek and Piotr Rudnicki. A Compendium of Continuous Lattices in MIZAR. *J. Autom. Reasoning*, 29(3-4):189–224, 2002. URL: `http://doi.org/10.1023/A:1021966832558`.

8 Christoph Benzmüller, Chad E. Brown, and Michael Kohlhase. Higher-order semantics and extensionality. *J. Symb. Log.*, 69:1027–1088, 2004.

9 Jasmin Christian Blanchette, Maximilian Haslbeck, Daniel Matichuk, and Tobias Nipkow. Mining the Archive of Formal Proofs. In Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge, editors, *Intelligent Computer Mathematics (CICM 2015)*, volume 9150 of *LNCS*, pages 3–17. Springer, 2015. doi:10.1007/978-3-319-20615-8_1.

10 Chad E. Brown. *The Egal Manual*, 2014. URL: http://grid01.ciirc.cvut.cz/~chad/egalmanual.pdf.

11 Chad E. Brown and Gert Smolka. Analytic Tableaux for Simple Type Theory and its First-Order Fragment. *Logical Methods in Computer Science*, 6(2), June 2010.

12 Alonzo Church. A Formulation of the Simple Theory of Types. *J. Symb. Log.*, 5:56–68, 1940.

13 Thibault Gauthier and Cezary Kaliszyk. Sharing HOL4 and HOL Light Proof Knowledge. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *20th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2015)*, volume 9450 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2015. doi:10.1007/978-3-662-48899-7_26.

14 Thibault Gauthier and Cezary Kaliszyk. Aligning Concepts across Proof Assistant Libraries. *J. Symbolic Computation*, 90:89–123, 2019. doi:10.1016/j.jsc.2018.04.005.

15 Michael Gordon. Set Theory, Higher Order Logic or Both? In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics, TPHOLs'96*, volume 1125 of *LNCS*, pages 191–201. Springer, 1996. doi:10.1007/BFb0105405.

16 Adam Grabowski, Artur Korniłowicz, and Adam Naumowicz. Four Decades of Mizar. *Journal of Automated Reasoning*, 55(3):191–198, 2015. doi:10.1007/s10817-015-9345-1.

17 A. Grothendieck and J.-L. Verdier. *Théorie des topos et cohomologie étale des schémas - (SGA 4) - vol. 1*, volume 269 of *Lecture notes in mathematics*. Springer-Verlag, 1972.

18 Thomas C. Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Le Truong Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, Quang Truong Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason M. Rute, Alexey Solovyev, Thi Hoai An Ta, Nam Trung Tran, Thi Diep Trieu, Josef Urban, Ky Vu, and Roland Zumkeller. A Formal Proof of the Kepler conjecture. *Forum of Mathematics, Pi*, 5, 2017. doi:10.1017/fmp.2017.1.

19 Leon Henkin. Completeness in the Theory of Types. *J. Symb. Log.*, 15:81–91, 1950.

20 Peter V. Homeier. A Design Structure for Higher Order Quotients. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 130–146. Springer, 2005. doi:10.1007/11541868_9.

21 Brian Huffman and Ondrej Kuncar. Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, volume 8307 of *LNCS*, pages 131–146. Springer, 2013. doi:10.1007/978-3-319-03545-1_9.

22 Cezary Kaliszyk and Karol Pąk. Isabelle Formalization of Set Theoretic Structures and Set Comprehensions. In Johannes Blamer, Temur Kutsia, and Dimitris Simos, editors, *Mathematical Aspects of Computer and Information Sciences, MACIS 2017*, volume 10693 of *LNCS*. Springer, 2017. doi:10.1007/978-3-319-72453-9_12.

23 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, *11th International Conference on Intelligent Computer Mathematics (CICM 2018)*, volume 11006 of *LNCS*, pages 131–146. Springer, 2018. doi:10.1007/978-3-319-96812-4_13.

24 Cezary Kaliszyk and Karol Pąk. Semantics of Mizar as an Isabelle Object Logic. *Journal of Automated Reasoning*, 2018. doi:10.1007/s10817-018-9479-z.

25 Cezary Kaliszyk, Karol Pąk, and Josef Urban. Towards a Mizar Environment for Isabelle: Foundations and Language. In Jeremy Avigad and Adam Chlipala, editors, *Proc. 5th Conference on Certified Programs and Proofs (CPP 2016)*, pages 58–65. ACM, 2016. doi:10.1145/2854065.2854070.

26 Cezary Kaliszyk and Christian Urban. Quotients revisited for Isabelle/HOL. In William C. Chu, W. Eric Wong, Mathew J. Palakal, and Chih-Cheng Hung, editors, *Proc. of the 26th ACM Symposium on Applied Computing (SAC'11)*, pages 1639–1644. ACM, 2011.

27 Akihiro Kanamori. *The higher infinite: Large cardinals in set theory from their beginnings*. Springer Monographs in Mathematics. Springer-Verlag Berlin Heidelberg, 2 edition, 2003.

28 Dominik Kirst and Gert Smolka. Large Model Constructions for Second-Order ZF in Dependent Type Theory. *Certified Programs and Proofs - 7th International Conference, CPP 2018, Los Angeles, USA, January 8-9, 2018*, January 2018.

29 Alexander Krauss and Andreas Schropp. A Mechanized Translation from Higher-Order Logic to Set Theory. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving (ITP 2010)*, volume 6172 of *LNCS*, pages 323–338. Springer, 2010.

30 Ondřej Kunčar. Reconstruction of the Mizar Type System in the HOL Light System. In Jiri Pavlu and Jana Safrankova, editors, *WDS Proceedings of Contributed Papers: Part I – Mathematics and Computer Sciences*, pages 7–12. Matfyzpress, 2010.

31 Stephan Merz. Mechanizing TLA in Isabelle. In Robert Rodošek, editor, *Workshop on Verification in New Orientations*, pages 54–74, Maribor, 1995. Univ. of Maribor.

32 Dennis Müller, Thibault Gauthier, Cezary Kaliszyk, Michael Kohlhase, and Florian Rabe. Classification of Alignments Between Concepts of Formal Mathematical Systems. In Herman Geuvers, Matthew England, Osman Hasan, Florian Rabe, and Olaf Teschke, editors, *10th International Conference on Intelligent Computer Mathematics (CICM'17)*, volume 10383 of *LNCS*, pages 83–98. Springer, 2017. doi:10.1007/978-3-319-62075-6_7.

33 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

34 Steven Obua. Partizan Games in Isabelle/HOLZF. In Kamel Barkaoui, Ana Cavalcanti, and Antonio Cerone, editors, *Theoretical Aspects of Computing - ICTAC 2006*, volume 4281 of *LNCS*, pages 272–286. Springer, 2006.

35 Steven Obua, Jacques D. Fleuriot, Phil Scott, and David Aspinall. ProofPeer: Collaborative theorem proving. *CoRR*, abs/1404.6186, 2014. arXiv:1404.6186.

36 Lawrence C. Paulson. Set Theory for Verification: I. From foundations to functions. *J. Autom. Reasoning*, 11(3):353–389, 1993. doi:10.1007/BF00881873.

37 Karol Pąk. Brouwer Fixed Point Theorem in the General Case. *Formalized Mathematics*, 19(**3**):151–153, 2011. doi:10.2478/v10037-011-0024-3.

38 Karol Pąk. Brouwer Invariance of Domain Theorem. *Formalized Mathematics*, 22(**1**):21–28, 2014. doi:10.2478/forma-2014-0003.

39 Karol Pąk. Topological Manifolds. *Formalized Mathematics*, 22(**2**):179–186, 2014. doi:10.2478/forma-2014-0019.

40 Florian Rabe. How to identify, translate and combine logics? *J. Log. Comput.*, 27(6):1753–1798, 2017. doi:10.1093/logcom/exu079.

41 Christoph Schwarzweller. The Ring of Integers, Euclidean Rings and Modulo Integers. *Formalized Mathematics*, 8(**1**):29–34, 1999.

42 Alfred Tarski. Über unerreichbare Kardinalzahlen. *Fundamenta Mathematica*, 30:68–89, 1938. URL: http://matwbn.icm.edu.pl/ksiazki/fm/fm30/fm30113.pdf.

43 Andrzej Trybulec. Tarski Grothendieck Set Theory. *Journal of Formalized Mathematics*, Axiomatics, 2002. Released 1989.

44 Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle Framework. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008*, volume 5170 of *LNCS*, pages 33–38. Springer, 2008. doi:10.1007/978-3-540-71067-7_7.

Generic Authenticated Data Structures, Formally

Matthias Brun
Department of Computer Science, ETH Zürich, Switzerland
mbrun@student.ethz.ch

Dmitriy Traytel
Institute of Information Security, Department of Computer Science, ETH Zürich, Switzerland
traytel@inf.ethz.ch

Abstract

Authenticated data structures are a technique for outsourcing data storage and maintenance to an untrusted server. The server is required to produce an efficiently checkable and cryptographically secure proof that it carried out precisely the requested computation. Recently, Miller et al. [10] demonstrated how to support a wide range of such data structures by integrating an authentication construct as a first class citizen in a functional programming language. In this paper, we put this work to the test of formalization in the Isabelle proof assistant. With Isabelle's help, we uncover and repair several mistakes and modify the small-step semantics to perform call-by-value evaluation rather than requiring terms to be in administrative normal form.

2012 ACM Subject Classification Security and privacy → Logic and verification

Keywords and phrases Authenticated Data Structures, Verifiable Computation, Isabelle/HOL, Nominal Isabelle

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.10

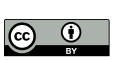
Supplement Material `https://isa-afp.org/entries/LambdaAuth.html`

Acknowledgements We thank David Basin for supporting this work and Andrew Miller for discussing our counterexamples and proposing a remedy to the issue with type soundness. Joshua Schneider and the anonymous ITP reviewers helped to improve the presentation through numerous comments.

1 Introduction

Consider a client that requests data from a server and trusts the server to answer its request truthfully, making financial or security-critical decisions based on the response. In this common scenario, a malicious actor can profit from causing the server to give incorrect answers to a client's query. Authenticated data structures (ADS) prevent this attack by effectively removing the need for the client to trust the server. To do so, they require the server to accompany all responses to queries with an efficiently verifiable proof that its answer is honest.

Merkle trees [9] are the prototypical example of ADS. They are binary trees that store data in their leaves. Every leaf node is augmented with a hash of the corresponding data and every inner node is augmented with a hash of its child nodes' hashes. An example Merkle tree is shown in Figure 1. The server stores this entire tree, whereas the client only stores the top hash H_0. The client can then query the server for any of the stored data. The server, upon being queried, traverses the tree to find the requested data and returns it along with the hashes needed to reconstruct the root hash. The client can then recompute the root hash to verify that it matches its stored root hash. In our example, querying the server for D_2 would result in it returning D_2 as well as the hashes HD_1 and H_2. The client can then verify that the result of $\mathsf{hash}\ (\mathsf{hash}\ (HD_1 \parallel \mathsf{hash}\ D_2) \parallel H_2)$ matches its stored root hash.



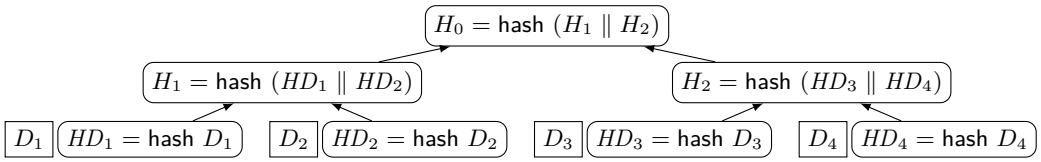


Figure 1 An example Merkle tree.

Early work on ADS [5,9,16] has focused on designing particular data structures for this purpose. More recently, Miller et al. [10] have put forward a more general view on the matter. In their paper, titled *Authenticated Data Structures, Generically* (*ADSG*), they introduce λ• (pronounced *lambda auth*), a purely functional language, which supports generic, user-specified ADS. The programs of λ• run in two modes. The server, which hosts the data, computes certain hash values and sends them to the client. The client verifies that the passed hash values are the expected ones. ADSG establishes correctness (verification succeeds if both parties correctly follow the protocol) and security (tricking the client requires discovering a hash collision) for all well-typed λ• programs. Given that ADS are intended to be used in security-critical applications, it is crucial that these correctness and security properties do in fact hold.

We formalized λ• in Isabelle/HOL and proved the claims stated in *ADSG*. During the formalization process, we identified several problems, many of which we rectified with relative ease. Nevertheless, a serious problem prevents us from reaching a fully satisfactory statement and proof of the conventional formulation of λ•'s type soundness.

In addition to finding and correcting mistakes, we also make a modification to the language semantics. "To keep the semantics simple," *ADSG* works with expressions in administrative normal form (ANF) [6]. ANF only supports recursive evaluation in arguments of let expressions and thus requires all other constructs to be applied to values (rather than unevaluated expressions). While this does not make the language any less powerful, the restrictive syntax makes λ• somewhat cumbersome to use, e.g., instead of writing $t\ u$ for expressions t and u one has to write $\mathsf{let}\ f = t\ \mathsf{in}\ \mathsf{let}\ x = u\ \mathsf{in}\ f\ x$. To hide this verbosity from the user, arbitrary expressions are typically translated into ANF in a separate step. However, such a translation would need to correctly handle λ•'s authentication construct. Instead, we extended the semantics to permit recursive argument evaluation for most expressions. We have performed this modification only after finishing the formalization of λ• and proving all the theorems for the ANF semantics. Isabelle allowed us to quickly discover all the ramifications of our changes. Thus, correcting the proofs that were affected by the modification was a matter of a few hours. In the following, we present only the modified semantics that supports recursive evaluation.

On the technical side, we used Nominal Isabelle [8,17] (Section 2) to model the syntax and semantics of λ• (Section 3), which involves several variable binding constructs. Of particular interest is our abstract modeling of a hash function that is compatible with Nominal and can be used in binding-aware definitions (Subsection 3.1). The small-step semantics of λ• is split into three transition relations that correspond to the client's, the server's, and an idealized view of the computation, respectively. Following *ADSG*, we relate programs evaluated under these three views using an inductive predicate (Section 4) and prove that if one of the related programs takes a step, the others can follow, unless a hash collision occurred (Section 5).

Related Work. *ADSG* [10] is our object of study. While our paper aspires to be self-contained with respect to the scope of the formalization, we refer to *ADSG* for the illuminating usages of the λ• language to implement Merkle trees, blockchains, and authenticated red-black trees.

The literature on formal studies of authenticated data structures is sparse, and in all cases focused on specific instances. Examples include the automatic verification of Merkle trees using weak monadic second-order logic on trees [12] and the formalization of blockchains [15] and cryptographic ledgers [20] (based on Merkle trees) in the Coq proof assistant. The two latter works both assume injective hash functions, which we avoid (Subsection 3.1).

A key feature of our formalization is the use of Nominal Isabelle [8, 17, 19], Isabelle's implementation of Nominal logic [7] on top of higher-order logic, to model a syntax involving binding of variables. More precisely, we use Nominal2 [8,17], the most recent implementation of Nominal Isabelle, which has previously been employed successfully in formalizations of Gödel's incompleteness theorems [13], lazy programming language semantics [3], and rewriting [11].

A frequently used alternative to the Nominal approach of modeling bound variables are de Bruijn indices, i.e., nameless pointers to binding constructors. We chose Nominal because it allows us to work more abstractly, without the need to manipulate pointers. We refer to Urban and Berghofer [18] for a comparison of the two approaches and to Blanchette et al. [2] for an extensive overview of the issue of binding variables in proof assistants and beyond.

2 Nominal Isabelle

The treatment of bound variables in pen and paper proofs is often informal, with renaming of clashing variables being implicitly assumed for most definitions. *ADSG* is no exception in this regard. In a formalization, a more rigorous approach is necessary. *Nominal Logic* [7] is a powerful such approach that is well-supported in Isabelle with the *Nominal* framework [8,17]. We sketch the most important features of Nominal and refer to Huffman and Urban [8] for a more extensive introduction.

Nominal allows us to closely follow the informal presentation of *ADSG* in the formalization by enforcing the Barendregt convention [1, p. 26]:

> If $M_1, \ldots, M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.

A central notion for achieving this flexibility is that of an object's support supp, which corresponds to the set of *atoms* (i.e., variable names) that occur free in it. An atom a outside of the support of x is *fresh* in x, written $a \sharp x \equiv a \notin \mathsf{supp}\ x$. We will use two kinds of atoms: type variables *tvar* and term variables *var*, which are embedded into the type of atoms using the overloaded function atom. We will often see statements of the kind $\mathsf{atom}\ a \sharp x$ in the premises of our definitions, making explicit the requirement that some (type) variable name a does not clash with any of the ones in x. These additional freshness assumptions are typically the only required modifications to an informal lemma's statement.

Nominal Isabelle provides commands for defining binding-aware datatypes, recursive functions, and inductive predicates, along with a proof method for performing binding-aware structural induction. The syntax of $\lambda\bullet$ (types *ty* and terms *term*), shown in Figure 2, is defined via the **nominal_datatype** command, which requires us to explicitly specify which names are bound in which constructors. For $\lambda\bullet$'s terms these are Lam, Rec, and Let, which model lambda abstractions (i.e., $\lambda x.\ t$ is written as $\mathsf{Lam}\ x\ t$), recursive functions, and let expressions, respectively, as well as Mu for recursive types. To define functions on a Nominal datatype we use the **nominal_function** command. The syntax for Nominal function definitions is the same as for normal functions except that freshness assumptions may be added when operating on datatype constructors that bind variables. For example,

nominal_datatype *term* =
  Unit
| Var *var*
| Lam (*x* :: *var*) (*t* :: *term*) **binds** *x* **in** *t*
| Rec (*x* :: *var*) (*t* :: *term*) **binds** *x* **in** *t*
| Inj1 *term*
| Inj2 *term*
| Pair *term term*
| Let *term* (*x* :: *var*) (*t* :: *term*) **binds** *x* **in** *t*
| App *term term*
| Case *term term term*
| Prj1 *term*
| Prj2 *term*
| Roll *term*
| Unroll *term*
| Auth *term*
| Unauth *term*
| Hash *hash*
| Hashed *hash term*

nominal_datatype *ty* =
  One
| Fun *ty ty*
| Sum *ty ty*
| Prod *ty ty*
| Mu (*α* :: *tvar*) (*τ* :: *ty*) **binds** *α* **in** *τ*
| Alpha *tvar*
| AuthT *ty*

inductive *value* :: *term* ⇒ *bool* **where**
  value Unit
| value (Var *x*)
| value (Lam *x e*)
| value (Rec *x e*)
| value *v* ⟶ value (Inj1 *v*)
| value *v* ⟶ value (Inj2 *v*)
| value v_1 ∧ value v_2 ⟶ value (Pair v_1 v_2)
| value *v* ⟶ value (Roll *v*)
| value (Hash *h*)
| value *v* ⟶ value (Hashed *h v*)

Figure 2 Syntax for terms and types.

the Lam case of the definition for capture-avoiding substitution, written $t[t'/x]$ and read as "in t substitute t' for x," is the following.

$$\text{atom } y \sharp (x, t') \longrightarrow (\text{Lam } y\ t)[t'/x] = \text{Lam } y\ (t[t'/x])$$

Definitions of inductive predicates use similar premises, as can be seen for example in our typing judgment's Lam rule in Figure 4. To enable binding-aware proofs by rule induction, Nominal can be instructed to prove a strong induction rule (after the user discharges a simpler abstract property, which is automatic for most definitions). The strong induction rule guarantees the absence of name clashes with a finite but arbitrary set of atoms.

Nominal is designed to support user-defined types as long as all objects have finite support. A particularly useful type for us will be that of finite maps, written (α, β) *fmap*, to model type environments and parallel substitutions. Finite maps are defined as the subtype of functions $\alpha \Rightarrow \beta$ *option* that map all but finitely many arguments to None. Other formalizations use association lists to represent type environments [18]. However, to ensure that any key in the list occurs at most once these require a validity predicate, cluttering the rules and proofs with implementation details. Finite maps nicely complemented our use of Nominal and allowed us to keep the statements of definitions and lemmas very close to those in *ADSG*. We use the syntax $\varnothing$ to denote the empty finite map, $\Gamma[x]$ to denote a lookup of x in the finite map Γ and $\Gamma[x \mapsto a]$ to denote an update to the finite map Γ, assigning a to x.

3 Syntax and Semantics of λ•

We formalize the terms and types for λ• as Nominal datatypes, along with an inductive predicate specifying which terms are considered to be values. These are listed in Figure 2. The terms and types are those of a standard lambda calculus with unit (One), product (Prod), sum (Sum), and recursive types (Mu), and the corresponding term constructors (e.g.,

nominal_function *shallow* :: *term* ⇒ *term* (⦇_⦈) **where**

⦇Unit⦈	= Unit	⦇Var v⦈	= Var v
⦇Lam x e⦈	= Lam x ⦇e⦈	⦇Rec x e⦈	= Rec x ⦇e⦈
⦇Inj1 e⦈	= Inj1 ⦇e⦈	⦇Inj2 e⦈	= Inj2 ⦇e⦈
⦇Pair e_1 e_2⦈	= Pair ⦇e_1⦈ ⦇e_2⦈	⦇Roll e⦈	= Roll ⦇e⦈
⦇Let e_1 x e_2⦈	= Let ⦇e_1⦈ x ⦇e_2⦈	⦇App e_1 e_2⦈	= App ⦇e_1⦈ ⦇e_2⦈
⦇Case e e_1 e_2⦈	= Case ⦇e⦈ ⦇e_1⦈ ⦇e_2⦈	⦇Prj1 e⦈	= Prj1 ⦇e⦈
⦇Prj2 e⦈	= Prj2 ⦇e⦈	⦇Unroll e⦈	= Unroll ⦇e⦈
⦇Auth e⦈	= Auth ⦇e⦈	⦇Unauth e⦈	= Unauth ⦇e⦈
⦇Hash h⦈	= Hash h	⦇Hashed h e⦈	= Hash h

Figure 3 The shallow projection.

Roll, the constructor of recursive types) and their inverses (e.g., Unroll, the destructor of recursive types) [14]. They also include the non-standard AuthT type constructor, Auth and Unauth term constructors, and auxiliary constructors Hashed consisting of a hash-value pair and Hash consisting of just a hash. We postpone the discussion of hash values and the type *hash* and introduce a few auxiliary functions first. Also the precise meaning of the Auth and Unauth constructors will become clear once we formally define the small-step semantics. Intuitively, Auth signals the client and server to compute a hash value, while Unauth signals the server to output a value to the client and the client to verify the hash of this value.

Substitution on terms and on types uses the syntax $t[u/x]$ for both. The definitions are standard, with simple, structural recursion on the non-standard constructs:

(Auth t)$[u/x]$ = Auth ($t[u/x]$)	(Unauth t)$[u/x]$ = Unauth ($t[u/x]$)
(Hash h)$[u/x]$ = Hash h	(Hashed h t)$[u/x]$ = Hashed h ($t[u/x]$)

Furthermore, we define a parallel substitution function psubst :: *term* ⇒ (*var*, *term*) *fmap* ⇒ *term*. It replaces all variables by terms assigned by the finite map given as its second argument:

psubst (Var y) Δ = (**case** $\Delta[y]$ **of** Some t ⇒ t | None ⇒ Var y)

For all other cases it is structurally recursive.

A *closed* term is one with empty support or, equivalently, closed t = ($\forall x$ :: *var*. atom x ♯ t).

ADSG also introduces the *shallow projection* function, written ⦇_⦈, whose formal definition is given in Figure 3. It replaces all Hashed h v subterms in a given term with Hash h.

3.1 Modeling the Hash Function

The security of λ• relies on a collision-resistant hash function. *ADSG* provides a useful modeling trick, which permits us to omit the formalization of this assumption or collision-resistance in general. In our formalization, we use very mild assumptions on how the hash function may behave. Our security statement is then a disjunction between the statements "everything worked out as planned" and "a hash collision has occurred." Clearly, if we use a collision-resistant hash function, the second disjunct will be violated with high probability. (This meta-argument is not captured in our formal modeling.)

We start by introducing a new type: **typedecl** *hash*. The only property we require of this type is that it does not contain any atoms, which we obtain by instantiating the *pure* type class. Doing so allows us to make use of the following lemma with α = *hash*.

▶ **Lemma 1** (No atoms occur in pure types)**.**

atom x ♯ (t :: α :: *pure*)

Because our desired hash function hash :: *term* ⇒ *hash* will be used in inductive predicates involving the *term* type, such as the small-step semantics, Nominal requires it to be *equivariant*, i.e., satisfy the strong property $\forall p.\ p \bullet \mathsf{hash}\ t = \mathsf{hash}\ (p \bullet t)$ for all terms t. Here, p is a permutation, i.e., a variable renaming, and $\bullet$ denotes its application to a an arbitrary object. (The application to the object's variables is defined by instantiating a type class, which is automatic for Nominal datatypes.) Since a hash contains no free variables, applying a permutation to it is the identity function. Clearly then, equivariance can *only* hold if permuting free variables does not change the hash – a counterintuitive requirement for a hash function, which we want to avoid.

For closed terms t the above property holds for any function hash. Moreover, it turns out that we will only apply hash to closed terms. Nominal, however, is blind to this fact and still requires us to prove equivariance for all terms. These two observations lead to the following solution. We declare a hash function using Isabelle's **consts** command, which introduces a new constant symbol without providing any specification of the constant beyond its type.

consts hash_term :: *term* ⇒ *hash*

This function is not necessarily equivariant. (We can neither prove nor disprove this.) Equivariance is established by composing hash_term with the function collapse_frees :: *term* ⇒ *term*, which maps all free variables of a term to a single fixed variable (definition omitted).

definition hash :: *term* ⇒ *hash* **where** hash = hash_term ∘ collapse_frees

The function hash is equivariant ($\forall p.\ p \bullet \mathsf{hash}\ t = \mathsf{hash}\ (p \bullet t)$) and equal to hash_term on closed terms (closed $t \longrightarrow \mathsf{hash}\ t = \mathsf{hash\_term}\ t$), because collapse_frees $t = t$ on closed terms t. Whenever we make use of the hash function hash, we ensure that its argument is closed.

3.2 Typing Judgement

The typing judgment $\Gamma \vdash e : \tau$, read "given the type environment Γ :: (*var*, *ty*) *fmap* the term e is well-typed and has type τ," for $\lambda\bullet$ is defined in Figure 4. The rules are standard except for the last two, which allow the introduction and elimination of authenticated types $\mathsf{AuthT}\ \tau$ via the Auth and Unauth constructors. In other words, these two rules fix the following types for the authentication constructors: $\mathsf{Auth} :: \tau \Rightarrow \mathsf{AuthT}\ \tau$ and $\mathsf{Unauth} :: \mathsf{AuthT}\ \tau \Rightarrow \tau$.

In addition to this typing judgment, we define an alternative, weaker typing judgment $\Gamma \vdash_W e : \tau$, which is not present in *ADSG*. This version replaces the last two rules with the ones in Figure 5, which do not introduce authenticated types, i.e., fixing $\mathsf{Auth} :: \tau \Rightarrow \tau$ and $\mathsf{Unauth} :: \tau \Rightarrow \tau$. This modification is motivated by an ambiguity in *ADSG*, which we will encounter when discussing type soundness. We use the unqualified *well-typed* to mean well-typed according to the original typing judgment and *weakly well-typed* to mean well-typed according to the modified rules.

Neither well-typed nor weakly well-typed terms may contain the Hashed and Hash term constructors, as there is no rule for them. These auxiliary constructors will arise only as the result of some computations and are not meant to be used as a language construct by the end-users of $\lambda\bullet$. Thus, the use of these constructors loosely resembles the use of memory locations as an auxiliary language construct in lambda calculi with references [14, Chapter 13].

$$\frac{}{\Gamma \vdash \mathsf{Unit} : \mathsf{One}} \qquad \frac{\Gamma[x] = \mathsf{Some}\ \tau}{\Gamma \vdash \mathsf{Var}\ x : \tau} \qquad \frac{\mathsf{atom}\ x \sharp \Gamma \qquad \Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \mathsf{Lam}\ x\ e : \mathsf{Fun}\ \tau_1\ \tau_2}$$

$$\frac{\mathsf{atom}\ x \sharp (\Gamma, e_1) \qquad \Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{Let}\ e_1\ x\ e_2 : \tau_2} \qquad \frac{\Gamma \vdash e : \mathsf{Fun}\ \tau_1\ \tau_2 \qquad \Gamma \vdash e' : \tau_1}{\Gamma \vdash \mathsf{App}\ e\ e' : \tau_2}$$

$$\frac{\mathsf{atom}\ x \sharp \Gamma \qquad \mathsf{atom}\ y \sharp (\Gamma, x) \qquad \Gamma[x \mapsto \mathsf{Fun}\ \tau_1\ \tau_2] \vdash \mathsf{Lam}\ y\ e : \mathsf{Fun}\ \tau_1\ \tau_2}{\Gamma \vdash \mathsf{Rec}\ x\ (\mathsf{Lam}\ y\ e) : \mathsf{Fun}\ \tau_1\ \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{Inj1}\ e : \mathsf{Sum}\ \tau_1\ \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{Inj2}\ e : \mathsf{Sum}\ \tau_1\ \tau_2} \qquad \frac{\Gamma \vdash e : \mathsf{Prod}\ \tau_1\ \tau_2}{\Gamma \vdash \mathsf{Prj1}\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \mathsf{Prod}\ \tau_1\ \tau_2}{\Gamma \vdash \mathsf{Prj2}\ e : \tau_2}$$

$$\frac{\Gamma \vdash e : \mathsf{Sum}\ \tau_1\ \tau_2 \qquad \Gamma \vdash e_1 : \mathsf{Fun}\ \tau_1\ \tau \qquad \Gamma \vdash e_2 : \mathsf{Fun}\ \tau_2\ \tau}{\Gamma \vdash \mathsf{Case}\ e\ e_1\ e_2 : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{Pair}\ e_1\ e_2 : \mathsf{Prod}\ \tau_1\ \tau_2}$$

$$\frac{\mathsf{atom}\ \alpha \sharp \Gamma \qquad \Gamma \vdash e : \tau[\mathsf{Mu}\ \alpha\ \tau/\alpha]}{\Gamma \vdash \mathsf{Roll}\ e : \mathsf{Mu}\ \alpha\ \tau} \qquad \frac{\mathsf{atom}\ \alpha \sharp \Gamma \qquad \Gamma \vdash e : \mathsf{Mu}\ \alpha\ \tau}{\Gamma \vdash \mathsf{Unroll}\ e : \tau[\mathsf{Mu}\ \alpha\ \tau/\alpha]}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{Auth}\ e : \mathsf{AuthT}\ \tau} \qquad \frac{\Gamma \vdash e : \mathsf{AuthT}\ \tau}{\Gamma \vdash \mathsf{Unauth}\ e : \tau}$$

Figure 4 The typing judgment.

$$\frac{\Gamma \vdash_W e : \tau}{\Gamma \vdash_W \mathsf{Auth}\ e : \tau} \qquad \frac{\Gamma \vdash_W e : \tau}{\Gamma \vdash_W \mathsf{Unauth}\ e : \tau}$$

Figure 5 Alternative, weaker typing rules.

3.3 Operational Small-Step Semantics

Figure 6 defines the small-step semantics as the inductive predicate $\langle \pi_1, e_1 \rangle\ m{\rightarrow}\ \langle \pi_2, e_2 \rangle$, meaning "the expression e_1 in combination with the *proof stream* π_1 can take a step in *mode* m to yield the expression e_2 and the proof stream π_2." A proof stream is simply a list of $\lambda\bullet$-expressions; the infix operator @ appends lists. The mode, which is a parameter of the semantics, can be one of three values:

datatype *mode* = I | P | V

The three modes I, P, and V are read as *ideal*, *prover*, and *verifier*, respectively. The ideal mode represents the unauthenticated evaluation. The authenticated evaluation proceeds with the prover mode running on the server, while the verifier mode runs on the client. Most rules are those of a standard lambda-calculus; they are shared for all three modes. Only the last six rules of $\langle \pi_1, e_1 \rangle\ m{\rightarrow}\ \langle \pi_2, e_2 \rangle$ for Auth and Unauth depend on the mode.

In the ideal mode, Auth and Unauth are simply removed, i.e., semantically they are identity functions. Upon encountering Auth v, both the prover and the verifier compute the hash of v's shallow projection. The prover uses the hash to generate the hash-value-pair Hashed (hash $(\!|v|\!)$) v, wheras the verifier generates just the hash Hash (hash v). The rules thus enforce that the Hashed constructor only ever arises in the prover mode and the Hash constructor only in the verifier mode. Thus, the shallow projection can be omitted for the verifier. The Unauth rules are the most interesting ones, as they establish the communication of the prover and the verifier via the proof stream. Unauth can only ever be applied to

$$\frac{\langle \pi,\, e_1\rangle\ m{\to}\ \langle \pi',\, e_1'\rangle}{\langle \pi,\, \mathsf{App}\ e_1\ e_2\rangle\ m{\to}\ \langle \pi',\, \mathsf{App}\ e_1'\ e_2\rangle}$$

$$\frac{\mathsf{value}\ v_1 \qquad \langle \pi,\, e_2\rangle\ m{\to}\ \langle \pi',\, e_2'\rangle}{\langle \pi,\, \mathsf{App}\ v_1\ e_2\rangle\ m{\to}\ \langle \pi',\, \mathsf{App}\ v_1\ e_2'\rangle}$$

$$\frac{\mathsf{value}\ v \qquad \mathsf{atom}\ x \sharp (v, \pi)}{\langle \pi,\, \mathsf{App}\ (\mathsf{Lam}\ x\ e)\ v\rangle\ m{\to}\ \langle \pi,\, e[v/x]\rangle}$$

$$\frac{\mathsf{value}\ v \qquad \mathsf{atom}\ x \sharp (v, \pi) \qquad e' = e[\mathsf{Rec}\ x\ e/x]}{\langle \pi,\, \mathsf{App}\ (\mathsf{Rec}\ x\ e)\ v\rangle\ m{\to}\ \langle \pi,\, \mathsf{App}\ e'\ v\rangle}$$

$$\frac{\mathsf{value}\ v \qquad \mathsf{atom}\ x \sharp (v, \pi)}{\langle \pi,\, \mathsf{Let}\ v\ x\ e\rangle\ m{\to}\ \langle \pi,\, e[v/x]\rangle}$$

$$\frac{\mathsf{atom}\ x \sharp (e_1, e_1', \pi, \pi') \qquad \langle \pi,\, e_1\rangle\ m{\to}\ \langle \pi',\, e_1'\rangle}{\langle \pi,\, \mathsf{Let}\ e_1\ x\ e_2\rangle\ m{\to}\ \langle \pi',\, \mathsf{Let}\ e_1'\ x\ e_2\rangle}$$

$$\frac{\langle \pi,\, e_1\rangle\ m{\to}\ \langle \pi',\, e_1'\rangle}{\langle \pi,\, \mathsf{Pair}\ e_1\ e_2\rangle\ m{\to}\ \langle \pi',\, \mathsf{Pair}\ e_1'\ e_2\rangle}$$

$$\frac{\mathsf{value}\ v_1 \qquad \langle \pi,\, e_2\rangle\ m{\to}\ \langle \pi',\, e_2'\rangle}{\langle \pi,\, \mathsf{Pair}\ v_1\ e_2\rangle\ m{\to}\ \langle \pi',\, \mathsf{Pair}\ v_1\ e_2'\rangle}$$

$$\frac{\langle \pi,\, e\rangle\ m{\to}\ \langle \pi',\, e'\rangle}{\langle \pi,\, \mathsf{Prj1}\ e\rangle\ m{\to}\ \langle \pi',\, \mathsf{Prj1}\ e'\rangle}$$

$$\frac{\langle \pi,\, e\rangle\ m{\to}\ \langle \pi',\, e'\rangle}{\langle \pi,\, \mathsf{Prj2}\ e\rangle\ m{\to}\ \langle \pi',\, \mathsf{Prj2}\ e'\rangle}$$

$$\frac{\mathsf{value}\ v_1 \qquad \mathsf{value}\ v_2}{\langle \pi,\, \mathsf{Prj1}\ (\mathsf{Pair}\ v_1\ v_2)\rangle\ m{\to}\ \langle \pi,\, v_1\rangle}$$

$$\frac{\mathsf{value}\ v_1 \qquad \mathsf{value}\ v_2}{\langle \pi,\, \mathsf{Prj2}\ (\mathsf{Pair}\ v_1\ v_2)\rangle\ m{\to}\ \langle \pi,\, v_2\rangle}$$

$$\frac{\langle \pi,\, e\rangle\ m{\to}\ \langle \pi',\, e'\rangle}{\langle \pi,\, \mathsf{Inj1}\ e\rangle\ m{\to}\ \langle \pi',\, \mathsf{Inj1}\ e'\rangle}$$

$$\frac{\langle \pi,\, e\rangle\ m{\to}\ \langle \pi',\, e'\rangle}{\langle \pi,\, \mathsf{Inj2}\ e\rangle\ m{\to}\ \langle \pi',\, \mathsf{Inj2}\ e'\rangle}$$

$$\frac{\mathsf{value}\ v}{\langle \pi,\, \mathsf{Case}\ (\mathsf{Inj1}\ v)\ e_1\ e_2\rangle\ m{\to}\ \langle \pi,\, \mathsf{App}\ e_1\ v\rangle}$$

$$\frac{\mathsf{value}\ v}{\langle \pi,\, \mathsf{Case}\ (\mathsf{Inj2}\ v)\ e_1\ e_2\rangle\ m{\to}\ \langle \pi,\, \mathsf{App}\ e_2\ v\rangle}$$

$$\frac{\mathsf{value}\ v}{\langle \pi,\, \mathsf{Unroll}\ (\mathsf{Roll}\ v)\rangle\ m{\to}\ \langle \pi,\, v\rangle}$$

$$\frac{\langle \pi,\, e\rangle\ m{\to}\ \langle \pi',\, e'\rangle}{\langle \pi,\, \mathsf{Case}\ e\ e_1\ e_2\rangle\ m{\to}\ \langle \pi',\, \mathsf{Case}\ e'\ e_1\ e_2\rangle}$$

$$\frac{\langle \pi,\, e\rangle\ m{\to}\ \langle \pi',\, e'\rangle}{\langle \pi,\, \mathsf{Unroll}\ e\rangle\ m{\to}\ \langle \pi',\, \mathsf{Unroll}\ e'\rangle}$$

$$\frac{\langle \pi,\, e\rangle\ m{\to}\ \langle \pi',\, e'\rangle}{\langle \pi,\, \mathsf{Roll}\ e\rangle\ m{\to}\ \langle \pi',\, \mathsf{Roll}\ e'\rangle}$$

$$\frac{\langle \pi,\, e\rangle\ m{\to}\ \langle \pi',\, e'\rangle}{\langle \pi,\, \mathsf{Auth}\ e\rangle\ m{\to}\ \langle \pi',\, \mathsf{Auth}\ e'\rangle}$$

$$\frac{\langle \pi,\, e\rangle\ m{\to}\ \langle \pi',\, e'\rangle}{\langle \pi,\, \mathsf{Unauth}\ e\rangle\ m{\to}\ \langle \pi',\, \mathsf{Unauth}\ e'\rangle}$$

$$\frac{\mathsf{value}\ v}{\langle \pi,\, \mathsf{Auth}\ v\rangle\ \mathsf{I}{\to}\ \langle \pi,\, v\rangle}$$

$$\frac{\mathsf{value}\ v}{\langle \pi,\, \mathsf{Unauth}\ v\rangle\ \mathsf{I}{\to}\ \langle \pi,\, v\rangle}$$

$$\frac{\mathsf{closed}\ (\!|v|\!) \qquad \mathsf{value}\ v}{\langle \pi,\, \mathsf{Auth}\ v\rangle\ \mathsf{P}{\to}\ \langle \pi,\, \mathsf{Hashed}\ (\mathsf{hash}\ (\!|v|\!))\ v\rangle}$$

$$\frac{\mathsf{value}\ v}{\langle \pi,\, \mathsf{Unauth}\ (\mathsf{Hashed}\ h\ v)\rangle\ \mathsf{P}{\to}\ \langle \pi\ @\ [(\!|v|\!)],\, v\rangle}$$

$$\frac{\mathsf{closed}\ v \qquad \mathsf{value}\ v}{\langle \pi,\, \mathsf{Auth}\ v\rangle\ \mathsf{V}{\to}\ \langle \pi,\, \mathsf{Hash}\ (\mathsf{hash}\ v)\rangle}$$

$$\frac{\mathsf{closed}\ s_0 \qquad \mathsf{hash}\ s_0 = h}{\langle s_0 \# \pi,\, \mathsf{Unauth}\ (\mathsf{Hash}\ h)\rangle\ \mathsf{V}{\to}\ \langle \pi,\, s_0\rangle}$$

$$\frac{}{\langle \pi,\, e\rangle\ m{\to_0}\ \langle \pi,\, e\rangle}$$

$$\frac{\langle \pi_1,\, e_1\rangle\ m{\to_i}\ \langle \pi_2,\, e_2\rangle \qquad \langle \pi_2,\, e_2\rangle\ m{\to}\ \langle \pi_3,\, e_3\rangle}{\langle \pi_1,\, e_1\rangle\ m{\to_{i+1}}\ \langle \pi_3,\, e_3\rangle}$$

Figure 6 The small-step semantics of $\lambda\bullet$.

expressions of type AuthT. Values of this type are always Hashed h v' and Hash h (for some h, v') in the prover and verifier modes, respectively. The prover appends the shallow projection of v' to the proof stream and continues to evaluate v'. The shallow projection ensures that any hash-value pairs within v' discard the value, keeping just the hash. The verifier consumes the first element of its input proof stream to verify that this value's hash is equal to the hash of its argument. Only if the check succeeds, the evaluation may proceed.

The rules demonstrate that the evaluation in all three modes is structurally identical but a compiler would have to substitute a different function for the Auth and Unauth functions for the prover and verifier modes. In this semantics any given expression can first be executed in mode P by the prover, generating a proof stream, and then in mode V by the verifier, consuming a proof stream. The execution in mode I does not modify or depend on the proof stream at all. The last two rules lift the single-step evaluation to multiple steps, while at the same time counting the number of taken steps.

The three Auth and Unauth rules that require hash computation all have a premise that ensures that hashes are only computed on closed terms. The small-step semantics given in *ADSG* is not restricted in this way. But the restriction is unproblematic: even though our semantics allows the prover and the verifier to evaluate strictly fewer expressions, we will show later that they can still simulate any ideal computation that starts with a closed formula.

Above, we have stated informally that the prover generates the proof stream and the verifier consumes the proof stream. We can formalize this notion in the following two lemmas that will be necessary for the correctness and security proofs.

▶ **Lemma 2** (Execution in mode P generates the proof stream)**.**

$$\langle \pi_1,\ e_P\rangle\ \mathsf{P}{\rightarrow_i}\ \langle \pi_2,\ e'_P\rangle \longrightarrow \exists \pi.\ \pi_2 = \pi_1 \mathbin{@} \pi$$

▶ **Lemma 3** (Execution in mode V consumes the proof stream)**.**

$$\langle \pi_1,\ e_V\rangle\ \mathsf{V}{\rightarrow_i}\ \langle \pi_2,\ e'_V\rangle \longrightarrow \exists \pi.\ \pi_1 = \pi \mathbin{@} \pi_2$$

Furthermore, we can show that in mode P we are allowed to add (or remove) a prefix to (from) the proof stream.

▶ **Lemma 4** (Add/remove prefix of prover proof stream)**.**

$$\langle \pi,\ e_P\rangle\ \mathsf{P}{\rightarrow_i}\ \langle \pi',\ e'_P\rangle \longleftrightarrow \langle X \mathbin{@} \pi,\ e_P\rangle\ \mathsf{P}{\rightarrow_i}\ \langle X \mathbin{@} \pi',\ e'_P\rangle$$

In mode V we can modify the proof stream by adding or removing a suffix.

▶ **Lemma 5** (Add/remove suffix of verifier proof stream)**.**

$$\langle \pi,\ e_V\rangle\ \mathsf{V}{\rightarrow_i}\ \langle \pi',\ e'_V\rangle \longleftrightarrow \langle \pi \mathbin{@} X,\ e_V\rangle\ \mathsf{V}{\rightarrow_i}\ \langle \pi' \mathbin{@} X,\ e'_V\rangle$$

In mode I we do not touch the proof stream at all, so we will not need to prepend, append or remove data from them during proofs. However, we do want to prove that the proof stream does not change during evaluation.

▶ **Lemma 6** (Ideal execution does not modify the proof stream)**.**

$$\langle \pi,\ e\rangle\ \mathsf{I}{\rightarrow_i}\ \langle \pi',\ e'\rangle \longrightarrow \pi = \pi'$$

3.4 Freshness Lemmas

In Section 2, we emphasized the importance of freshness when working with Nominal. In many instances, we have to show in our proofs that a certain variable is fresh with respect to some term, proof stream, or type environment. In this section, we discuss some of the more interesting freshness lemmas we needed to prove. One of the most useful lemmas is the following, relating freshness in a typing environment with freshness in terms. We show the lemma for the weak typing judgment, but similar statements hold for the strong typing judgment and for agreement, which will be introduced in Section 4.

▶ **Lemma 7** (Freshness in environment implies freshness in terms)**.**

$$\mathsf{atom}\ x \sharp \Gamma \wedge \Gamma \vdash_W e : \tau \longrightarrow \mathsf{atom}\ x \sharp e$$

Proof. The proof is by induction on $\Gamma \vdash_W e : \tau$, with the only interesting case being the one for $\mathsf{Var}\ x$. Since $\mathsf{Var}\ x$ can only be well-typed if the type environment assigns a type to x, it is easy to show that a being fresh in Γ implies $a \neq x$. Hence, $\mathsf{atom}\ a \sharp \mathsf{Var}\ x$. ◀

For the small-step semantics we have lemmas showing that evaluation preserves freshness in some object, for example in the term when evaluating in mode P.

▶ **Lemma 8** (Prover evaluation preserves freshness in terms)**.**

$$\mathsf{atom}\ x \sharp e \wedge \langle \pi, e \rangle\ \mathsf{P}{\rightarrow}\ \langle \pi', e' \rangle \longrightarrow \mathsf{atom}\ x \sharp e'$$

For the proof stream this only holds if the atom is fresh in both the term and the proof stream.

▶ **Lemma 9** (Prover evaluation preserves freshness in proof streams)**.**

$$\mathsf{atom}\ x \sharp e \wedge \mathsf{atom}\ x \sharp \pi \wedge \langle \pi, e \rangle\ \mathsf{P}{\rightarrow}\ \langle \pi', e' \rangle \longrightarrow \mathsf{atom}\ x \sharp \pi'$$

3.5 Type Soundness

Now that we have defined the typing judgment and the small-step semantics of $\lambda\bullet$, we turn our attention to type soundness for the execution in mode I. We proceed by proving the standard progress and preservation lemmas.

▶ **Lemma 10** (Progress)**.**

$$\varnothing \vdash_W e : \tau \longrightarrow \mathsf{value}\ e \vee (\exists e'. \langle [], e \rangle\ \mathsf{I}{\rightarrow}\ \langle [], e' \rangle)$$

▶ **Lemma 11** (Preservation)**.**

$$\langle [], e \rangle\ \mathsf{I}{\rightarrow}\ \langle [], e' \rangle \wedge \varnothing \vdash_W e : \tau \longrightarrow \varnothing \vdash_W e' : \tau$$

Using Lemma 10 and Lemma 11, type soundness for weakly well-typed terms follows easily.

▶ **Lemma 12** (Type Soundness)**.**

$$\varnothing \vdash_W e : \tau \longrightarrow \mathsf{value}\ e \vee (\exists e'.\ \langle [], e \rangle\ \mathsf{I}{\rightarrow}\ \langle [], e' \rangle \wedge \varnothing \vdash_W e' : \tau)$$

```
nominal_function erase :: ty ⇒ ty where
  erase One           = One
| erase (Fun τ1 τ2)   = Fun (erase τ1) (erase τ2)
| erase (Sum τ1 τ2)   = Sum (erase τ1) (erase τ2)
| erase (Prod τ1 τ2)  = Prod (erase τ1) (erase τ2)
| erase (Mu α τ)      = Mu α (erase τ)
| erase (Alpha α)     = Alpha α
| erase (AuthT τ)     = erase τ
```

Figure 7 The erase function.

There are two differences in our Lemma 12 compared to *ADSG*'s type soundness statement (Lemma 1). First, *ADSG* formulates the lemma for an arbitrary environment Γ (and consequently for terms that may contain free variables) in the judgment – an oversight which trivially invalidates the lemma: for example, Prj1 (Var x) is not a value and cannot take a step.

The second difference is that we formulate type soundness using the weak typing judgment. Type soundness does not hold for the original set of typing rules. Consider, for example, the well-typed expression Auth Unit of type AuthT One. Since it is not a value it must take a step. However, the resulting expression Unit has the different type One, violating type soundness (namely the preservation property). *ADSG* notes that "for mode I, authenticated values of type $\bullet\tau$ [i.e., AuthT τ] are merely values of type τ." This remark seems to imply that $\forall\tau$. AuthT $\tau \equiv \tau$, a property that is essential to a successful type soundness proof. Our weak typing judgment simulates syntactic equality of authenticated types by simply omitting them and allowing the introduction of the Auth and Unauth constructors without a change of types. However, although this interpretation is necessary for type soundness, it is undesirable. The main purpose of authenticated types is to ensure that Unauth can only be applied to expressions to which Auth has been applied previously. This disallows terms such as Unauth Unit, whose semantics is well-defined in the ideal execution mode but not in the prover and verifier modes. In the weakened typing judgment such terms are considered well-typed.

Since type soundness does not hold for the strong typing judgment, we show the weaker property that well-typed terms are also weakly well-typed after removing any AuthT annotations from its type and type environment. For this purpose we define the function erase (Figure 7), which erases all AuthT annotations in a type but leaves it otherwise unchanged. Using erase we can state and prove the relationship between the weak and the strong typing judgment. The function fmmap :: $(\beta \Rightarrow \gamma) \Rightarrow (\alpha, \beta)$ *fmap* $\Rightarrow (\alpha, \gamma)$ *fmap* is the canonical map function for the type of finite maps.

▶ **Lemma 13** (Well-typedness implies weak well-typedness)**.**

$$\Gamma \vdash e : \tau \longrightarrow \text{fmmap erase } \Gamma \vdash_W e : \text{erase } \tau$$

4 Agreement

When introducing the small-step semantics we have discussed the intended interpretation of the mode. Any expression can be evaluated in mode I, performing a simple unauthenticated computation; in mode P, performing the computation and generating the proof stream; or in mode V, performing the computation and verifying the proof stream. Even though the three

$$\frac{}{\Gamma \vdash \mathsf{Unit},\ \mathsf{Unit},\ \mathsf{Unit} : \mathsf{One}} \qquad \frac{\mathsf{atom}\ x \sharp \Gamma \qquad \Gamma[x \mapsto \tau_1] \vdash e,\ e_P,\ e_V : \tau_2}{\Gamma \vdash \mathsf{Lam}\ x\ e,\ \mathsf{Lam}\ x\ e_P,\ \mathsf{Lam}\ x\ e_V : \mathsf{Fun}\ \tau_1\ \tau_2}$$

$$\frac{\Gamma[x] = \mathsf{Some}\ \tau}{\Gamma \vdash \mathsf{Var}\ x,\ \mathsf{Var}\ x,\ \mathsf{Var}\ x : \tau} \qquad \frac{\Gamma \vdash e_1,\ e_{P1},\ e_{V1} : \mathsf{Fun}\ \tau_1\ \tau_2 \qquad \Gamma \vdash e_2,\ e_{P2},\ e_{V2} : \tau_1}{\Gamma \vdash \mathsf{App}\ e_1\ e_2,\ \mathsf{App}\ e_{P1}\ e_{P2},\ \mathsf{App}\ e_{V1}\ e_{V2} : \tau_2}$$

$$\frac{\mathsf{atom}\ x \sharp (\Gamma, e_1, e_{P1}, e_{V1}) \qquad \Gamma \vdash e_1,\ e_{P1},\ e_{V1} : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash e_2,\ e_{P2},\ e_{V2} : \tau_2}{\Gamma \vdash \mathsf{Let}\ e_1\ x\ e_2,\ \mathsf{Let}\ e_{P1}\ x\ e_{P2},\ \mathsf{Let}\ e_{V1}\ x\ e_{V2} : \tau_2}$$

$$\frac{\mathsf{atom}\ x \sharp \Gamma \qquad \mathsf{atom}\ y \sharp (\Gamma, x) \qquad \Gamma[x \mapsto \mathsf{Fun}\ \tau_1\ \tau_2] \vdash \mathsf{Lam}\ y\ e,\ \mathsf{Lam}\ y\ e_P,\ \mathsf{Lam}\ y\ e_V : \mathsf{Fun}\ \tau_1\ \tau_2}{\Gamma \vdash \mathsf{Rec}\ x\ (\mathsf{Lam}\ y\ e),\ \mathsf{Rec}\ x\ (\mathsf{Lam}\ y\ e_P),\ \mathsf{Rec}\ x\ (\mathsf{Lam}\ y\ e_V) : \mathsf{Fun}\ \tau_1\ \tau_2}$$

$$\frac{\Gamma \vdash e,\ e_P,\ e_V : \tau_1}{\Gamma \vdash \mathsf{Inj1}\ e,\ \mathsf{Inj1}\ e_P,\ \mathsf{Inj1}\ e_V : \mathsf{Sum}\ \tau_1\ \tau_2} \qquad \frac{\Gamma \vdash e,\ e_P,\ e_V : \tau_1}{\Gamma \vdash \mathsf{Inj2}\ e,\ \mathsf{Inj2}\ e_P,\ \mathsf{Inj2}\ e_V : \mathsf{Sum}\ \tau_1\ \tau_2}$$

$$\frac{\Gamma \vdash e,\ e_P,\ e_V : \mathsf{Sum}\ \tau_1\ \tau_2 \qquad \Gamma \vdash e_1,\ e_{P1},\ e_{V1} : \mathsf{Fun}\ \tau_1\ \tau \qquad \Gamma \vdash e_2,\ e_{P2},\ e_{V2} : \mathsf{Fun}\ \tau_2\ \tau}{\Gamma \vdash \mathsf{Case}\ e\ e_1\ e_2,\ \mathsf{Case}\ e_P\ e_{P1}\ e_{P2},\ \mathsf{Case}\ e_V\ e_{V1}\ e_{V2} : \tau}$$

$$\frac{\Gamma \vdash e_1,\ e_{P1},\ e_{V1} : \tau_1 \qquad \Gamma \vdash e_2,\ e_{P2},\ e_{V2} : \tau_2}{\Gamma \vdash \mathsf{Pair}\ e_1\ e_2,\ \mathsf{Pair}\ e_{P1}\ e_{P2},\ \mathsf{Pair}\ e_{V1}\ e_{V2} : \mathsf{Prod}\ \tau_1\ \tau_2}$$

$$\frac{\Gamma \vdash e,\ e_P,\ e_V : \mathsf{Prod}\ \tau_1\ \tau_2}{\Gamma \vdash \mathsf{Prj1}\ e,\ \mathsf{Prj1}\ e_P,\ \mathsf{Prj1}\ e_V : \tau_1} \qquad \frac{\Gamma \vdash e,\ e_P,\ e_V : \mathsf{Prod}\ \tau_1\ \tau_2}{\Gamma \vdash \mathsf{Prj2}\ e,\ \mathsf{Prj2}\ e_P,\ \mathsf{Prj2}\ e_V : \tau_2}$$

$$\frac{\mathsf{atom}\ \alpha \sharp \Gamma \qquad \Gamma \vdash e,\ e_P,\ e_V : \tau[\mathsf{Mu}\ \alpha\ \tau/\alpha]}{\Gamma \vdash \mathsf{Roll}\ e,\ \mathsf{Roll}\ e_P,\ \mathsf{Roll}\ e_V : \mathsf{Mu}\ \alpha\ \tau} \qquad \frac{\mathsf{atom}\ \alpha \sharp \Gamma \qquad \Gamma \vdash e,\ e_P,\ e_V : \mathsf{Mu}\ \alpha\ \tau}{\Gamma \vdash \mathsf{Unroll}\ e,\ \mathsf{Unroll}\ e_P,\ \mathsf{Unroll}\ e_V : \tau[\mathsf{Mu}\ \alpha\ \tau/\alpha]}$$

$$\frac{\Gamma \vdash e,\ e_P,\ e_V : \tau}{\Gamma \vdash \mathsf{Auth}\ e,\ \mathsf{Auth}\ e_P,\ \mathsf{Auth}\ e_V : \mathsf{AuthT}\ \tau} \qquad \frac{\Gamma \vdash e,\ e_P,\ e_V : \mathsf{AuthT}\ \tau}{\Gamma \vdash \mathsf{Unauth}\ e,\ \mathsf{Unauth}\ e_P,\ \mathsf{Unauth}\ e_V : \tau}$$

$$\frac{\mathsf{value}\ v \qquad \mathsf{value}\ v_P \qquad \varnothing \vdash v,\ v_P,\ (\!|v_P|\!) : \tau \qquad \mathsf{hash}\ (\!|v_P|\!) = h}{\Gamma \vdash v,\ \mathsf{Hashed}\ h\ v_P,\ \mathsf{Hash}\ h : \mathsf{AuthT}\ \tau}$$

Figure 8 The agreement predicate.

modes differ in their semantics and their terms may differ at any point during evaluation, their evaluations are structurally identical. This observation is captured by the agreement relation, written as $\Gamma \vdash e,\ e_P,\ e_V : \tau$ and read as "in environment Γ, ideal expression e, prover expression e_P, and verifier expression e_V all agree at type τ" (quoted from *ADSG* [10]).

We formalize agreement as an inductive predicate, with the introduction rules presented in Figure 8. Most rules are straightforward extensions of the (strong) typing judgment to three terms. This immediately gives us the following result, which states that any well-typed expression can be used in the ideal, prover, and verifier positions to yield an agreeing triple.

▶ **Lemma 14** (Well-typedness implies agreement)**.**

$$\Gamma \vdash e : \tau \longrightarrow \Gamma \vdash e,\ e,\ e : \tau$$

The interesting exception to the agreement rules being extensions of the typing rules is the last rule. It is modeled after the Auth small-step rules for the three modes. This rule allows the three expressions to diverge during the evaluation of Auth and still be in agreement. Note that the agreeing triple in the rule's premises may not contain any free variables. This property is enforced by the empty type environment, using the agreement version of Lemma 7. Therefore, the use of the hash function in this rule is unproblematic.

Lemma 14 states that well-typedness implies agreement. Ideally, we would also like to show the other direction of this property: agreement implying well-typedness. Unfortunately this does not hold. This is due to the extra agreement rule, allowing the introduction of authenticated types for any ideal value. Consider for example that with $\varnothing \vdash \mathsf{Unit}, \mathsf{Unit}, \mathsf{Unit} : \mathsf{One}$, we can obtain $\varnothing \vdash \mathsf{Unit}, \mathsf{Hashed}\ h\ \mathsf{Unit}, \mathsf{Hash}\ h : \mathsf{AuthT}\ \mathsf{One}$. Clearly we cannot show $\varnothing \vdash \mathsf{Unit} : \mathsf{AuthT}\ \mathsf{One}$. However, we can show weak well-typedness:

▶ **Lemma 15** (Reformulated Lemma 2.3 from *ADSG*).

$$\Gamma \vdash e,\ e_P,\ e_V : \tau \longrightarrow \mathsf{fmmap\ erase}\ \Gamma \vdash_W e : \mathsf{erase}\ \tau$$

We now prove Lemma 16 and Lemma 17 that are used extensively in later proofs.

▶ **Lemma 16** (Lemma 2.1 from *ADSG*).

$$\Gamma \vdash e,\ e_P,\ e_V : \tau \longrightarrow (\!|e_P|\!) = e_V$$

▶ **Lemma 17** (Lemma 2.4 from *ADSG*).

$$\Gamma \vdash e,\ e_P,\ e_V : \tau \longrightarrow (\mathsf{value}\ e \wedge \mathsf{value}\ e_P \wedge \mathsf{value}\ e_V) \vee (\neg\mathsf{value}\ e \wedge \neg\mathsf{value}\ e_P \wedge \neg\mathsf{value}\ e_V)$$

In addition to Lemmas 15, 16, and 17, *ADSG* also states the following false property as Lemma 2.2. (Although *ADSG* states the property as a lemma, we did not encounter a situation where this statement was required to complete a proof.)

$$\Gamma \vdash e,\ e_P,\ e_V : \tau \wedge \Gamma \vdash e,\ e'_P,\ e'_V : \tau \longrightarrow e_P = e'_P \wedge e_V = e'_V$$

To demonstrate why this property does not hold we construct a counterexample. We define $\mathsf{h} = \mathsf{Hash\ Unit}$ and we abbreviate Unit as u for better readability. Let us first consider the following two agreeing triples.

$\varnothing \vdash \mathsf{u}, \mathsf{u}, \mathsf{u} : \mathsf{One}$
$\varnothing \vdash \mathsf{u}, \mathsf{Hashed\ h\ u}, \mathsf{Hash\ h} : \mathsf{AuthT\ One}$

The second triple can be generated from the first one by applying the last agreement rule. Both triples share the environment and the first term but disagree in the second and third term as well as their type. Using the Pair rule we obtain the following two agreeing triples.

$\varnothing \vdash \mathsf{Pair\ u\ u}, \mathsf{Pair\ u\ u}, \mathsf{Pair\ u\ u} : \mathsf{Prod\ One\ One}$
$\varnothing \vdash \mathsf{Pair\ u\ u}, \mathsf{Pair\ u\ (Hashed\ h\ u)}, \mathsf{Pair\ u\ (Hash\ h)} : \mathsf{Prod\ One\ (AuthT\ One)}$

Applying Prj1 to these triples removes the difference in the types but preserves the differences in the second and third terms, completing our counterexample to *ADSG*'s Lemma 2.2.

$\varnothing \vdash \mathsf{Prj1\ (Pair\ u\ u)}, \mathsf{Prj1\ (Pair\ u\ u)}, \mathsf{Prj1\ (Pair\ u\ u)} : \mathsf{One}$
$\varnothing \vdash \mathsf{Prj1\ (Pair\ u\ u)}, \mathsf{Prj1\ (Pair\ u\ (Hashed\ h\ u))}, \mathsf{Prj1\ (Pair\ u\ (Hash\ h))} : \mathsf{One}$

In the following we prove that, given a well-typed $\lambda\bullet$ term, containing only free variables of authenticated types, substituting agreeing values of the same type produces an agreeing triple. This property is significant because it occurs in the following practical scenario. The verifier must represent the data structure in a query it sends to the prover. It does so by replacing it with a free variable, for which the prover substitutes its representation of the data structure. The prover then returns the generated proof stream to the verifier, who substitutes the free variable with its hash of the data structure and verifies the proof stream. We formalized this lemma as stated below, with *fmdom* returning a finite map's domain as a finite set and $|\in|$ denoting membership on finite sets.

▶ **Lemma 18** (Reformulated Lemma 3 from *ADSG*). *For* Δ, Δ_P, Δ_V :: $(var, term)$ *fmap:*

$$\left(\begin{array}{l}\Gamma \vdash e : \tau \wedge \\ \mathsf{fmdom}\ \Delta = \mathsf{fmdom}\ \Gamma \wedge \mathsf{fmdom}\ \Delta_P = \mathsf{fmdom}\ \Gamma \wedge \mathsf{fmdom}\ \Delta_V = \mathsf{fmdom}\ \Gamma \wedge \\ \left(\begin{array}{l}\forall x.\ x \mid\in\mid \mathsf{fmdom}\ \Gamma \longrightarrow (\exists \tau', v, v_P, h.\ \Gamma[x] = \mathsf{Some}\ (\mathsf{AuthT}\ \tau') \wedge \\ \quad \Delta[x] = \mathsf{Some}\ v \wedge \Delta_P[x] = \mathsf{Some}\ (\mathsf{Hashed}\ h\ v_P) \wedge \Delta_V[x] = \mathsf{Some}\ (\mathsf{Hash}\ h) \wedge \\ \quad \varnothing \vdash v, \mathsf{Hashed}\ h\ v_P, \mathsf{Hash}\ h : \mathsf{AuthT}\ \tau')\end{array}\right)\end{array}\right) \longrightarrow$$
$$\varnothing \vdash \mathsf{psubst}\ e\ \Delta, \mathsf{psubst}\ e\ \Delta_P, \mathsf{psubst}\ e\ \Delta_V : \tau$$

ADSG's Lemma 3 includes an additional premise:

> $\Gamma \vdash e : \tau$ where e contains no values of type $\mathsf{AuthT}\ \tau$

Since variables are values, this premise implies that e contains neither bound nor free variables of type $\mathsf{AuthT}\ \tau$ (only for this particular τ, it can contain other variables with other authenticated types). The premise does not impose any further restrictions, since variables are the only expressions that are values and can have type $\mathsf{AuthT}\ \sigma$ for some σ. We are unclear as to what this premise's purpose is. Fortunately, the lemma holds without it.

Finally, we prove a straightforward but crucial lemma, which states that substituting agreeing values of the correct type for a free variable in an agreeing triple preserves agreement.

▶ **Lemma 19** (Lemma 4 from *ADSG*).

$$\left(\begin{array}{l}\Gamma[x \mapsto \tau'] \vdash e,\ e_P,\ e_V : \tau \wedge \varnothing \vdash v,\ v_P,\ v_V : \tau' \wedge \\ \mathsf{value}\ v \wedge \mathsf{value}\ v_P \wedge \mathsf{value}\ v_V\end{array}\right) \longrightarrow \Gamma \vdash e[v/x],\ e_P[v_P/x],\ e_V[v_V/x] : \tau$$

5 Correctness

Having formalized $\lambda\bullet$ and proved a number of lemmas about it, we now take a look at the main claims formulated in *ADSG*, concerning the correctness and security of $\lambda\bullet$. We start with some agreeing terms e, e_P, e_V. The properties we would then like to obtain can be informally stated as follows:

1. *Correctness*: If e takes i steps in mode I, then e_P and e_V can also take i steps in their respective modes, with the verifier consuming the prover's output proof stream. The resulting terms agree.
2. *Security*: If e_V takes i steps in mode V, consuming the proof stream π (which may be legit or created by an adversary trying to trick the verifier) then either e and e_P can also take i steps in their respective modes, with the prover generating π and the resulting terms agreeing, or otherwise there exists a term in the proof stream π, such that we can show the presence of a hash collision.

Besides these primary claims *ADSG* formulates a third claim (named *Remark 1*) that starts with the prover's computation and lets the other two modes follow:

3. *Remark 1*: If e_P takes i steps in mode P generating the proof stream π, then e and e_V can also take i steps in their respective modes, with the verifier consuming π. The resulting terms agree.

In a first step we formulate and prove these three properties on the single-step relation. Afterwards we will lift these lemmas to obtain the main results on the multi-step relation.

▶ **Lemma 20** (Single step version of Correctness, Lemma 5 from *ADSG*).

$$\varnothing \vdash e,\ e_P,\ e_V : \tau \wedge \langle [],\ e\rangle\ \mathsf{I}{\rightarrow}\ \langle [],\ e'\rangle \longrightarrow \left(\begin{array}{l}\exists e'_P, e'_V, \pi.\ \varnothing \vdash e',\ e'_P,\ e'_V : \tau \wedge \\ \quad (\forall \pi_P.\ \langle \pi_P,\ e_P\rangle\ \mathsf{P}{\rightarrow}\ \langle \pi_P \mathbin{@} \pi,\ e'_P\rangle) \wedge (\forall \pi'.\ \langle \pi \mathbin{@} \pi',\ e_V\rangle\ \mathsf{V}{\rightarrow}\ \langle \pi',\ e'_V\rangle)\end{array}\right)$$

Proof. The proof is by induction on the agreement relation. Most cases are straightforward, using the lemmas about agreement and various freshness lemmas. The most interesting cases are those for Let, Auth and Unauth. Let is the only construct with a binder that allows recursive evaluation, requiring an additional freshness lemma to show that the recursive step preserves freshness. The Auth and Unauth cases require us to show that the expressions being hashed are closed. In both cases we have an agreeing triple with an empty typing context, so we can apply the counterpart of Lemma 7 for agreement to show that property. ◀

▶ **Lemma 21** (Single step version of Security, Lemma 6 in *ADSG*).

$$\varnothing \vdash e,\ e_P,\ e_V : \tau \wedge \langle \pi_A,\ e_V\rangle\ \mathsf{V}{\rightarrow}\ \langle \pi',\ e'_V\rangle \longrightarrow \left(\begin{array}{l}\exists e', e'_P, \pi.\ \langle [],\ e\rangle\ \mathsf{I}{\rightarrow}\ \langle [],\ e'\rangle \wedge (\forall \pi_P.\ \langle \pi_P,\ e_P\rangle\ \mathsf{P}{\rightarrow}\ \langle \pi_P \mathbin{@} \pi,\ e'_P\rangle) \wedge \\ \quad ((\varnothing \vdash e',\ e'_P,\ e'_V : \tau \wedge \pi_A = \pi \mathbin{@} \pi') \vee \\ \quad (\exists s, s'.\ \pi = [s] \wedge \pi_A = [s'] \mathbin{@} \pi' \wedge s \neq s' \wedge \mathsf{hash}\ s = \mathsf{hash}\ s' \wedge \mathsf{closed}\ s \wedge \mathsf{closed}\ s'))\end{array}\right)$$

Proof. The proof is similar to that of Lemma 20, though the Unauth case here does not involve hashes and therefore does not need special treatment. ◀

▶ **Lemma 22** (Single step version of Remark 1).

$$\varnothing \vdash e,\ e_P,\ e_V : \tau \wedge \langle \pi_P,\ e_P\rangle\ \mathsf{P}{\rightarrow}\ \langle \pi_P \mathbin{@} \pi,\ e'_P\rangle \longrightarrow (\exists e', e'_V.\ \varnothing \vdash e',\ e'_P,\ e'_V : \tau \wedge \langle [],\ e\rangle\ \mathsf{I}{\rightarrow}\ \langle [],\ e'\rangle \wedge \langle \pi,\ e_V\rangle\ \mathsf{V}{\rightarrow}\ \langle [],\ e'_V\rangle)$$

Proof. The proof is by straightforward induction on the agreement relation, without any of the special cases of Lemmas 20 and 21. ◀

Having proven Lemmas 20, 21 and 22 we can now lift the results to the small-step semantics' transitive closure to obtain the main results, described informally above.

▶ **Theorem 23** (Correctness, Theorem 1 in *ADSG*).

$$\varnothing \vdash e,\ e_P,\ e_V : \tau \wedge \langle [],\ e\rangle\ \mathsf{I}{\rightarrow_i}\ \langle [],\ e'\rangle \longrightarrow (\exists e'_P, e'_V, \pi.\ \varnothing \vdash e',\ e'_P,\ e'_V : \tau \wedge \langle [],\ e_P\rangle\ \mathsf{P}{\rightarrow_i}\ \langle \pi,\ e'_P\rangle \wedge \langle \pi,\ e_V\rangle\ \mathsf{V}{\rightarrow_i}\ \langle [],\ e'_V\rangle)$$

▶ **Theorem 24** (Security, Theorem 1 in *ADSG*).

$$\varnothing \vdash e,\ e_P,\ e_V : \tau \wedge \langle \pi_A,\ e_V\rangle\ \mathsf{V}{\rightarrow_i}\ \langle \pi',\ e'_V\rangle \longrightarrow \left(\begin{array}{l}(\exists e', e'_P, \pi.\ \langle [],\ e\rangle\ \mathsf{I}{\rightarrow_i}\ \langle [],\ e'\rangle \wedge \langle [],\ e_P\rangle\ \mathsf{P}{\rightarrow_i}\ \langle \pi,\ e'_P\rangle \wedge \\ \quad \pi_A = \pi \mathbin{@} \pi' \wedge \varnothing \vdash e',\ e'_P,\ e'_V : \tau) \vee \\ (\exists e'_P, j, \pi_0, \pi'_0, s, s'.\ j \leq i \wedge \langle [],\ e_P\rangle\ \mathsf{P}{\rightarrow_j}\ \langle \pi_0 \mathbin{@} [s],\ e'_P\rangle \wedge \\ \quad \pi_A = \pi_0 \mathbin{@} [s'] \mathbin{@} \pi'_0 \mathbin{@} \pi' \wedge s \neq s' \wedge \mathsf{hash}\ s = \mathsf{hash}\ s' \wedge \mathsf{closed}\ s \wedge \mathsf{closed}\ s')\end{array}\right)$$

▶ **Theorem 25** (Remark 1 in *ADSG*).

$$\varnothing \vdash e,\ e_P,\ e_V : \tau \wedge \langle \pi_P,\ e_P\rangle\ \mathsf{P}{\rightarrow_i}\ \langle \pi_P \mathbin{@} \pi,\ e'_P\rangle \longrightarrow (\exists e', e'_V.\ \varnothing \vdash e',\ e'_P,\ e'_V : \tau \wedge \langle [],\ e\rangle\ \mathsf{I}{\rightarrow_i}\ \langle [],\ e'\rangle \wedge \langle \pi,\ e_V\rangle\ \mathsf{V}{\rightarrow_i}\ \langle [],\ e'_V\rangle)$$

The statement of Theorem 24 differs from the one in *ADSG*. In the case where colliding hashes cause the verifier to falsely accept a computation as correct, the theorem ensures that the offending proof stream π_A has a specific shape. *ADSG* claims this shape to be $\pi_A = \pi_0 \mathbin{@} [s'] \mathbin{@} \pi'$, i.e., the evaluation must stop after a hash collision is encountered. For Lemma 21, the single-step version, this holds, since we only evaluate a single step. However, this fact is no longer true when taking multiple steps, since the verifier may continue to evaluate and consume valid (or invalid) elements of the proof stream after encountering the hash collision. In fact, the verifier cannot recognize that a hash collision has occurred. Formally, this means that $\pi_A = \pi_0 \mathbin{@} [s'] \mathbin{@} \pi_0' \mathbin{@} \pi'$ for some π_0', as our corrected theorem states. We illustrate the problem with *ADSG*'s formulation with a concrete counterexample:

$$\mathsf{Let}\ (\mathsf{Unauth}\ (\mathsf{Auth}\ (\mathsf{Inj1}\ \mathsf{Unit})))\ x\ (\mathsf{Let}\ (\mathsf{Unauth}\ (\mathsf{Auth}\ \mathsf{Unit}))\ y\ (\mathsf{Var}\ x))$$

This term can be evaluated in the prover mode to generate the proof stream $[\mathsf{Inj1\ Unit}, \mathsf{Unit}]$. We assume a hash function, which satisfies $\mathsf{hash}\ (\mathsf{Inj1\ Unit}) = \mathsf{hash}\ (\mathsf{Inj2\ Unit})$ and $\mathsf{hash\ Unit} \neq \mathsf{hash}\ t$ for all $t \neq \mathsf{Unit}$. Note that, since all theorems are formulated to be agnostic to the choice of the hash function, this is an entirely reasonable hash function to use in a counterexample. A verifier using the adversarial proof stream $\pi_A = [\mathsf{Inj2\ Unit}, \mathsf{Unit}]$ evaluates the given term to $\mathsf{Inj2\ Unit}$. The original statement of the theorem would require the proof stream to be of the shape $\pi_A = \pi_0 \mathbin{@} [s'] \mathbin{@} \pi'$ with $\pi' = []$. However, our adversarial proof stream does not fit this pattern since the term with a colliding hash is not the last term from the proof stream that is evaluated. With our amended, formally verified version, the shape $\pi_A = \pi_0 \mathbin{@} [s'] \mathbin{@} \pi_0' \mathbin{@} \pi'$ can be matched as $\pi_A = [] \mathbin{@} [\mathsf{Inj1\ Unit}] \mathbin{@} [\mathsf{Unit}] \mathbin{@} []$.

Since *ADSG* requires terms to be in administrative normal form, the above counterexample cannot be expressed in *ADSG*'s definition of $\lambda\bullet$. However, in our formalization we include a (more verbose) counterexample in administrative normal form.

6 Discussion

We have formalized $\lambda\bullet$ and proved its correctness and security in Isabelle/HOL. Our work can be seen as the mechanized supplement to Miller et al.'s *ADSG* [10]. Ultimately, *ADSG* passed the test of formalization. However, achieving this result turned out to be harder than we first had expected, given the mistakes and imprecisions we had to overcome. We discovered major problems in the paper's Lemmas 1 and 2.2. We repaired Lemma 1 in a rather unsatisfactory fashion. However, in our view type soundness, and more specifically type preservation, is not very relevant for $\lambda\bullet$; what is more important is the preservation of agreement, which correctness and security establish. Lemma 2.2 could not be salvaged. Moreover, we removed a redundant (and nonsensical) assumption from *ADSG*'s Lemma 3 and corrected a slip in the formal statement of *ADSG*'s main security theorem. We have not reported here the minor typos we found in *ADSG*'s informal definitions and refer to the first author's Bachelor's thesis [4] for such an overview. Taken together, our findings confirm the value of formal proofs. The formalization could (and arguably should) have been undertaken as part of the research on *ADSG*.

The last point is typically countered by the disproportional effort needed to obtain the formalization. However, in this case the effort was modest: The main difficulties stemmed from the fact that on several occasions we first tried to prove false statements from *ADSG*.

At 3500 lines of proof, our formalization is concise. In our view, Nominal was the main asset behind this conciseness, because it allowed us to closely follow the informal proofs, while discharging straightforward freshness obligations along the way. Nominal's seamless integration with the type of finite maps provided the right level of abstraction to reason about type environments and term substitutions.

However, we also noticed a few points where Nominal could provide a better user experience. First, the introduction of binding-aware recursive functions and inductive predicates requires some boilerplate proofs, which in many cases seem automatable. This impression is confirmed by the fact that we could literally copy these proofs from unrelated formalizations that were also using Nominal and perform minor adjustments to make them work in our case. Second, *ADSG* uses terms of the form rec x $\lambda y.\ t$ for defining recursive functions, which we model with the term Rec x (Lam y t). The more faithful way to model this form would be a single Nominal datatype constructor that simultaneously binds two variables:

Rec $(x :: \mathit{var})$ $(y :: \mathit{var})$ $(t :: \mathit{term})$ **binds** x **and** y **in** t

Nominal supports this declaration. However, the reasoning infrastructure it provides for such constructors is significantly more difficult to use than the one for the special case of constructors binding a single variable. We had started our formalization with the above formulation, but soon switched to the presented Rec constructor that only binds the recursive variable x. Note that both typing and agreement require Rec's second argument to be of a function type, which is what the above form used in *ADSG* aims to hardwire into the syntax. Third, unlike *ADSG* we do not consider actually running $\lambda\bullet$ programs. Here, in our opinion, Nominal does not score very well by not being integrated with Isabelle's code generator. And moreover, it is not clear in general how to execute recursive functions that carry freshness assumptions. Executability can be regained by translating the Nominal types to a nameless representation (e.g., de Bruijn indices) and lifting all definitions to this representation. Developing a more principled approach to executing Nominal programs is interesting future work.

References

1. Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 40 of *Studies in Logic*. Elsevier, 1984.
2. Jasmin Christian Blanchette, Lorenzo Gheri, Andrei Popescu, and Dmitriy Traytel. Bindings as bounded natural functors. *PACMPL*, 3(POPL):22:1–22:34, 2019. doi:10.1145/3290335.
3. Joachim Breitner. The adequacy of Launchbury's natural semantics for lazy evaluation. *J. Funct. Program.*, 28:e1, 2018. doi:10.1017/S0956796817000144.
4. Matthias Brun. *Authenticated Data Structures in Isabelle/HOL*. B.Sc. thesis, ETH Zürich, 2019.
5. Premkumar T. Devanbu, Michael Gertz, Charles U. Martel, and Stuart G. Stubblebine. Authentic third-party data publication. In Bhavani M. Thuraisingham, Reind P. van de Riet, Klaus R. Dittrich, and Zahir Tari, editors, *DBSec 2000*, volume 201 of *IFIP Conference Proceedings*, pages 101–112. Kluwer, 2000. doi:10.1007/0-306-47008-X_9.
6. Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In Robert Cartwright, editor, *PLDI 1993*, pages 237–247. ACM, 1993. doi:10.1145/155090.155113.
7. Murdoch Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Asp. Comput.*, 13(3-5):341–363, 2002. doi:10.1007/s001650200016.
8. Brian Huffman and Christian Urban. A new foundation for Nominal Isabelle. In Matt Kaufmann and Lawrence C. Paulson, editors, *ITP 2010*, volume 6172 of *LNCS*, pages 35–50. Springer, 2010. doi:10.1007/978-3-642-14052-5_5.
9. Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *CRYPTO 1987*, volume 293 of *LNCS*, pages 369–378. Springer, 1987. doi:10.1007/3-540-48184-2_32.

10 Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. In Suresh Jagannathan and Peter Sewell, editors, *POPL 2014*, pages 411–424. ACM, 2014. doi:10.1145/2535838.2535851.

11 Julian Nagele, Vincent van Oostrom, and Christian Sternagel. A short mechanized proof of the Church-Rosser theorem by the Z-property for the $\lambda\beta$-calculus in Nominal Isabelle. *CoRR*, abs/1609.03139, 2016.

12 Mizuhito Ogawa, Eiichi Horita, and Satoshi Ono. Proving properties of incremental Merkle trees. In Robert Nieuwenhuis, editor, *CADE 2005*, volume 3632 of *LNCS*, pages 424–440. Springer, 2005. doi:10.1007/11532231_31.

13 Lawrence C. Paulson. A mechanised proof of Gödel's incompleteness theorems using Nominal Isabelle. *J. Autom. Reasoning*, 55(1):1–37, 2015. doi:10.1007/s10817-015-9322-8.

14 Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

15 George Pîrlea and Ilya Sergey. Mechanising blockchain consensus. In June Andronick and Amy P. Felty, editors, *CPP 2018*, pages 78–90. ACM, 2018. doi:10.1145/3167086.

16 Roberto Tamassia. Authenticated data structures. In Giuseppe Di Battista and Uri Zwick, editors, *ESA 2003*, volume 2832 of *LNCS*, pages 2–5. Springer, 2003. doi:10.1007/978-3-540-39658-1_2.

17 Christian Urban and Cezary Kaliszyk. General bindings and alpha-equivalence in Nominal Isabelle. *Logical Methods in Computer Science*, 8(2), 2012. doi:10.2168/LMCS-8(2:14)2012.

18 Christian Urban and Julien Narboux. Formal SOS-proofs for the lambda-calculus. *Electr. Notes Theor. Comput. Sci.*, 247:139–155, 2009. doi:10.1016/j.entcs.2009.07.053.

19 Christian Urban and Christine Tasson. Nominal techniques in Isabelle/HOL. In Robert Nieuwenhuis, editor, *CADE 2005*, volume 3632 of *LNCS*, pages 38–53. Springer, 2005. doi:10.1007/11532231_4.

20 Bill White. A theory for lightweight cryptocurrency ledgers. Accessed on 30.03.2019, 2015. URL: https://github.com/input-output-hk/qeditas-ledgertheory.

A Verified and Compositional Translation of LTL to Deterministic Rabin Automata

Julian Brunner
Technische Universität München, Germany
julian.brunner@tum.de

Benedikt Seidl
Technische Universität München, Germany
benedikt.seidl@tum.de

Salomon Sickert[1]
Technische Universität München, Germany
salomon.sickert@tum.de

Abstract

We present a formalisation of the unified translation approach from linear temporal logic (LTL) to ω-automata from [19]. This approach decomposes LTL formulas into "simple" languages and allows a clear separation of concerns: first, we formalise the purely logical result yielding this decomposition; second, we develop a generic, executable, and expressive automata library providing necessary operations on automata to re-combine the "simple" languages; third, we instantiate this generic theory to obtain a construction for deterministic Rabin automata (DRA). We extract from this particular instantiation an executable tool translating LTL to DRAs. To the best of our knowledge this is the first verified translation of LTL to DRAs that is proven to be double-exponential in the worst case which asymptotically matches the known lower bound.

2012 ACM Subject Classification Theory of computation → Automata over infinite objects; Theory of computation → Modal and temporal logics; Theory of computation → Interactive proof systems

Keywords and phrases Automata Theory, Automata over Infinite Words, Deterministic Automata, Linear Temporal Logic, Model Checking, Verified Algorithms

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.11

Supplement Material The described Isabelle/HOL development is archived in the "Archive of Formal Proofs" and is split into the entries [10] and [39].

Funding This work was partially funded and supported by the German Research Foundation (DFG) project "Verified Model Checkers" (317422601).

Acknowledgements The authors want to thank Manuel Eberl, Javier Esparza, Lars Hupel, Peter Lammich, and Tobias Nipkow for their helpful comments and technical expertise.

1 Introduction

As time has shown again and again, bugs in hardware and software can have dramatic costs, ranging from monetary damages over destroyed property to life-threatening situations. In order to prevent the introduction of unwanted behaviour into software or hardware designs, an immense amount of testing and debugging is applied. However, for critical systems such methods are not enough, since they simply cannot guarantee the absence of bugs in general. Formal methods offer here a way forward by applying mathematical rigour to detect and rule out unwanted behaviour. Model checking [14] is one of the most successful techniques

[1] Corresponding author

in the area of formal methods. A key component for model checking reactive systems, i.e., non-terminating systems interacting with an open environment, against a temporal specification language, in our case linear temporal logic (LTL), is the translation to a suitable automaton model over infinite words.

Throughout the last decades, a wide variety of translation strategies to different types of ω-automata have been proposed and implemented, e.g. [23, 22, 2, 4, 43, 17]. However, as mentioned before, software development seems to be inherently error-prone, not to mention there might be mistakes in the definition of these constructions themselves. So, how can we trust these implementations to produce the correct automata for identifying bugs or proving their absence? Who watches the watchers?

Exactly that train of thought lead to the development of the Cava LTL model checker [20] which is verified in Isabelle and exported as an executable tool. The model checker includes a translation from LTL to nondeterministic Büchi automata due to [23]. However, for model checking other structures, such as probabilistic systems, other types of automata are necessary, such as limit-deterministic [44, 15] or deterministic automata [5]. Consequently, there is the need to formalise new translations from scratch which seems wasteful and cumbersome.

It would be desirable to have a separation of concerns: a theory that captures the common essence of LTL for all desired translations and that leaves a small gap to deal with the specifics of a chosen automaton model. The logical framework of [19] sketches an approach to such a modularisation: a theorem decomposing an LTL formula φ into "simple" languages, named $L^1_{\varphi,X}$, $L^2_{X,Y}$, and $L^3_{X,Y}$, such that:

$$\mathcal{L}(\varphi) = \bigcup_{\substack{X \subseteq \nu(\varphi) \\ Y \subseteq \mu(\varphi)}} (L^1_{\varphi,X} \cap L^2_{X,Y} \cap L^3_{X,Y})$$

where X and Y are sets of least- and greatest-fixed operators – hence the names ν and μ – that are subformulas of φ. We will later see a formal definition of these sets. This decomposition outlines a simple strategy to obtain a translation from LTL to our chosen automaton model: first, we define constructions for the "simple" languages; second, we implement two Boolean operations, namely union and intersection, in the automaton model; third, we combine the automata for $L^1_{\varphi,X}$, $L^2_{X,Y}$, and $L^3_{X,Y}$ using these Boolean operations.

Contribution

We provide a formalisation of [19] in Isabelle and contribute the following components: (1) a generic and expressive automata library[2] providing the necessary Boolean operations, (2) a formalisation of the Master Theorem [19] decomposing LTL formulas, (3) a combination of these two components to obtain an executable and verified translation from LTL to deterministic Rabin automata (DRA) of asymptotic optimal size, and (4) an implementation extracted from the Isabelle theory combined with an LTL parser, a verified LTL simplifier, and a serialisation to the HOA format [3], a textual format for ω-automata. Note that the resulting implementation is just one use-case and using the same framework we can also obtain a construction for other types of ω-automata, e.g. nondeterministic Büchi automata (NBA) or deterministic generalised Rabin automata. However, this would exceed the scope and space of this paper.

[2] The scope of the library is actually wider than just the support of ω-automata: automata on finite words and abstract transition systems can also be expressed.

Isabelle/HOL [36] is a proof assistant based on Higher-Order Logic (HOL), which can be thought of as a combination of functional programming and logic. Formalisations done in Isabelle are trustworthy for two reasons: First, Isabelle's LCF architecture guarantees that all proofs are checked using a very small logical core which is rarely modified but tested extensively over time. This reduces the trusted code base to a minimum. Second, bugs in the core rarely lead to accidentally proving false propositions. Bugs that have large effects are easily caught, while the limited applicability of bugs with small effects is unlikely to coincide with a logical mistake in the large-scale structure of the proof. In order to export executable code, we use the Isabelle code generator in conjunction with the monadic refinement framework [26] and automatic refinement [27]. Finally, we use several entries from the "Archive of Formal Proofs" (AFP), a collection of formalisations for Isabelle that are maintained and continuously machine-checked.

Related Work

A substantial amount of work has already been invested into verifying translations from linear temporal logic (LTL) to nondeterministic Büchi automata (NBA[3]): We already mentioned [20] which includes a translation to NBAs following the tableau construction from [23]. Further, the translation proposed by [22], which translates LTL via very-weak alternating automata to NBAs, has been formalised by [25] in HOL4. This work also includes an executable refinement of the abstract algorithm.

Alternating automata have been previously studied in [34] with an application to the translation of LTL to alternating ω-automata. However, the translation from alternating automaton to NBAs is not included. At the other end of the spectrum the publication [17], with the formal proof development archived in [40], presents a direct, verified, and executable translation from LTL to deterministic generalised Rabin automata. However, this construction is only shown to be triple-exponential and thus one exponential larger than the known, optimal lower bound. It is also important to mention that with the help of the Isabelle formalisation errors in the original publication [18] were uncovered and removed for the journal version [17]. This highlights again how important such a rigorous development for verification tools is.

Another interesting point is that the DRA constructions we provide for the "simple" languages can be seen as a version of Brzozowski's derivatives [13] applied to LTL formulas. Derivative-based constructions seem to be more natural in the functional programming paradigm as the work on regular expression equivalence from [37] shows.

Outline

After a brief introduction of the preliminaries in Section 2 we discuss the used automata formalisation in Section 3. We then give an overview of the LTL decomposition results in Section 4 and finally derive an executable LTL to DRA translation in Section 5.

[3] In the context of this paper we do not distinguish minor variations of acceptance conditions and the term NBA includes also nondeterministic generalised Büchi automata as well as transition-based Büchi automata. Similar we use the term DRA also for deterministic generalised Rabin automata.

2 Preliminaries

Locales

Isabelle provides a mechanism for parameterized theory contexts in the form of locales [6]. In a simplified sense, this means that a named context can be defined that is both parameterized by types and terms as well as augmented with assumptions. It is then possible to add various definitions and theorems within this context. Finally, by instantiating the parameters and proving the assumptions, these definitions and theorems also become instantiated and available to the enclosing context.

ω-Words

Let Σ be a finite alphabet. An ω-word w over Σ is an infinite sequence of letters $a_0 a_1 a_2 \ldots$ with $a_i \in \Sigma$ for all $i \geq 0$ and an ω-language is a set of ω-words. We use two different representations for ω-words over a type α: as a function "$\alpha\,\mathsf{word} = \mathsf{nat} \Rightarrow \alpha$" and as a codatatype "$\alpha\,\mathsf{stream} = \alpha \mathbin{\#\#} \alpha\,\mathsf{stream}$". The reason for this division is historic and is due to the fact that the material building on the *LTL* entry [41] predates the development of the codatatype package [7]. Observe that these two types are isomorphic.

The function $\mathsf{prefix}\ i\ w$ returns the finite prefix of w of length i and the function $\mathsf{suffix}\ i\ w$ gives the infinite suffix of w starting at i. The concatenation operator $w' \frown w$ prepends the finite word w' to w.

We introduce the constants scan and sscan for lists and streams, respectively. They work like the identically named function in Haskell, in that they perform a fold with accumulation. That is, they fold over a list or stream and collect the state of the fold at each step and return this collection as a list or stream, respectively. Thus, unlike fold, it is also possible to define this function on infinite sequences.

We also introduce the constant "$\mathsf{infs} :: (\alpha \Rightarrow \mathsf{bool}) \Rightarrow \alpha\ \ \mathsf{stream} \Rightarrow \mathsf{bool}$" that indicates if a predicate is fulfilled infinitely often in a stream. We will use infs to define acceptance conditions for ω-automata.

Linear Temporal Logic

We base our contribution on the *LTL* entry found in the AFP [41] and extend it where necessary. The datatype we use for LTL syntactically enforces formulas to be in negation normal form. In order to preserve the expressiveness of LTL with negation, we need to include for the $\mathbf{U}$ (Until) operator its dual $\mathbf{R}$ (Release). For the logical decomposition result it is also essential to include $\mathbf{W}$ (Weak-Until) and $\mathbf{M}$ (Strong-Release). As usual we use $\mathbf{F}\ \varphi$ (Eventually) as an abbreviation for $\mathbf{tt}\ \mathbf{U}\ \varphi$ and $\mathbf{G}\ \psi$ (Always) for $\mathbf{ff}\ \mathbf{R}\ \psi$.

▶ **Definition 1** (Linear Temporal Logic).

$$\begin{aligned}\mathbf{datatype}\ \ \alpha\,\mathsf{ltl} = {} & \mathbf{tt} \mid \mathbf{ff} \mid \alpha \mid \neg\alpha \mid (\alpha\,\mathsf{ltl}) \wedge (\alpha\,\mathsf{ltl}) \mid (\alpha\,\mathsf{ltl}) \vee (\alpha\,\mathsf{ltl}) \mid \mathbf{X}\ (\alpha\,\mathsf{ltl}) \\ & \mid (\alpha\,\mathsf{ltl})\ \mathbf{U}\ (\alpha\,\mathsf{ltl}) \mid (\alpha\,\mathsf{ltl})\ \mathbf{R}\ (\alpha\,\mathsf{ltl}) \mid (\alpha\,\mathsf{ltl})\ \mathbf{W}\ (\alpha\,\mathsf{ltl}) \mid (\alpha\,\mathsf{ltl})\ \mathbf{M}\ (\alpha\,\mathsf{ltl})\end{aligned}$$

The type variable α determines the type of the atomic propositions. We write $\mathsf{atoms}\ \varphi$ to refer to the set of atomic propositions in a formula φ. The function $\mathsf{sf}\ \varphi$ computes all subformulas of φ, i.e., all subtrees of its syntax tree. Additionally, we define $\mathsf{subformulas}_\mu\ \varphi$ as set of subformulas of the form $\psi\ \mathbf{U}\ \chi$ or $\psi\ \mathbf{M}\ \chi$, and $\mathsf{subformulas}_\nu\ \varphi$ as the set of subformulas of the form $\psi\ \mathbf{R}\ \chi$ or $\psi\ \mathbf{W}\ \chi$.

▶ **Definition 2** (Semantics). *The entailment relation* $\models :: \alpha\ \mathsf{set}\ \mathsf{word} \Rightarrow \alpha\ \mathsf{ltl} \Rightarrow \mathsf{bool}$ *is defined recursively as follows:*

$$
\begin{array}{lll}
w \models \mathbf{tt} & & w \models \mathbf{X}\ \varphi \quad = \mathsf{suffix}\ 1\ w \models \varphi \\
w \not\models \mathbf{ff} & & \\
w \models a \quad = a \in w\ 0 & & w \models \varphi\ \mathbf{U}\ \psi = \exists i.\ \mathsf{suffix}\ i\ w \models \psi \wedge (\forall j < i.\ \mathsf{suffix}\ j\ w \models \varphi) \\
w \models \neg a \quad = a \notin w\ 0 & & w \models \varphi\ \mathbf{R}\ \psi = \forall i.\ \mathsf{suffix}\ i\ w \models \psi \vee (\exists j < i.\ \mathsf{suffix}\ j\ w \models \varphi) \\
w \models \varphi \wedge \psi = w \models \varphi \wedge w \models \psi & & w \models \varphi\ \mathbf{W}\ \psi = \forall i.\ \mathsf{suffix}\ i\ w \models \varphi \vee (\exists j \leq i.\ \mathsf{suffix}\ j\ w \models \psi) \\
w \models \varphi \vee \psi = w \models \varphi \vee w \models \psi & & w \models \varphi\ \mathbf{M}\ \psi = \exists i.\ \mathsf{suffix}\ i\ w \models \varphi \wedge (\forall j \leq i.\ \mathsf{suffix}\ j\ w \models \psi)
\end{array}
$$

We define the set of all words over an alphabet Σ *satisfying a formula* φ*:*

$$\mathsf{language}\ \Sigma\ \varphi = \{w.\ w \models \varphi \wedge \mathsf{range}\ w \subseteq \Sigma\}\,.$$

Equivalence Relations over LTL

We define three equivalence relations over LTL formulas: The largest equivalence relation is *language equivalence*. Two formulas are *(language-)equivalent* if they are satisfied by exactly the same words.

A smaller relation is defined by *propositional equivalence*. We interpret an LTL formula φ in propositional logic by treating every subformula that is a literal (a, $\neg a$) or a modal operator ($\mathbf{X}$, $\mathbf{U}$, $\mathbf{M}$, $\mathbf{R}$, $\mathbf{W}$) as a propositional variable. If a set of these subformulas $\mathcal{I}$ is a propositional model for φ, we write $\mathcal{I} \models_p \varphi$. Two formulas are *propositionally equivalent* if are satisfied by the same propositional models.

▶ **Definition 3** (Propositional Semantics). *The propositional entailment relation* $\models_p :: \alpha\ \mathsf{ltl}\ \mathsf{set} \Rightarrow \alpha\ \mathsf{ltl} \Rightarrow \mathsf{bool}$ *is defined recursively as follows:*

$$
\begin{array}{lll}
\mathcal{I} \models_p \mathbf{tt} & & \mathcal{I} \models_p \mathbf{X}\ \varphi \quad = (\mathbf{X}\ \varphi) \in \mathcal{I} \\
\mathcal{I} \not\models_p \mathbf{ff} & & \\
\mathcal{I} \models_p a \quad = a \in \mathcal{I} & & \mathcal{I} \models_p \varphi\ \mathbf{U}\ \psi = (\varphi\ \mathbf{U}\ \psi) \in \mathcal{I} \\
\mathcal{I} \models_p \neg a \quad = (\neg a) \in \mathcal{I} & & \mathcal{I} \models_p \varphi\ \mathbf{R}\ \psi = (\varphi\ \mathbf{R}\ \psi) \in \mathcal{I} \\
\mathcal{I} \models_p \varphi \wedge \psi = \mathcal{I} \models_p \varphi \wedge \mathcal{I} \models_p \psi & & \mathcal{I} \models_p \varphi\ \mathbf{W}\ \psi = (\varphi\ \mathbf{W}\ \psi) \in \mathcal{I} \\
\mathcal{I} \models_p \varphi \vee \psi = \mathcal{I} \models_p \varphi \vee \mathcal{I} \models_p \psi & & \mathcal{I} \models_p \varphi\ \mathbf{M}\ \psi = (\varphi\ \mathbf{M}\ \psi) \in \mathcal{I}
\end{array}
$$

Finally, *constants equivalence* is the smallest of the three equivalence relations. We use the function $\mathsf{eval} :: \alpha\ \mathsf{ltl} \Rightarrow \mathsf{tvl}$ with the three-valued logic "$\mathsf{tvl} = \mathsf{Yes} \mid \mathsf{No} \mid \mathsf{Maybe}$". It returns Yes iff φ is propositionally equivalent to $\mathbf{tt}$, and No iff φ is propositionally equivalent to $\mathbf{ff}$, respectively. Otherwise, Maybe is returned. The actual Isabelle formalisation does not refer to propositional equivalence, but in order to simplify the presentation we use the presented characterisation. Two formulas φ and ψ are *constants-equivalent* iff they are (syntactically) identical or "$\mathsf{eval}\ \varphi = \mathsf{eval}\ \psi \neq \mathsf{Maybe}$".

▶ **Definition 4** (Equivalence Relations). *For* $\varphi :: \alpha\ \mathsf{ltl}$ *and* $\psi :: \alpha\ \mathsf{ltl}$*, we define:*

$$
\begin{array}{lcl}
\varphi \sim_l \psi & = & \forall w.\ w \models \varphi \longleftrightarrow w \models \psi \\
\varphi \sim_p \psi & = & \forall \mathcal{I}.\ \mathcal{I} \models_p \varphi \longleftrightarrow \mathcal{I} \models_p \psi \\
\varphi \sim_c \psi & = & (\varphi = \psi \vee (\mathsf{eval}\ \varphi = \mathsf{eval}\ \psi \wedge \mathsf{eval}\ \psi \neq \mathsf{Maybe}))
\end{array}
$$

▶ **Lemma 5** (Order of Equivalence Relations).

$$\sim_c \leq \sim_p \leq \sim_l$$

Note that this order also corresponds to the computational complexity, with $\sim_c$ being the easiest to compute and $\sim_l$ the hardest.

3 Transition Systems and Automata

Automata are a popular subject in their own right in theoretical computer science and also have many applications, like regular expression matching and model checking. As such, it suggests itself to formalise these concepts separately and generically as a library to be shared. We first establish our goals for such a library. The deceptively simple term automaton covers a diverse range of objects that need to be supported. These differ in various ways, including but not limited to: successors (deterministic, nondeterministic), labelling (state-labeled, transition-labeled), and acceptance condition (finite, Büchi, Rabin, etc.). For each automaton type, we want to formalise fundamental concepts like path, reachability, and language. We would also like to formalise constructions like Boolean operations (union, intersection, complementation), and degeneralisation of Büchi acceptance conditions. As an overall goal, we want to share as much of the formalisation as possible by keeping it abstract. This avoids duplication and often makes definitions and proofs simpler and more elegant. Finally, we want to do all of this while providing good usability and automation, especially concerning the basic concepts that constitute the foundation of the library.

With these goals in mind, we look at the formalisations that are already available in the Isabelle ecosystem. First off, there are many ad-hoc formalisations of transition systems and automata done as part of other formalisations [40, 21, 1, 31]. Furthermore, there have been a few major formalisations as part of the Cava project [21], although not all of them were preserved or published. Stephan Merz and Alexander Schimpf formalised NBAs and NGBAs in preliminary work of the Cava project [38] and then later as part of Cava itself. Peter Lammich is the author of the current Cava automata library [28], which includes state-labeled NBAs and NGBAs. These formalisations cover a very specific set of automata, making them convenient to use, but only if one happens to need exactly that type of automaton. Another unpublished formalisation by Thomas Tuerk is more generic and covers DFAs, NFAs, NBAs, and NGBAs. It achieves this genericity by modelling some of these automata as special cases of others, which allows for sharing of definitions and proofs. For instance, a deterministic transition system would naturally be modelled using the type "$\alpha \Rightarrow \rho \Rightarrow \rho$". Alternatively, it can also be treated as a special case of a nondeterministic transition system with the type "$(\rho \times \alpha \times \rho)\,\mathsf{set}$". However, this causes several issues. Firstly, since the type is too weak, a uniqueness predicate on the term level is needed to only allow those transition relations that act like functions. These predicates then have to be carried around in all proofs explicitly, rather than being encoded in the type. Secondly, due to the type being a poor fit, we can no longer do things like folding over the successor function. Lastly, the user is restricted to a single representation, rather than, for instance, being able to choose between explicit ("$(\rho \times \alpha \times \rho)\,\mathsf{set}$") and implicit ("$\alpha \Rightarrow \rho \Rightarrow \rho\,\mathsf{set}$") representations.

We use these experiences to design a new architecture in order to achieve the goals we set earlier. Since our primary goal is sharing via abstraction, this is what will mainly motivate our decisions. There are two observations to be made. Firstly, acceptance conditions are far too diverse and specific to be treated abstractly. Thus, our abstract representation will cover transition systems instead of automata, with acceptance conditions being added on a more concrete level at a later stage. This idea is not new and was in fact used in most of the earlier formalisations as well. Secondly, as mentioned in the previous paragraph, specialisation as a mechanism of abstraction has various issues. Instead, we choose to use the mechanism of instantiation via locales (Section 2), the advantages of which will become apparent in the following sections. Thus, the library formalises *abstract transition systems* (Section 3.1), which are then instantiated and used as building blocks for *concrete automata* (Section 3.2).

We try to formalise as much as possible in the context of abstract transition systems, since this both often leads to elegance and conciseness and is shared between all concrete automata. Thanks to this, adding a new automaton requires only a minimal amount of setup, allowing users to use the library in conjunction with their own custom automata representations. That being said, the set of automata supplied with the library is also growing and becoming more useful, making this less and less necessary. In the end, we supply both a collection of useful automata as well as the tools to easily add custom ones as needed.

3.1 Abstract Transition Systems

Having decided on our architecture, the central decision lies in the specification of the locale for transition systems. We focus on the defining property of a transition system: its ability to use transitions to move from state to state. This leads us directly to the specification in terms of its types (*transition* and *state*) and its terms (execute and enabled).

▶ **Definition 6.**

locale transition-system =
  fixes execute :: *transition* ⇒ *state* ⇒ *state*
  fixes enabled :: *transition* ⇒ *state* ⇒ bool

Given a transition and a source state, the function execute specifies the target state for that transition. Analogously, the function enabled determines whether the given transition is enabled at the given source state. Together, these functions capture the essence of a transition system in terms of its ability to transition between states. Given the types, it may seem appealing to combine both constants into a single one with result type "*state* option". This sounds great in theory, but unfortunately, is very inconvenient to work with in practice. It mixes the issue of finding the target of a transition with that of whether that transition was valid in the first place. Keeping these two things separate makes definitions simpler and allows for better automation in proofs.

Having defined the transition-system locale, we now develop some abstract theory within this context. So far, we can only execute single transitions, so we look at finite and infinite sequences of transitions. We introduce the following constants based on the execute function.

▶ **Definition 7.**

target = fold execute :: *transition* list ⇒ *state* ⇒ *state*
trace = scan execute :: *transition* list ⇒ *state* ⇒ *state* list
strace = sscan execute :: *transition* stream ⇒ *state* ⇒ *state* stream

Given a sequence of transitions and a source state, these functions give the target state and the finite and infinite sequence of traversed states, respectively. Note both the simplicity and elegance of these definitions and how each of them is simply a lifted version of execute.

We can do something similar for the enabled function.

▶ **Definition 8.**

inductive path :: *transition* list ⇒ *state* ⇒ bool **where**
  path [] *p*
  enabled *a p* ⟹ path *r* (execute *a p*) ⟹ path (*a* # *r*) *p*
coinductive spath :: *transition* stream ⇒ *state* ⇒ bool **where**
  enabled *a p* ⟹ spath *r* (execute *a p*) ⟹ spath (*a* ## *r*) *p*

These constants are (co)inductively defined predicates that capture the notion of all the transitions in a sequence being enabled at their respective states. Like in the previous paragraph, these are basically lifted versions of enabled, which is also reflected in their types.

Together, these form the very foundation of the library, since almost every other concept is in some way related to sequences of transitions. The nice thing about these definitions is that they lend themselves very well to automation. In the case of definitions lifted from execute, we can define simplification rules. In the case of definitions lifted from enabled, we can define safe introduction and elimination rules. This works for both the constructors of sequences (# and ##), as well as the operators for concatenation (@, @−). Convenience and automation regarding the basic concepts was a major shortcoming of earlier libraries.

Next, we define the constant reachable for the set of reachable states from a source state. Like path, this is an inductively defined predicate. Alternatively, we could have defined reachable in terms of target and path. Instead, it is defined directly based on execute and enabled and the connection to target and path is shown as a lemma.

There are some interesting things we can formalise even on this very abstract level. We present one such example in the construction of infinite paths.

▶ **Lemma 9** (Recurring Condition).

$$\begin{aligned}
&\textbf{fixes } P :: \mathit{state} \Rightarrow \mathsf{bool} \textbf{ and } p :: \mathit{state} \\
&\textbf{assumes } P\ p \textbf{ and } \bigwedge p.\ P\ p \implies \exists r.\ r \neq [] \wedge \mathsf{path}\ r\ p \wedge P\ (\mathsf{target}\ r\ p) \\
&\textbf{obtains } r :: \mathit{transition}\,\mathsf{stream} \\
&\textbf{where } \mathsf{spath}\ r\ p \textbf{ and } \mathsf{infs}\ P\ (p\ \#\#\ \mathsf{strace}\ r\ p)
\end{aligned}$$

Here, the premises only guarantee the repeated existence of a finite extension to an existing finite path, which we want to use to construct an infinite path. Proving a statement like this is cumbersome, as it requires skolemisation of the premise, construction of a stream via iteration combinators and finally proving the properties via coinduction. By providing generic rules like these, all this complexity is hidden and users can restrict themselves to easy-to-work-with constants like spath and infs.

3.2 Concrete Automata

In order for our formalisation of abstract transition systems to be useful, it needs to be able to express a wide range of transition system types and their representations. We now present instantiations for various transition systems with labels of type α and states of type ρ.

Given a successor function "$\mathsf{succ} :: \alpha \Rightarrow \rho \Rightarrow \rho\,\mathsf{option}$", we instantiate as follows.

▶ **Example 10** (Incomplete Deterministic Transition System).

$$\begin{aligned}
\mathsf{execute} &= \lambda a\ p.\ \mathsf{the}\ (\mathsf{succ}\ a\ p) & \mathit{transition} &= \alpha \\
\mathsf{enabled} &= \lambda a\ p.\ \mathsf{succ}\ a\ p \neq \mathsf{None} & \mathit{state} &= \rho
\end{aligned}$$

Note how the deterministic successor function fits the interface straightforwardly.

Given a successor function "$\mathsf{succ} :: \alpha \Rightarrow \rho \Rightarrow \rho$" we instantiate as follows.

▶ **Example 11** (Complete Deterministic Transition System).

$$\begin{aligned}
\mathsf{execute} &= \mathsf{succ} & \mathit{transition} &= \alpha \\
\mathsf{enabled} &= \top & \mathit{state} &= \rho
\end{aligned}$$

Things get interesting when considering "succ :: $\alpha \Rightarrow \rho \Rightarrow \rho$ set". Textbooks teach us that deterministic transitions systems are a special case of nondeterministic ones. At first glance, it may seem like we are trying to do the impossible opposite here. However, since we get to instantiate the type variables, there is a surprisingly elegant solution.

▶ **Example 12** (Implicit Nondeterministic Transition System).

$$\text{execute} = \lambda(a, q)\ p.\ q \qquad \textit{transition} = \alpha \times \rho$$
$$\text{enabled} = \lambda(a, q)\ p.\ q \in \text{succ}\ a\ p \qquad \textit{state} = \rho$$

Note how unlike in the first two examples, the type variable *transition* gets instantiated in a nontrivial way. While it may seem backwards at first, this actually works out perfectly and gives our constants the strongest possible type for this scenario. For instance, we get "path :: $(\alpha \times \rho)$ list $\Rightarrow \rho \Rightarrow$ bool". That is, the path predicate expects a source state as well as a list of the traversed labels and states. This expression contains exactly the necessary amount of information, nothing more, nothing less. Note that the fact that we are dealing with pairs is not an issue, as Isabelle has good automation for those. We also added some more automation for sequences of pairs as part of this library. In the end, neither the deterministic nor the nondeterministic case necessitates inconvenient wellformedness predicates while sharing the same abstract formalisation.

Finally, we consider an explicit representation in "trans :: $(\rho \times \alpha \times \rho)$ set". Being isomorphic to the previous case, the type variables as well as execute are instantiated the same way.

▶ **Example 13** (Explicit Nondeterministic Transition System).

$$\text{execute} = \lambda(a, q)\ p.\ q \qquad \textit{transition} = \alpha \times \rho$$
$$\text{enabled} = \lambda(a, q)\ p.\ (p, a, q) \in \text{trans} \qquad \textit{state} = \rho$$

Unsurprisingly, an isomorphic change in representation does not make a difference since the instantiation absorbs such details.

Having shown that we can instantiate a variety of transition systems using our abstract theory, we can now use these as building blocks for concrete automata. Since the abstraction is achieved via type instantiation and locales, it only minimally impacts the usability compared to a fully specific formalisation. Moreover, since it does not restrict the type of the automaton at all, the user can use a representation that exactly fits their needs.

There are some definitions that one would expect to be part of a general automata library that unfortunately cannot be formalised on transition systems. One of these are Boolean operations, since they require information about the automaton's successors, labelling, and acceptance condition. With some effort, they could be formalised on intermediate abstraction over a family of similar automata (for instance, DBA, DCA, DRA). However, we could not justify the effort for our purposes, since these formalisations do not contain much substance.

Degeneralisation, which plays an important part in defining aforementioned Boolean operations, can be generalised a little easier. The reason for this is that it is independent from successors and labelling, requiring only the concept of state-based Büchi acceptance. Thanks to this, we were able to abstractly formalise degeneralisation in a transition system locale augmented with an acceptance condition. This intermediate abstraction is then instantiated in order to facilitate the formalisation of Boolean operations on DBAs and DCAs.

3.3 Predefined Automata

While the focus of the automata library is on the abstract part and the provision of tools to build concrete automata, it also comes with a growing collection of the latter. At the time of writing, it contains (non)deterministic finite automata, (non)deterministic Büchi automata, as well as deterministic co-Büchi and Rabin automata. Each of these incurs around 50 lines of proof text in order to set-up the automaton and to define its language. The latter is fairly simple to achieve, as all the constituents (paths and acceptance conditions) are already available and just need to be composed to yield a language definition.

3.4 Executable Implementation

One of our goals is also the ability to implement executable versions of some algorithms. As mentioned earlier, we will use the refinement frameworks and the Isabelle code generator for this. Most of this needs to be done on concrete automata, as it depends on details of the representation. Furthermore, in many cases it is advantageous to be able to choose data structures depending on the representation. Because of these reasons, all the executable implementations are done on the concrete level, with only some proofs being reused.

We build on existing algorithms for graph structures to implement versions that work with automata. For instance, we use the AFP entry about depth-first search [32, 33] to explore all reachable states of an automaton. This is used to generate explicit representations of automata in order to be able to serialise and output them. In the case of NBAs we consider the successor function "$\mathsf{succ} :: \alpha \Rightarrow \rho \Rightarrow \rho\,\mathsf{set}$", which implicitly represents the transitions of the automaton. The algorithm can then turn this into an explicit set of transitions "$\mathsf{trans} :: (\rho \times \alpha \times \rho)\,\mathsf{set}$". We also implement an algorithm for translating an automaton with an arbitrary state type into one whose states are natural numbers. Furthermore, we use the AFP entry about Gabow's algorithm for strongly-connected components [29, 30] to decide language emptiness of NBAs.

3.5 Formalisation

The library is available in the form of the AFP entry *Transition Systems and Automata* [10]. At the time of writing, it comprises about 5800 lines of theory text. Other than in this paper, the library is used in the partial order reduction optimisation [12, 11] of the Cava model checker [21]. It is also used as the foundation of the AFP entry about rank-based complementation of Büchi automata [9].

3.6 Contributions to the Translation Formalisation

For this paper, we contribute deterministic Büchi, co-Büchi, and Rabin automata. For instance, the constructor for deterministic Büchi automata is "$\mathsf{dba} :: \alpha\,\mathsf{set} \Rightarrow \rho \Rightarrow (\alpha \Rightarrow \rho \Rightarrow \rho) \Rightarrow (\rho \Rightarrow \mathsf{bool}) \Rightarrow (\alpha, \rho)\ \mathsf{dba}$". Furthermore, we add corresponding union and intersection operations to the library (Figure 1). In addition to those operations, we also implement a specialised operation dbcrai that provides the intersection of a DBA and a DCA resulting in a DRA. We prove both their correctness in terms of language as well as upper bounds on the number of states of the resulting automata. Since the resulting automata are implicit, we also provide an executable algorithm for exploration and subsequent conversion to an explicit representation together with a numbering of the states.

Automaton	$\cap$ (Pair)	$\bigcap$ (List)	$\cup$ (Pair)	$\bigcup$ (List)
DBA		dbail	dbau	dbaul
DCA	dcai	dcail		dcaul
DRA				draul

Figure 1 Boolean Operations on Deterministic ω-Automata. Shown are the Boolean operations that were implemented for deterministic Büchi, co-Büchi, and Rabin automata.

4 The Master Theorem: Decomposing LTL Formulas

The centrepiece for all translations is the *Master Theorem* [19] that decomposes LTL formulas into a Boolean combination, in our case union and intersection, of "simple" languages. We will recall important definitions from [19] in order to state the theorem itself and to highlight obstacles we encountered in our formalisation. For an in-depth discussion and exposition of the theory and its proof we refer the reader to the primary source [19].

We will now introduce the functions used in the scope of the Master Theorem: the "after"-function $\mathsf{af}\ \varphi\ w$, read "φ after w", and the two "advice" functions $\varphi[X]_\nu$ and $\psi[Y]_\mu$ which are pronounced as "φ with **GF**-advice X" and "ψ with **FG**-advice Y", respectively.

4.1 The "after"-Function

Let us begin with the definition of the "after"-function [18, 17, 19]. The function application $\mathsf{af}\ \varphi\ w$ computes a new formula such that for every infinite word w' we have:

▶ **Lemma 14** ([19]).

$$w \frown w' \models \varphi \iff w' \models \mathsf{af}\ \varphi\ w.$$

We can intuitively see af as a function that returns a formula representing the language that we obtain *after* reading the prefix w. We achieve this by using well-known LTL expansion rules combined with partial evaluation.

▶ **Definition 15** ("after"-Function [19]). *The function* $\mathsf{af} :: \alpha\ \mathsf{ltl} \Rightarrow \alpha\ \mathsf{set} \Rightarrow \alpha\ \mathsf{ltl}$ *is defined for a single letter recursively as follows:*

$$\begin{aligned}
\mathsf{af}\ \mathbf{tt}\ \sigma &= \mathbf{tt} & \mathsf{af}\ (\mathbf{X}\ \varphi)\ \sigma &= \varphi \\
\mathsf{af}\ \mathbf{ff}\ \sigma &= \mathbf{ff} & & \\
\mathsf{af}\ a\ \sigma &= \mathbf{if}\ a \in \sigma\ \mathbf{then}\ \mathbf{tt}\ \mathbf{else}\ \mathbf{ff} & \mathsf{af}\ (\varphi\ \mathbf{U}\ \psi)\ \sigma &= (\mathsf{af}\ \psi\ \sigma) \vee ((\mathsf{af}\ \varphi\ \sigma) \wedge (\varphi\ \mathbf{U}\ \psi)) \\
\mathsf{af}\ (\neg a)\ \sigma &= \mathbf{if}\ a \notin \sigma\ \mathbf{then}\ \mathbf{tt}\ \mathbf{else}\ \mathbf{ff} & \mathsf{af}\ (\varphi\ \mathbf{R}\ \psi)\ \sigma &= (\mathsf{af}\ \psi\ \sigma) \wedge ((\mathsf{af}\ \varphi\ \sigma) \vee (\varphi\ \mathbf{R}\ \psi)) \\
\mathsf{af}\ (\varphi \wedge \psi)\ \sigma &= (\mathsf{af}\ \varphi\ \sigma) \wedge (\mathsf{af}\ \psi\ \sigma) & \mathsf{af}\ (\varphi\ \mathbf{W}\ \psi)\ \sigma &= (\mathsf{af}\ \psi\ \sigma) \vee ((\mathsf{af}\ \varphi\ \sigma) \wedge (\varphi\ \mathbf{W}\ \psi)) \\
\mathsf{af}\ (\varphi \vee \psi)\ \sigma &= (\mathsf{af}\ \varphi\ \sigma) \vee (\mathsf{af}\ \psi\ \sigma) & \mathsf{af}\ (\varphi\ \mathbf{M}\ \psi)\ \sigma &= (\mathsf{af}\ \psi\ \sigma) \wedge ((\mathsf{af}\ \varphi\ \sigma) \vee (\varphi\ \mathbf{M}\ \psi))
\end{aligned}$$

We generalise this definition to finite words by overloading $\mathsf{af} :: \alpha\ \mathsf{ltl} \Rightarrow \alpha\ \mathsf{set\ list} \Rightarrow \alpha\ \mathsf{ltl}$*:*

$$\mathsf{af}\ \varphi\ w = \mathsf{foldl}\ \mathsf{af}\ \varphi\ w.$$

▶ Remark 16. The reader might have noticed that the definition of af resembles the idea of Brzozowski's derivatives for regular expressions [13]. In fact, as we will see later, the DRA construction relies on af and the previously introduced LTL equivalence relations again mirroring the idea of Brzozowski. However, this approach alone can only be applied to fragments of LTL.

4.2 Syntactic Fragments of LTL

We already teased the idea of the "simple" languages, but what is special about these? What is the mechanism to achieve this? These languages are made simple by the fact that they can be expressed by fragments of LTL. To be more precise, let μLTL be the fragment that only contains modal operators that can be expressed as least-fixed points, i.e., we disallow the operators $\mathbf{R}$ and $\mathbf{W}$. Dually, νLTL contains only modal operators that can be expressed as greatest-fixed points, i.e., we disallow the operators $\mathbf{U}$ and $\mathbf{M}$. The fragments $\mathbf{GF}(\mu\text{LTL})$ and $\mathbf{FG}(\nu\text{LTL})$ contain all formulas $\mathbf{GF}\varphi$ and $\mathbf{FG}\psi$ where $\varphi \in \mu\text{LTL}$ and $\psi \in \nu\text{LTL}$, respectively. For these fragments one can easily define translations to NBAs or DRAs, e.g. [19].

Let us now think about how to make use of this: Assume one gets a promise set $X = \{a\,\mathbf{U}\,b\}$ guaranteeing that $a\,\mathbf{U}\,b$ holds infinitely often, i.e., $w \models \mathbf{GF}(a\,\mathbf{U}\,b)$, and assume we have access to a translation for νLTL. Can we simplify $\varphi = \mathbf{G}(a\,\mathbf{U}\,b) \vee \mathbf{G}c$ with this information? Since $w \models \mathbf{GF}(a\,\mathbf{U}\,b)$ implies that b is infinitely often true, we can replace the $\mathbf{U}$ by an $\mathbf{W}$. Under the assumption that X is a correct promise, we simplify φ to an equivalent formula $\mathbf{G}(a\,\mathbf{W}\,b) \vee \mathbf{G}c$ which is a formula of νLTL. Then we can apply our translation for the νLTL fragment.

Formally, we define the functions $\varphi[X]_\nu$ and $\varphi[Y]_\mu$ such that $\varphi[X]_\nu$ takes a promise set X and produces a formula of νLTL, and such that $\varphi[Y]_\mu$ takes a promise set Y and produces a formula of μLTL:

▶ **Definition 17** ("Advice"-Functions [19]). *The function* $\cdot\,[\,\cdot\,]_\nu :: \alpha\,\mathsf{ltl} \Rightarrow \alpha\,\mathsf{ltl\,set} \Rightarrow \alpha\,\mathsf{ltl}$ *is defined for the cases* $\mathbf{U}$ *and* $\mathbf{M}$ *as follows:*

$$(\varphi\,\mathbf{U}\,\psi)[X]_\nu = \textbf{if } (\varphi\,\mathbf{U}\,\psi) \in X \textbf{ then } (\varphi[X]_\nu)\,\mathbf{W}\,(\psi[X]_\nu) \textbf{ else ff}$$
$$(\varphi\,\mathbf{M}\,\psi)[X]_\nu = \textbf{if } (\varphi\,\mathbf{M}\,\psi) \in X \textbf{ then } (\varphi[X]_\nu)\,\mathbf{R}\,(\psi[X]_\nu) \textbf{ else ff}$$

The function $\cdot\,[\,\cdot\,]_\mu :: \alpha\,\mathsf{ltl} \Rightarrow \alpha\,\mathsf{ltl\,set} \Rightarrow \alpha\,\mathsf{ltl}$ *is defined for the cases* $\mathbf{R}$ *and* $\mathbf{W}$ *as follows:*

$$(\varphi\,\mathbf{R}\,\psi)[Y]_\mu = \textbf{if } (\varphi\,\mathbf{R}\,\psi) \in Y \textbf{ then tt else } (\varphi[Y]_\mu)\,\mathbf{M}\,(\psi[Y]_\mu)$$
$$(\varphi\,\mathbf{W}\,\psi)[Y]_\mu = \textbf{if } (\varphi\,\mathbf{W}\,\psi) \in Y \textbf{ then tt else } (\varphi[Y]_\mu)\,\mathbf{U}\,(\psi[Y]_\mu)$$

For all other cases, both functions are defined as a recursive descent over the syntax tree.

4.3 The Master Theorem

We are now equipped with the necessary definitions to state the Master Theorem. Note that the formulation we use is taken nearly verbatim from the Isabelle theory, apart from the annotations $L^1_{\varphi,X}$, $L^2_{X,Y}$, and $L^3_{X,Y}$ that we added to relate to the introduction.

▶ **Theorem 18** (Master Theorem [19]).

$$\begin{aligned} w \models \varphi \iff (\exists X \subseteq \mathsf{subformulas}_\mu\ \varphi.\ \exists Y \subseteq \mathsf{subformulas}_\nu\ \varphi.&\\ (\exists i.\ \mathsf{suffix}\ i\ w \models \mathsf{af}\ \varphi\ (\mathsf{prefix}\ i\ w)[X]_\nu) &\quad -- L^1_{\varphi,X}\\ \wedge\ (\forall \psi \in X.\ w \models \mathbf{G}\ (\mathbf{F}\ \psi[Y]_\mu)) &\quad -- L^2_{X,Y}\\ \wedge\ (\forall \psi \in Y.\ w \models \mathbf{F}\ (\mathbf{G}\ \psi[X]_\nu)) &\quad -- L^3_{X,Y} \end{aligned}$$

The proof of this theorem intrinsically depends on the fact that we can check promise sets bottom-up, as formalised by the following lemma. We highlight this intermediate lemma, because we needed to introduce a custom induction mechanism over finite sets to our theory. The remaining material needed to show Theorem 18 is obtained in straight-forward manner and closely resembles the proofs of [19].

▶ **Lemma 19** ([19]).

$$\begin{aligned}
&\textbf{fixes } w :: \alpha\,\mathsf{set\,word} \textbf{ and } \varphi :: \alpha\,\mathsf{ltl}\\
&\textbf{assumes } X \subseteq \mathsf{subformulas}_\mu\ \varphi \textbf{ and } Y \subseteq \mathsf{subformulas}_\nu\ \varphi\\
&\textbf{and } \forall \psi \in X.\ w \models \mathbf{G}\ (\mathbf{F}\ \psi[Y]_\mu) \textbf{ and } \forall \psi \in Y.\ w \models \mathbf{F}\ (\mathbf{G}\ \psi[X]_\nu)\\
&\textbf{shows } \forall \psi \in X.\ w \models \mathbf{G}\ (\mathbf{F}\ \psi) \textbf{ and } \forall \psi \in Y.\ w \models \mathbf{F}\ (\mathbf{G}\ \psi)
\end{aligned}$$

The corresponding proof from [19] proceeds by constructing a sequence of pairs (X_i, Y_i) where we have $(X_0, Y_0) = (\emptyset, \emptyset)$ and $(X_n, Y_n) = (X, Y)$. Moreover, in each step a single formula $\psi_i \in X \uplus Y$ is added to either X_i or Y_i, depending on whether $\psi_i \in X$ or $\psi_i \in Y$. However, ψ_i cannot be chosen arbitrarily and ψ_i must respect the subformula order, i.e., if $\psi_i \in \mathsf{sf}\ \psi_j$, then $i \leq j$. Then the proof proceeds by an induction over this sequence.

Since to the best of our knowledge there has been at the time of writing no matching induction rule in Isabelle or its libraries, we derived a suitable induction rule for our purposes. First, note that instead of sorting the formulas by the subformula order, it is sufficient to order them by their size, because all subformulas of a formula φ are smaller than φ. Second, an induction over pairs of sets seemed inconvenient to us in the context of our theorem prover. Hence we combined the two disjoint sets into a single one and used a suitable case distinction. Finally, we arrived at the following, general induction rule[4] for finite sets with an additional order constraint:

▶ **Lemma 20** (Finite Ordered Induction).

$$\begin{aligned}
&\textbf{fixes } S :: \alpha\,\mathsf{set} \textbf{ and } P :: \alpha\,\mathsf{set} \Rightarrow \mathsf{bool} \textbf{ and } f :: \alpha \Rightarrow (\beta :: \mathsf{linorder})\\
&\textbf{assumes } \mathsf{finite}\ S \textbf{ and } P\ \emptyset\\
&\textbf{and } \bigwedge x\ S.\ \mathsf{finite}\ S \wedge (\forall y.\ y \in S \longrightarrow f\ y \leq f\ x) \wedge P\ S \implies P\ (\mathsf{insert}\ x\ S)\\
&\textbf{shows } P\ S
\end{aligned}$$

5 Deriving the DRA Construction

With the necessary decomposition theorem in place, we now can follow our automata construction blue-print to obtain a translation from LTL to DRAs. We will first build automata for $L^1_{\varphi,X}$, $L^2_{X,Y}$, and $L^3_{X,Y}$, named $\mathfrak{A}_1$, $\mathfrak{A}_2$, and $\mathfrak{A}_3$, respectively. In the subsequent section, we will assemble these pieces to the final automaton and end the section with a description of the extracted, verified tool.

5.1 Constructing Automata for $L^1_{\varphi,X}$, $L^2_{X,Y}$, and $L^3_{X,Y}$

We parametrise our automata constructions for the "simple" components by an equivalence relation $\sim$. The most important requirement for $\sim$ is that $\sim_c \leq \sim \leq \sim_l$ holds, i.e., that $\sim$ does not consider two formulas with different languages equivalent and $\sim$ eventually detects equivalence to $\mathbf{tt}$ and $\mathbf{ff}$ for certain fragments. This abstraction has two advantages over fixing a concrete equivalence: first, our proofs stay as abstract as possible and the proof automation does not rely accidentally on irrelevant properties of the chosen equivalence relation; second, we can instantiate the final automaton with any suitable equivalence relation. In Section 5.3 we exemplarily use propositional equivalence but one can easily replace it by a different equivalence without any additional effort to speak of.

[4] This induction rule has now been included in Isabelle/HOL, is located in `HOL/Lattices_Big.thy`, and is named `finite_ranking_induct`.

In this paper, we will only discuss the construction of $\mathfrak{A}_2$ for $L^2_{X,Y}$. The constructions for $L^1_{\varphi,X}$ and $L^3_{X,Y}$ as defined by [19] are formalised analogously. Remember that $L^2_{X,Y}$ is defined as "$\bigcap \psi \in X$. language UNIV ($\mathbf{GF}(\psi[Y]_\mu)$)" for the finite sets X and Y. Hence it suffices to define a translation for formulas of the fragment $\mathbf{GF}(\mu\text{LTL})$ and then apply the intersection construction from the automaton library.

For the translation of formulas from the fragment $\mathbf{GF}(\mu\text{LTL})$ we make use of the following lemma. It states the we can monitor a formula from μLTL using af and the constrained equivalence relation $\sim$, and if a word satisfies the formula, then we will notice this after a finite amount of steps. Furthermore, the lemma states that we can deal with $\mathbf{GF}(\mu\text{LTL})$ by repeatedly doing this:

▶ **Lemma 21** (Logical Characterisation of μLTL and $\mathbf{GF}(\mu\text{LTL})$ [19, 42][5]).

$$\begin{aligned}
&\textbf{assumes } \varphi \in \mu\text{LTL} \textbf{ and } \sim_c \leq \sim \leq \sim_l \\
&\quad \textbf{shows } w \models \varphi \iff \exists i.\ \mathsf{af}\ \varphi\ (\mathsf{prefix}\ i\ w) \sim \mathsf{tt} \\
&\quad\quad \textbf{and } w \models \mathbf{G}\ (\mathbf{F}\ \varphi) \iff \forall i.\ \exists j.\ \mathsf{af}\ (\mathbf{F}\ \varphi)\ (\mathsf{prefix}\ j\ (\mathsf{suffix}\ i\ w)) \sim \mathsf{tt}
\end{aligned}$$

Since $\sim$ is such a fundamental ingredient throughout the formalisation of the automata constructions, we use locales in Isabelle to fix $\sim$ and assumptions about it. In particular, we use the equivalence classes of $\sim$ as states in our constructed automata. To define the quotient type for a given equivalence relation, we use the Isabelle's *Quotient* package introduced in [8] and revised in [24]. However, it is not possible to define such a quotient type within a locale. Thus we present a primitive, ad-hoc mechanism to simulate the quotient type in our locale. We fix a type parameter γ and the functions Rep and Abs that compute the representative of an equivalence class and the equivalence class of a formula, respectively. In other words we use Rep and Abs to map between equivalence classes and representatives. Further, we assume the quotient type invariant "Abs (Rep x) = x" and require that equality on γ is equivalent to $\sim$ on formulas. Thus we can pretend γ to be a quotient type over $\sim$ which resembles "duck typing" found in programming languages such as Python.

▶ **Definition 22** (Locale for LTL to DRA translation[6]).

$$\begin{aligned}
&\textbf{locale } \mathsf{ltl\text{-}to\text{-}dra} = \\
&\qquad \textbf{fixes } \sim :: \alpha\ \mathsf{ltl} \Rightarrow \alpha\ \mathsf{ltl} \Rightarrow \mathsf{bool} \\
&\qquad\ \textbf{and } \mathsf{Rep} :: \gamma \Rightarrow \alpha\ \mathsf{ltl}\ \ \textbf{and } \mathsf{Abs} :: \alpha\ \mathsf{ltl} \Rightarrow \gamma \\
&\quad \textbf{assumes } \mathsf{equivp} \sim \textbf{ and } \sim_c \leq \sim \leq \sim_l \\
&\qquad\ \textbf{and } \mathsf{Abs}\ (\mathsf{Rep}\ x) = x\ \textbf{ and } \mathsf{Abs}\ \varphi = \mathsf{Abs}\ \psi \longleftrightarrow \varphi \sim \psi \\
&\qquad\ \textbf{and } \varphi \sim \psi \implies (\mathsf{af}\ \varphi\ \sigma \sim \mathsf{af}\ \psi\ \sigma) \wedge (\varphi[X]_\nu \sim \psi[X]_\nu)
\end{aligned}$$

In this definition two new assumptions can be found that we have not talked about yet: We also demand that af and $\cdot\,[\,\cdot\,]_\nu$ are congruent with respect to $\sim$. This is due to the fact that our the automata use equivalence classes as states and for computing the successor with af the choice of the representative must be irrelevant.

[5] This lemma is a generalised version of [19] which only considers the special case for $\sim_p$.

[6] We only present the final combination of several locales defined in our Isabelle formalisation to give an overview of all assumptions required by our proofs.

Within this locale we now define the deterministic Büchi automaton $\mathfrak{A}_\mu^{\mathbf{GF}}$ for a single formula of the fragment $\mathbf{GF}(\mu\text{LTL})$. The DBA $\mathfrak{A}_2$ for $L^2_{X,Y}$ is then computed by a Büchi intersection (dbail). Note that this intersection construction requires the operands to be ordered. Hence we represent the advice sets X and Y as the lists xs and ys and propagate this order to dbail.

▶ **Definition 23.**

$$\mathfrak{A}_\mu^{\mathbf{GF}}\ \varphi = \mathsf{dba}\ \mathsf{UNIV}\ (\mathsf{Abs}\ (\mathbf{F}\ \varphi))\ (\mathsf{af}_F\ \varphi)\ (\lambda\psi.\ \psi = \mathsf{Abs}\ \mathbf{tt})$$
$$\mathsf{af}_F\ \varphi\ \sigma\ \psi = \mathbf{if}\ \ \psi = \mathsf{Abs}\ \mathbf{tt}\ \ \mathbf{then}\ \ \mathsf{Abs}\ (\mathbf{F}\ \varphi)\ \ \mathbf{else}\ \ \mathsf{Abs}\ (\mathsf{af}\ (\mathsf{Rep}\ \psi)\ \sigma)$$
$$\mathfrak{A}_2\ xs\ ys = \mathsf{dbail}\ (\mathsf{map}\ (\lambda\psi.\ \mathfrak{A}_\mu^{\mathbf{GF}}\ (\psi[\mathsf{set}\ ys]_\mu))\ xs)$$

Using Lemma 21 we show correctness for a single component and using the lemmas from the automata library we also prove the intersection correct. The constructions for $L^1_{\varphi,X}$ and $L^3_{X,Y}$ are analogous and thus skipped from the presentation in this paper.

5.2 Assembling the Pieces

It now remains to intersect the (co-)Büchi automata "$\mathfrak{A}_1\ \varphi\ xs$", "$\mathfrak{A}_2\ xs\ ys$", and "$\mathfrak{A}_3\ xs\ ys$", representing $L^1_{\varphi,X}$, $L^2_{X,Y}$, and $L^3_{X,Y}$, respectively. Again we need to use a list representation for X and Y to fix an iteration order and thus we use xs and ys. We call the resulting Rabin automaton "$\mathfrak{A}\ \varphi\ xs\ ys$". To finish the construction, we then iterate over all possible choices for $X \subseteq \mathsf{subformulas}_\mu\ \varphi$ and $Y \subseteq \mathsf{subformulas}_\nu\ \varphi$ and take the union of all languages accepted by "$\mathfrak{A}\ \varphi\ xs\ ys$" with draul (DRA union):

▶ **Definition 24.**

$$\mathsf{ltl\text{-}to\text{-}dra}\ \varphi = \mathsf{draul}\ (\mathsf{map}\ (\lambda(xs, ys).\ \mathfrak{A}\ \varphi\ xs\ ys)\ (\mathsf{advice\text{-}sets}\ \varphi)).$$

Using the Master Theorem (Theorem 18) and the correctness lemmas for the intermediate constructions, we obtain the correctness of the translation:

▶ **Theorem 25.**

$$\mathsf{language}\ (\mathsf{ltl\text{-}to\text{-}dra}\ \varphi) = \mathsf{language}\ \mathsf{UNIV}\ \varphi.$$

5.3 A Verified LTL Translator

We extract the executable translation of LTL formulas into ω-automata by instantiating the locale with a suitable equivalence relation. As mentioned above we use $\sim_p$ and we show for this equivalence relation that the constructed automaton indeed has at most a double-exponential number of states in the size of the formula. Hence an exploration by depth-first search terminates, and more importantly, this makes the construction the first LTL to DRA translation with a formally verified double exponential size bound.

▶ **Lemma 26.**

$$\mathsf{card}\ (\mathsf{nodes}\ (\mathsf{ltl\text{-}to\text{-}dra}\ \varphi)) \leq 2\ \hat{}\ 2\ \hat{}\ (2 * \mathsf{size}\ \varphi + \mathsf{floorlog}\ 2\ (\mathsf{size}\ \varphi) + 4).$$

Exporting code for the LTL part needs only minor adjustments through code lemmas, e.g. we instantiate $\sim_p$ with code provided by [35]. For the parts related to automata we rely on the code export feature of the automata library, see Section 3.4. Notice that Theorem 25

refers to the potentially infinite alphabet UNIV. Choosing UNIV as the alphabet simplified the proofs leading up to the result, but potentially infinite alphabets make an exploration using depth-first search using a naive enumeration of letters impossible. Consequently, we restrict the alphabet to a finite set for the code export by only considering atomic propositions occurring in φ. The resulting constant ltl-to-draei has the signature α ltl $\Rightarrow$ (α set, nat) draei which is then exported to Standard ML. The overall correctness theorem is as follows:

▶ **Theorem 27.**

language (draei-dra (ltl-to-draei φ)) = language (Pow (atoms φ)) φ.

Note that the constant language is only defined for DRAs with a transition function (dra) while we obtain from the translation a DRA with a list of transitions (draei). The constant draei-dra converts an automaton of type draei back to one of type dra.

In the final tool, we combine the function ltl-to-draei with an unverified LTL parser and an unverified serialisation to the Hanoi Omega Automata format [3], a text-based format for representing ω-automata. It is then compiled with `mlton` or `polyc` using the build scripts included in the formalisation [39].

▶ **Example 28.** The following command translates the formula $\mathbf{FG}a$ to a DRA in HOA format and then, using `autfilt` from Spot [16], prints it in the `dot`-format. The result gets rendered by `dot` and is written to a PDF file.

```
./ltl_to_dra "F G a" | autfilt --dot --merge-transitions | dot -Tpdf -O
```

6 Concluding Remarks

The formalisation of the "Master Theorem" itself did not pose major obstacles and did not require special care except for the mentioned techniques. However, the LTL entry [41] and dependencies are host to several LTL datatypes and matching lemmas and notation. This excessive amount of copy-pasting is due the inability to define fragments of datatypes, i.e., restrictions on the constructors used. While one could use **typedef** to carve out restricted types using a predicate, this new type misses the structure of the type we started with. Thus we choose in some cases to have separate datatypes connected by translations, while in other cases we used simple predicates to capture fragments. We think the addition of a mechanism addressing this issue – the definition of datatype fragments and the addition of necessary constants and proof automation – would be worthwhile, since we conjecture it would significantly reduce the size and complexity of LTL related theories.

There are several topics we want to investigate going forward: First, we also want to derive constructions for NBAs and LDBAs. Second, we plan to reduce the size of the generated automata by restricting the possible choices for the advice sets X and Y. Third, we want to provide implementations using better instantiations for the equivalence relation to further reduce the size of the computed automata. Fourth, provide constructions for DRA variants, e.g., transition-based or generalised acceptance. Fifth, while adding some of the Boolean operations, we realised that constructions for ω-automata could potentially be shared and consolidated in an intermediate abstraction.

References

1 Romain Aïssat, Frédéric Voisin, and Burkhart Wolff. Infeasible Paths Elimination by Symbolic Execution Techniques: Proof of Correctness and Preservation of Paths. *Archive of Formal Proofs*, 2016, 2016. URL: `https://www.isa-afp.org/entries/InfPathElimination.shtml`.

2 Tomás Babiak, Thomas Badie, Alexandre Duret-Lutz, Mojmír Křetínský, and Jan Strejcek. Compositional Approach to Suspension and Other Improvements to LTL Translation. In Ezio Bartocci and C. R. Ramakrishnan, editors, *Model Checking Software - 20th International Symposium, SPIN 2013, Stony Brook, NY, USA, July 8-9, 2013. Proceedings*, volume 7976 of *Lecture Notes in Computer Science*, pages 81–98. Springer, 2013. doi:10.1007/978-3-642-39176-7_6.

3 Tomás Babiak, Frantisek Blahoudek, Alexandre Duret-Lutz, Joachim Klein, Jan Křetínský, David Müller, David Parker, and Jan Strejcek. The Hanoi Omega-Automata Format. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 479–486. Springer, 2015. doi:10.1007/978-3-319-21690-4_31.

4 Tomás Babiak, Frantisek Blahoudek, Mojmír Křetínský, and Jan Strejcek. Effective Translation of LTL to Deterministic Rabin Automata: Beyond the (F, G)-Fragment. In Dang Van Hung and Mizuhito Ogawa, editors, *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, volume 8172 of *Lecture Notes in Computer Science*, pages 24–39. Springer, 2013. doi:10.1007/978-3-319-02444-8_4.

5 Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

6 Clemens Ballarin. Locales: A Module System for Mathematical Theories. *J. Autom. Reasoning*, 52(2):123–153, 2014. doi:10.1007/s10817-013-9284-7.

7 Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Truly Modular (Co)datatypes for 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 93–110. Springer, 2014. doi:10.1007/978-3-319-08970-6_7.

8 Maksym Bortin and Christoph Lüth. Structured Formal Development with Quotient Types in Isabelle/HOL. In Serge Autexier, Jacques Calmet, David Delahaye, Patrick D. F. Ion, Laurence Rideau, Renaud Rioboo, and Alan P. Sexton, editors, *Intelligent Computer Mathematics, 10th International Conference, AISC 2010, 17th Symposium, Calculemus 2010, and 9th International Conference, MKM 2010, Paris, France, July 5-10, 2010. Proceedings*, volume 6167 of *Lecture Notes in Computer Science*, pages 34–48. Springer, 2010. doi:10.1007/978-3-642-14128-7_5.

9 Julian Brunner. Büchi complementation. *Archive of Formal Proofs*, 2017, 2017. URL: https://www.isa-afp.org/entries/Buchi_Complementation.html.

10 Julian Brunner. Transition Systems and Automata. *Archive of Formal Proofs*, 2017, 2017. URL: https://www.isa-afp.org/entries/Transition_Systems_and_Automata.html.

11 Julian Brunner. Partial Order Reduction. *Archive of Formal Proofs*, 2018, 2018. URL: https://www.isa-afp.org/entries/Partial_Order_Reduction.html.

12 Julian Brunner and Peter Lammich. Formal Verification of an Executable LTL Model Checker with Partial Order Reduction. *J. Autom. Reasoning*, 60(1):3–21, 2018. doi:10.1007/s10817-017-9418-4.

13 Janusz A. Brzozowski. Derivatives of Regular Expressions. *J. ACM*, 11(4):481–494, 1964. doi:10.1145/321239.321249.

14 Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018. doi:10.1007/978-3-319-10575-8.

15 Costas Courcoubetis and Mihalis Yannakakis. The Complexity of Probabilistic Verification. *J. ACM*, 42(4):857–907, 1995. doi:10.1145/210332.210339.

16 Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 - A Framework for LTL and ω-Automata Manipulation. In Cyrille Artho, Axel Legay, and Doron Peled, editors, *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*, volume 9938 of *Lecture Notes in Computer Science*, pages 122–129, 2016. doi:10.1007/978-3-319-46520-3_8.

17 Javier Esparza, Jan Křeínský, and Salomon Sickert. From LTL to deterministic automata - A safraless compositional approach. *Formal Methods in System Design*, 49(3):219–271, 2016. doi:10.1007/s10703-016-0259-2.

18 Javier Esparza and Jan Křetínský. From LTL to Deterministic Automata: A Safraless Compositional Approach. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 192–208. Springer, 2014. doi:10.1007/978-3-319-08867-9_13.

19 Javier Esparza, Jan Křetínský, and Salomon Sickert. One Theorem to Rule Them All: A Unified Translation of LTL into ω-Automata. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 384–393. ACM, 2018. doi:10.1145/3209108.3209161.

20 Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A Fully Verified Executable LTL Model Checker. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 463–478. Springer, 2013. doi:10.1007/978-3-642-39799-8_31.

21 Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A Fully Verified Executable LTL Model Checker. *Archive of Formal Proofs*, 2014, 2014. URL: https://www.isa-afp.org/entries/CAVA_LTL_Modelchecker.shtml.

22 Paul Gastin and Denis Oddoux. Fast LTL to Büchi Automata Translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2001. doi:10.1007/3-540-44585-4_6.

23 Rob Gerth, Doron A. Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In Piotr Dembinski and Marek Sredniawa, editors, *Protocol Specification, Testing and Verification XV, Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification, Warsaw, Poland, June 1995*, volume 38 of *IFIP Conference Proceedings*, pages 3–18. Chapman & Hall, 1995.

24 Brian Huffman and Ondrej Kuncar. Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, volume 8307 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2013. doi:10.1007/978-3-319-03545-1_9.

25 Simon Jantsch and Michael Norrish. Verifying the LTL to Büchi Automata Translation via Very Weak Alternating Automata. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer Science*, pages 306–323. Springer, 2018. doi:10.1007/978-3-319-94821-8_18.

26 Peter Lammich. Refinement for Monadic Programs. *Archive of Formal Proofs*, 2012, 2012. URL: https://www.isa-afp.org/entries/Refine_Monadic.shtml.

27 Peter Lammich. Automatic Data Refinement. *Archive of Formal Proofs*, 2013, 2013. URL: https://www.isa-afp.org/entries/Automatic_Refinement.shtml.

28 Peter Lammich. The CAVA Automata Library. *Archive of Formal Proofs*, 2014, 2014. URL: https://www.isa-afp.org/entries/CAVA_Automata.shtml.

29 Peter Lammich. Verified Efficient Implementation of Gabow's Strongly Connected Component Algorithm. 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 325–340. Springer, 2014. doi:10.1007/978-3-319-08970-6_21.

30 Peter Lammich. Verified Efficient Implementation of Gabow's Strongly Connected Components Algorithm. *Archive of Formal Proofs*, 2014, 2014. URL: https://www.isa-afp.org/entries/Gabow_SCC.shtml.

31 Peter Lammich and Markus Müller-Olm. Formalization of Conflict Analysis of Programs with Procedures, Thread Creation, and Monitors. *Archive of Formal Proofs*, 2007, 2007. URL: https://www.isa-afp.org/entries/Program-Conflict-Analysis.shtml.

32 Peter Lammich and René Neumann. A Framework for Verifying Depth-First Search Algorithms. In Xavier Leroy and Alwen Tiu, editors, *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*, pages 137–146. ACM, 2015. doi:10.1145/2676724.2693165.

33 Peter Lammich and René Neumann. A Framework for Verifying Depth-First Search Algorithms. *Archive of Formal Proofs*, 2016, 2016. URL: https://www.isa-afp.org/entries/DFS_Framework.shtml.

34 Stephan Merz. Weak Alternating Automata in Isabelle/HOL. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs 2000, Portland, Oregon, USA, August 14-18, 2000, Proceedings*, volume 1869 of *Lecture Notes in Computer Science*, pages 424–441. Springer, 2000. doi:10.1007/3-540-44659-1_26.

35 Tobias Nipkow. Boolean Expression Checkers. *Archive of Formal Proofs*, 2014, 2014. URL: https://www.isa-afp.org/entries/Boolean_Expression_Checkers.shtml.

36 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.

37 Tobias Nipkow and Dmitriy Traytel. Unified Decision Procedures for Regular Expression Equivalence. 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 450–466. Springer, 2014. doi:10.1007/978-3-319-08970-6_29.

38 Alexander Schimpf, Stephan Merz, and Jan-Georg Smaus. Construction of Büchi Automata for LTL Model Checking Verified in Isabelle/HOL. 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 424–439. Springer, 2009. doi:10.1007/978-3-642-03359-9_29.

39 Benedikt Seidl and Salomon Sickert. A Compositional and Unified Translation of LTL into ω-Automata. *Archive of Formal Proofs*, 2019, 2019. URL: https://isa-afp.org/entries/LTL_Master_Theorem.html.

40 Salomon Sickert. Converting Linear Temporal Logic to Deterministic (Generalised) Rabin Automata. *Archive of Formal Proofs*, 2015, 2015. URL: https://www.isa-afp.org/entries/LTL_to_DRA.shtml.

41 Salomon Sickert. Linear Temporal Logic. *Archive of Formal Proofs*, 2016, 2016. URL: https://www.isa-afp.org/entries/LTL.shtml.

42 Salomon Sickert. *A Unified Translation of Linear Temporal Logic to ω-Automata*. PhD thesis, Technical University Munich, Germany, 2019.

43 Salomon Sickert, Javier Esparza, Stefan Jaax, and Jan Křetínský. Limit-Deterministic Büchi Automata for Linear Temporal Logic. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 312–332. Springer, 2016. doi:10.1007/978-3-319-41540-6_17.

44 Moshe Y. Vardi. Automatic Verification of Probabilistic Concurrent Finite-State Programs. In *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985*, pages 327–338. IEEE Computer Society, 1985. doi:10.1109/SFCS.1985.12.

Formalizing Computability Theory via Partial Recursive Functions

Mario Carneiro

Carnegie Mellon University, Pittsburgh, PA, USA
mcarneir@andrew.cmu.edu

Abstract

We present an extension to the `mathlib` library of the Lean theorem prover formalizing the foundations of computability theory. We use primitive recursive functions and partial recursive functions as the main objects of study, and we use a constructive encoding of partial functions such that they are executable when the programs in question provably halt. Main theorems include the construction of a universal partial recursive function and a proof of the undecidability of the halting problem. Type class inference provides a transparent way to supply Gödel numberings where needed and encapsulate the encoding details.

2012 ACM Subject Classification Theory of computation → Recursive functions; Theory of computation → Interactive proof systems

Keywords and phrases Lean, computability, halting problem, primitive recursion

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.12

Related Version A full version of the paper is available at `https://arxiv.org/abs/1810.08380`.

Supplement Material The formalization is a part of the `mathlib` Lean library at `https://github.com/leanprover-community/mathlib`, and a snapshot as of this publication is available at `http://github.com/digama0/mathlib-ITP2019`.

Funding This material is based upon work supported by AFOSR grant FA9550-18-1-0120 and a grant from the Sloan Foundation.

Acknowledgements I would like to thank my advisor Jeremy Avigad for his support and encouragement, and for his reviews of early drafts of this work, as well as Yannick Forster, Rob Y. Lewis, and the anonymous reviewers for their many helpful comments.

1 Introduction

Computability theory is the study of the limitations of computers, first brought into focus in the 1930s by Alan Turing by his discoveries on the existence of universal Turing machines and the unsolvability of the halting problem [16], and Alonso Church with the λ-calculus as a model of computation [3]. Together with Kleene's μ-recursive functions [10], that these all give the same collection of "computable functions" gave credence to the thesis [3] that this is the "right" notion of computation, and that all others are equivalent in power. Today, this work lies at the basis of programming language semantics and the mathematical analysis of computers.

Like many areas of mathematics, computability theory remains somewhat "formally ambiguous" about its foundations, in the sense that most theorems and proofs can be stated with respect to a number of different concretizations of the ideas in play. This can be considered a feature of informal mathematics, because it allows us to focus on the essential aspects without getting caught up in details which are more an artifact of the encoding than aspects that are relevant to the theory itself, but it is one of the harder things to deal with as a formalizer, because definitions must be made relative to *some* encoding, and this colors the rest of the development.

In computability theory, we have three or four competing formulations of "computable," which are all equivalent, but each present their own view on the concept. As a pragmatic matter, Turing machines have become the de facto standard formulation of computable functions, but they are also notorious for requiring a lot of tedious encoding in order to get the theory off the ground, to the extent that the term "Turing tarpit" is now used for languages in which "everything is possible but nothing of interest is easy." [14] Asperti and Riccioti [1] have formalized the construction of a universal Turing machine in Matita, but the encoding details make the process long and arduous. Norrish [13] uses the lambda calculus in HOL4, which is cleaner but still requires some complications with respect to the handling of partiality and type dependence.

Instead, we build our theory on Kleene's theory of μ-recursive functions. In this theory, we have a collection of functions $\mathbb{N}^k \to \mathbb{N}$, in which we can perform basic operations on $\mathbb{N}$, as well as recursive constructions on the natural number arguments. This produces the primitive recursive functions, and adding an unbounded recursion operator $\mu x.P(x)$ gives these functions the same expressive power as Turing-computable functions. We hope to show that the "main result" here, the existence of a universal machine, is easiest to achieve over the partial recursive functions, avoiding the complications of explicit substitution in the λ-calculus and encoding tricks in Turing Machines, and moreover that the usage of typeclasses for Gödel numbering provides a rich and flexible language for discussing computability over arbitrary types.

This theory has been developed in the Lean theorem prover, a relatively young proof assistant based on dependent type theory with inductive types, written primarily by Leonardo de Moura at Microsoft Research [4]. The full development is available in the `mathlib` standard library (see the Supplemental Material). In Section 2 we describe our extensible approach to Gödel numbering, in Section 3 we look at primitive recursive functions, extended to partial recursive functions in Section 4. Section 5 deals with the universal partial recursive function and its properties, including its application to unsolvability of the halting problem.

2 Encodable sets

As mentioned in the introduction, we would like to support some level of formal ambiguity when encoding problems, such as defining languages as subsets of $\mathbb{N}$ vs. subsets of $\{0,1\}^*$, or even Σ^* where Σ is some finite or countable alphabet. Similarly, we would like to talk about primitive recursive functions of type $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, or the partial recursive function $\mathsf{eval} : \mathsf{code} \times \mathbb{N} \nrightarrow \mathbb{N}$ that evaluates a partial function specified by a code (see Section 5).

Unfortunately it is not enough just to know that these types are countable. While the exact bijection to $\mathbb{N}$ is not so important, it is important that we not use one bijection in a proof and a different bijection in the next proof, because these differ by an automorphism of $\mathbb{N}$ which may not be computable. (For example, if we encode the halting Turing machines as even numbers and the non-halting ones as odd numbers, and then the halting problem becomes trivial.) In complexity theory it becomes even more important that these bijections are "simple" and do not smuggle in any additional computational power.

To support these uses, we make use of Lean's typeclass resolution mechanism, which is a way of inferring structure on types in a syntax-directed way. The major advantage of this approach is that it allows us to fix a uniform encoding that we can then apply to all types constructed from a few basic building blocks, which avoids the multiple encoding problem, and still lets us use the types we would like to (or even construct new types like code whose explicit structure reflects the inductive construction of partial recursive functions, rather than the encoding details).

	0	1	2	3	...
0	0	1	4	9	
1	2	3	5	10	
2	6	7	8	11	
3	12	13	14	15	
⋮					⋱

Figure 1 The pairing function `mkpair a b = if a < b then b*b + a else a*a + a + b`.

```
class encodable (α : Type u) :=
(encode : α → nat)
(decode : nat → option α)
(encodek : ∀ a, decode (encode a) = some a)

variables {α β} [encodable α] [encodable β]
def encode_sum : α ⊕ β → ℕ
| (inl a) := 2 * encode a
| (inr b) := 2 * encode b + 1

def encode_prod : α × β → ℕ
| (a, b) := mkpair (encode a) (encode b)

def encode_option : option α → ℕ
| none     := 0
| (some a) := succ (encode a)
```

Figure 2 The encodable typeclass, and some example definitions of the encoding functions for the disjoint sum and product operators on types. (The corresponding decode functions are omitted.)

At the core of this is the function mkpair : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$, and its inverse unpair : $\mathbb{N} \to \mathbb{N} \times \mathbb{N}$ forming a bijection (see Figure 1). There is very little we need about these functions except their definability, and that mkpair and the two components of unpair are primitive recursive.

We say that a type α is *encodable* if we have a function encode : $\alpha \to \mathbb{N}$, and a partial inverse decode : $\mathbb{N} \to$ option α which correctly decodes any value in the image of encode. Here option α is the type consisting of the elements some a for $a : \alpha$, and an extra element none representing failure or undefinedness. If the decode function happens to be total (that is, never returns none), then α is called *denumerable*. Importantly, these notions are "data" in the sense that they impose additional structure on the type – there are nonequivalent ways for a type to be encodable, and we will want these properties to be inferred in a consistent way. (This definition does not originate with us; Lean has had the encodable typeclass almost since the beginning, and MathComp has a similar class, called Countable.)

Classically, an encodable instance on α is just an injection to $\mathbb{N}$, and a denumerable instance is just a bijection to $\mathbb{N}$. But constructively these are not equivalent, and since these notions lie in the executable fragment of Lean (they don't use any classical axioms), one can actually run these encoding functions on concrete values of the types, i.e. we can evaluate encode (some $(2, 3)$) $= 12$.

```
inductive primrec : (ℕ → ℕ) → Prop
| zero : primrec (λ n, 0)
| succ : primrec succ
| left : primrec (λ n, fst (unpair n))
| right : primrec (λ n, snd (unpair n))
| pair {f g} : primrec f → primrec g →
  primrec (λ n, mkpair (f n) (g n))
| comp {f g} : primrec f → primrec g →
  primrec (f ∘ g)
| prec {f g} : primrec f → primrec g →
  primrec (unpaired (λ z n, nat.rec_on n (f z)
     (λ y IH, g (mkpair z (mkpair y IH)))))
```

Figure 3 The definition of primitive recursive on $\mathbb{N}$ in Lean. The unpaired function turns a function $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ into $\mathbb{N} \to \mathbb{N}$ by composing with unpair, and nat.rec_on : $\mathbb{N} \to \alpha \to (\mathbb{N} \to \alpha \to \alpha) \to \alpha$ is Lean's built-in recursor for $\mathbb{N}$.

3 Primitive recursive functions

The traditional definition of primitive recursive functions looks something like this:

▶ **Definition 1.** *The primitive recursive functions are the least subset of functions* $\mathbb{N}^k \to \mathbb{N}$ *satisfying the following conditions:*

- *The function* $n \mapsto 0$ *is prim. rec.*
- *The function* $n \mapsto n + 1$ *is prim. rec.*
- *The function* $(n_0, \ldots, n_{k-1}) \mapsto n_i$ *is prim. rec. for each* $0 \leq i < k$.
- *If* $f : \mathbb{N}^k \to \mathbb{N}$ *and* $g_i : \mathbb{N}^m \to \mathbb{N}$ *for* $i \leq k$ *are prim. rec., then so is the* n*-way composition* $v \mapsto f(g_0(v), \ldots, g_{k-1}(v))$.
- *If* $f : \mathbb{N}^m \to \mathbb{N}$ *and* $g : \mathbb{N}^{m+2} \to \mathbb{N}$ *are prim. rec., then the function* $h : \mathbb{N}^{m+1} \to \mathbb{N}$ *defined by*

$$h(\vec{z}, 0) = f(\vec{z})$$
$$h(\vec{z}, n+1) = g(\vec{z}, n, h(\vec{z}, n))$$

  is also prim. rec.

CIC is quite good at expressing these kinds of constructions as inductively defined predicates. See Figure 3 for the definition that appears in Lean. But there is an important difference in this formulation: rather than dealing with n-ary functions, we utilize the pairing function on $\mathbb{N}$ to write everything as a function $\mathbb{N} \to \mathbb{N}$ with only one argument. This drastically simplifies the composition rule to just the usual function composition, and in the primitive recursion rule we need only one auxiliary parameter $z : \mathbb{N}$ rather than $\vec{z} : \mathbb{N}^m$. Then the projection functions are replaced with the left and right cases for the components of unpair : $\mathbb{N} \to \mathbb{N} \times \mathbb{N}$, and in order to express composition with higher arity functions, we need the pair constructor to explicitly form the map $x \mapsto (f\ x, g\ x)$. (See Section 3.1 if you think this definition is a cheat.)

Now that we have a definition of "primitive recursive" that works for functions on $\mathbb{N}$, we would like to extend it to other types using the encodable mechanism discussed in Section 2. There is a problem though, because given an arbitrary encodable instance we can combine the decode : $\mathbb{N} \to$ option α with the function encode : option $\alpha \to \mathbb{N}$ defined on option α

induced by this encodable instance to form a new function encode ∘ decode : ℕ → ℕ, which may or may not be primitive recursive. If it is not, then it brings new power to the primitive recursive functions and so it is not a pure translation of primrec to other types. To resolve this, we define primcodable α to mean exactly that α has an encodable instance for which this composition is primitive recursive. All of the encodable constructions we have discussed (indeed, all those defined in Lean) are primcodable, so this is not a severe restriction.

Now we can say that a function between arbitrary primcodable types is primitive recursive if when we pass f through the encode and decode functions we get a primitive recursive function on ℕ:

```
def primrec {α β} [primcodable α] [primcodable β] (f : α → β) : Prop :=
nat.primrec (λ n, encode (option.map f (decode α n)))
```

Note. The function option.map lifts f to a function on option types before applying it to decode. The result has type option β, which has an encode function because β does.

Now we are in a position to recover the textbook definition of primitive recursive, because $\mathbb{N}^k$ is primcodable, so we have the language to say that $f : \mathbb{N}^k \to \mathbb{N}$ is primitive recursive, and indeed this is equivalent to Definition 1.

But we can now say much more: the some : $\alpha \to$ option α function is primitive recursive because it is just encoded as succ. The constant function $\lambda a.b : \alpha \to \beta$ is primitive recursive because it encodes to some constant function (composed with a function that filters out values not in the domain α). The composition of prim. rec. functions on arbitrary types is prim. rec. The pair of primitive recursive functions $\lambda a.\,(f\ a, g\ a)$, where $f : \alpha \to \beta$ and $g : \alpha \to \gamma$, is primitive recursive.

Indeed all the usual basic operations on inductive types like sum, prod, and option are primitive recursive. We define convenient syntax primrec$_2$ for prim. rec. binary functions $\alpha \to \beta \to \gamma$ (a common case), expressed by uncurrying to $\alpha \times \beta \to \gamma$, and primrec_pred for primitive recursive predicates $\alpha \to$ Prop, which are decidable predicates which are primitive recursive when coerced to bool (which is encodable).

The big caveat comes in theorems like the following:

> If α and β are primcodable types and $f : \alpha \to \beta$ and $g : \alpha \to \mathbb{N} \to \beta \to \beta$ are prim. rec., then the function $h : \alpha \to \mathbb{N} \to \beta$ defined by
>
> $$\begin{aligned} h\ a\ 0 &= f\ a \\ h\ a\ (n+1) &= g\ a\ n\ (h\ a\ n) \end{aligned}$$
>
> is also prim. rec.

This is of course just the generalization of the primitive recursion clause to arbitrary types, but it requires that the target type be primcodable, which means in particular that it is countable, so we cannot define an object of function type by recursion. (The universal partial recursive function will give us a way to get around this later.) But this is in some sense "working as intended," since this is exactly why the Ackermann function

$$\begin{aligned} A(0, n) &= n + 1 \\ A(m+1, 0) &= A(m, 1) \\ A(m+1, n+1) &= A(m, A(m+1, n)) \end{aligned}$$

is not primitive recursive.

Another restriction placed on us relative to Lean's built-in notion of primitive recursion on $\mathbb{N}$ is that that while nat.rec_on has a dependent type, we have no mechanism for supporting dependent types via encodable. We follow the tradition of HOL based provers here and encode dependencies using option types so we can fail on a garbage input. However, it is possible to support a dependent family via a separate typeclass. For example we could define primcodable$_2$ F, where $F : \alpha \to$ Type and α is encodable, to mean that $\Pi a,$ encodable $(F\ a)$, and moreover this family of encode/decode functions is prim. rec. jointly in both arguments. In the end we did not pursue this because of the added complexity and lack of compelling use cases.

One other primcodable type we have not yet discussed is list α, the type of finite lists of values of type α. The encode and decode functions are defined recursively via the bijection list $\alpha \simeq$ option $(\alpha \times$ list $\alpha)$. (Note that this is not a particularly good encoding for complexity theory, as it grows super-exponentially in the length of the list.) Even without using this instance, we can prove that any function $f : \alpha \to \beta$ is prim. rec. when α is finite, by getting the elements of α as a list, and writing f as the composition of an index lookup of a_i in $[a_0, \ldots, a_{n-1}]$ and the ith element function in $[f\ a_0, \ldots, f\ a_{n-1}]$ to map a_i to $f\ a_i$.

The proof that primcodable (list α) is a bit delicate. The definition of the encode/decode functions in Lean is a well-founded recursion, but to show it is primitive recursive we must construct the function without any higher-order features. First, we prove that the foldl : $(\alpha \to \beta \to \alpha) \to \alpha \to$ list $\beta \to \alpha$ function is prim. rec. when its arguments are. To do this, given $f : \alpha \to \beta \to \alpha$, we construct an accumulator $\alpha \times$ list β with the initial inputs, and then repeatedly transform it so that $(a, []) \mapsto (a, [])$ and $(a, b :: l) \mapsto (f\ a\ b, l)$. Since the encoding scheme satisfies encode $l \geq$ length l for all lists l, if we iterate this map encode l times, we exhaust the input list and the accumulator will contain the desired result. We can then use foldl to define reverse, and combine them to define foldr, which is what we need to define the primcodable function for list α.

Complicating matters, we needed a primcodable instance for primcodable (list α) to state the original theorem that foldl is prim.rec., so we have a circularity. To resolve this, we use list $\mathbb{N}$ as a bootstrap, which is trivially primcodable because it is denumerable.

Once we allow the list itself to be an input, we get some more interesting possibilities. In particular, the function list.nth : list $\alpha \to \mathbb{N} \to$ option α, which gets an element from a list by index (or returns none if the index is out of bounds), is primitive recursive, and this fact expresses an equivalent of Gödel's sequence number theorem [8] (for a different encoding than Gödel's original encoding). From this we can prove the following "strong recursion" theorem:

```
theorem nat_strong_rec
  (f : α → ℕ → σ)
  {g : α → list σ → option σ}
  (hg : primrec₂ g)
  (H : ∀ a n, g a (map (f a) (range n)) = some (f a n)) :
  primrec₂ f
```

Ignoring the parameter a, the main hypothesis says essentially that $f(n) = g(f \restriction [0, \ldots, n-1])$, where the first n values of f have been written in a list (and the length of the list tells g what value of f we are constructing). The reason g has optional return value is to allow for it to fail when the input is not valid.

Once we have lists, the dependent type vector $\alpha\ n$ is just a subtype of list α, so it has an easy primcodable instance, and most of the vector functions follow from their list counterparts. Similarly for functions fin $n \to \alpha$, which are isomorphic to vector $\alpha\ n$.

```
def unpair (n : ℕ) : ℕ × ℕ :=
let s := sqrt n in
if n - s*s < s then (n - s*s, s) else (s, n - s*s - s)
```

Figure 4 The function unpair : $\mathbb{N} \to \mathbb{N} \times \mathbb{N}$. (Here sqrt : $\mathbb{N} \to \mathbb{N}$ is actually the function $n \mapsto \lfloor \sqrt{n} \rfloor$.)

3.1 The textbook definition

Now that we have a proper theory, we can return to the question of how to show equivalence to Definition 1. We do this by defining nat.primrec′ : $\forall n, (\mathsf{vector}\ \mathbb{N}\ n \to \mathbb{N}) \to \mathsf{Prop}$ with only 5 clauses matching Definition 1. It is easy to show at this point that primrec′ implies primrec, since all of the functions appearing in Definition 1 are known to be primitive recursive. For the converse, most of the clauses are easy, but our earlier cheat was to axiomatize that mkpair and unpair are primitive recursive, even though the definition involves addition, multiplication and case analysis in mkpair and even square root in the inverse function (see Figure 4). So we must show that all these operations are primitive recursive by the textbook definition. The square root case is not as difficult as it may sound; since it grows by at most 1 at each step we can define it by primitive recursion as

$$\begin{aligned} \lfloor \sqrt{0} \rfloor &= 0 \\ \lfloor \sqrt{n+1} \rfloor &= \text{if } n+1 < (y+1)^2 \text{ then } y \text{ else } y+1 \\ &\qquad \text{where } y = \lfloor \sqrt{n} \rfloor. \end{aligned}$$

This alternate basis for primrec is useful for reductions, for example, to show that some other basis for computation like Turing machines can simulate every primitive recursive function.

4 Partial recursive functions

The partial recursive functions are an extension of primitive recursive functions by adding an operator $\mu n.\ p(n)$, where $p : \mathbb{N} \to \mathsf{bool}$ is a predicate, which denotes the least value of n such that $p(n)$ is true. Intuitively, this value is found by starting at 0 and testing ever larger values until a satisfying instance is found. This function is not always defined, in the sense that even when all the inputs are well typed it may not return a value – it can result in an "infinite loop."

Before we tackle the partial recursive functions we must understand partiality itself, and in particular how to represent unbounded computation, computably, in a proof assistant that can only represent terminating computations. As Lean is based on dependent type theory, which is strongly normalizing, all expression evaluation terminates, and so the problem is prima facie unsolvable – we may as well turn to functional relations as a representation. However, as we shall see, it is actually possible with no additional modifications to CIC or extra axioms.

4.1 The partiality monad

We have already discussed the option α type for representing a possible failure state, but nontermination is a slightly different kind of "failure" in that the program is not able to tell that it has failed while executing, and this difference makes itself known in the type system.

To address this distinction, we introduce the part α type:

```
def part (α : Type*) := Σ p : Prop, (p → α)
```

That is, an element p : part α is a dependent pair of a proposition p_1 and a function $p_2 : p_1 \to \alpha$ from proofs of p_1 to α. A value of type part α is a nondecidable optional value, in the sense that there is not necessarily a decision procedure for determining if the part α contains a value, but if it does then you can extract the value using the function component. This type has a monad structure, as follows:

$$\mathsf{pure} : \alpha \to \mathsf{part}\ \alpha$$
$$\mathsf{pure}\ a = \langle \mathsf{true}, \lambda\_.\ a \rangle$$
$$\mathsf{bind} : \mathsf{part}\ \alpha \to (\alpha \to \mathsf{part}\ \beta) \to \mathsf{part}\ \beta$$
$$\mathsf{bind}\ \langle p, f \rangle\ g = \langle (\exists h : p, (g\ (f\ h))_1), (\lambda h.\ (g\ (f\ h_1))_2\ h_2) \rangle$$

Also, there is an element $\bot = \langle \mathsf{false}, \mathsf{exfalso} \rangle$: part α representing an undefined value. We can map option $\alpha \to$ part α by sending some a to pure a and none to $\bot$, and assuming the law of excluded middle in Type we can also define an inverse map and show option $\alpha \simeq$ part α, but this breaks the computational interpretation of part α.

The definition of bind, also written in Haskell style as the infix operator >>=, is slightly intricate but is "exactly what you would expect" in terms of its behavior. Given a partial value p : part α and a function $f : \alpha \to$ part β, the resulting partial value p >>= f : part β is defined when p is defined to be some $a : \alpha$, and $f\ a$ is defined, in which case it evaluates to $f\ a$.

It is convenient to abstract from the definition to a relational version, where $a \in p$ means $\exists h : p_1, p_2\ h = a$ – that is, $a \in p$ says that p is defined and equal to a. (This relation is functional because of proof irrelevance.) With this definition the bind operator can be much more easily expressed by the theorem

$$b \in p \gg\!= f \leftrightarrow \exists a \in p, b \in f\ a$$

which is shared with many other collection-based monad structures. Also, like every other monad there is a map operator, written <$>, which applies a pure function to a partial value:

$$\mathsf{map} : (\alpha \to \beta) \to \mathsf{part}\ \alpha \to \mathsf{part}\ \beta$$
$$f <\$> p = \langle p_1, f \circ p_2 \rangle$$

Because they come up often, we will use the notation $\alpha \nrightarrow \beta = \alpha \to$ part β for the type of all partial functions from α to β.

One important function that is (constructively) definable on this type is fix, which has the following properties:

$$\mathsf{fix}\ (f : \alpha \nrightarrow \beta \oplus \alpha) : \alpha \nrightarrow \beta$$
$$b \in \mathsf{fix}\ f\ a \leftrightarrow \mathsf{inl}\ b \in f\ a \vee \exists a', \mathsf{inr}\ a' \in f\ a \wedge b \in \mathsf{fix}\ f\ a'$$

Given an input a, it evaluates f to get either inl b or inr a'. In the first case it returns b, and in the second case it starts over with the value a'. The function fix f is defined when this process eventually terminates with a value, if we assume this then we can construct the value that fix f returns. So even though Lean's type theory does not permit unbounded recursion, by working in this partiality monad we get computable unbounded recursion.

```
inductive partrec : (ℕ →. ℕ) → Prop
| zero : partrec (pure 0)
| succ : partrec succ
| left : partrec (λ n, fst (unpair n))
| right : partrec (λ n, snd (unpair n))
| pair {f g} : partrec f → partrec g →
  partrec (λ n, f n >>= λ a, g n >>= λ b, pure (mkpair a b))
| comp {f g} : partrec f → partrec g →
  partrec (λ n, g n >>= f)
| prec {f g} : partrec f → partrec g →
  partrec (unpaired (λ a n, nat.rec_on n (f a)
    (λ y IH, IH >>= λ i,
      g (mkpair a (mkpair y i)))))
| find {f} : partrec f → partrec (λ a,
  find (λ n, (λ m, m = 0) <$> f (mkpair a n)))
```

Figure 5 The definition of partial recursive on $\mathbb{N}$ in Lean.

The minimization operator $\mathsf{find}\ p = \mu n.\, p(n)$, which finds the smallest value satisfying the (partial) boolean predicate p can be defined in terms of fix as follows:

$$\mathsf{find} : (\mathbb{N} \nrightarrow \mathsf{bool}) \nrightarrow \mathbb{N}$$
$$\mathsf{find}\ p = \mathsf{fix}\ (\lambda n.\ \text{if } p\ n \text{ then } \mathsf{inl}\ n \text{ else } \mathsf{inr}(n+1))\ 0$$

As an aside, we note that while this monad supports many of the operations one expects on partial recursive functions, one thing it does not support is parallel computation. That is, we would like to have a nondeterministic choice function $\mathsf{<|>} : \mathsf{part}\ \alpha \to \mathsf{part}\ \alpha \to \mathsf{part}\ \alpha$ such that $p\ \mathsf{<|>}\ q$ is defined if either p or q is defined (with value arbitrarily chosen from the two). This is possible for partial recursive functions, but it is not constructively definable for part. For this, we must restrict the propositions to be *semidecidable* [2], which means essentially that they are a Σ_1 proposition, that is, a proposition of the form $\exists n.\ f(n) = \mathsf{true}$ for some $f : \mathbb{N} \to \mathsf{bool}$. Every partial recursive function is semidecidable as a consequence of the eval_k function (see Section 5.2).

4.2 partrec and computable

The definition nat.partrec is given in Figure 5. The first 7 cases are almost the same as those of primrec, except that we must now worry about partiality in all the operations that build functions. So for example `λ n, f n >>= λ a, g n >>= λ b, pure (mkpair a b)` is the function $n \mapsto (f\ n, g\ n)$ except that if the computation of either $f\ n$ or $g\ n$ fails to return a value, then this is not defined. (In other words, this operation is "strict" in both arguments). Similarly, the composition is now expressed as `λ n, g n >>= f`, which says that $g\ n$ should be evaluated first, and if it is defined and equals a, then $f\ a$ is the resulting value.

The interesting case is the last one, which incorporates the find function on $\mathbb{N}$. Ignoring partiality, it says that $\lambda a.\, \mu n.\, f(a, n) = 0$ is partial recursive if f is. This is of course the source of the partiality – all the other constructors produce total functions from total functions but this can be partial if the function f is never zero.

Although this defines a class of partial functions, some of the functions happen to be total anyway, and we call a total partial-recursive function *computable*. It is an easy fact that every primitive recursive function is computable.

As before, we can compose with encode and decode to extend these definitions to any primcodable type. Although we could define an analogue of primcodable using computable functions instead of primitive recursive functions, since we want to stick to simple encodings (usually not just primitive recursive but polynomial time), and we already have encodings for all the important types, so primcodable is enough.

One aspect of this definition which is not obviously a problem until one works out all the details is the strictness of the prec constructor. In conventional notation, it says that if $f : \alpha \nrightarrow \beta$ and $g : \alpha \to \mathbb{N} \to \beta \nrightarrow \beta$ are partial recursive functions, then so is the function $h : \alpha \to \mathbb{N} \nrightarrow \beta$ defined by

$$\begin{aligned} h(a, 0) &= f(a) \\ h(a, n+1) &= g(a, n, h(a, n)). \end{aligned}$$

Importantly, $h(a, n+1)$ is only defined if $h(a, n)$ is defined and $g(a, n, h(a, n))$ is defined. It does not matter if g does not make use of the argument at all, for example if it is the first projection. This comes up in the definition of the lazy conditional $\mathsf{ifz}[f, g]$, defined when $f : \alpha \nrightarrow \beta$, $g : \alpha \nrightarrow \beta$ by:

$$\mathsf{ifz}[f, g] : \alpha \to \mathbb{N} \nrightarrow \beta$$
$$\mathsf{ifz}[f, g](a, n) = \begin{cases} f(a) & \text{if } n = 0 \\ g(a) & \text{if } n \neq 0 \end{cases},$$

where in particular $\mathsf{ifz}[f, g](a, 1) = g(a)$ regardless of whether $f(a)$ is defined. This is the basis of "if statements" that resemble execution paths in a computer – we need a way to choose which subcomputation to perform, without needing to evaluate both. The usual way of implementing ifz is to use primitive recursion on the argument n, using f in the zero case and $g \circ \pi_1$ in the successor case. But because of the strictness constraint, this will result in $\mathsf{ifz}[\bot, g](a, 1) = (g \circ \pi_1)(a, 0, f(a)) = \bot$ because $f(a) = \bot$, rather than the desired result $g(a)$. In fact, we won't have the tools to solve this problem until Section 5.3.

5 Universality

5.1 Codes for functions

Because partrec is an inductive predicate, we can read off a corresponding data type of syntactic representations witnessing that a function $\mathbb{N} \nrightarrow \mathbb{N}$ is partial recursive:

```
inductive code : Type
| zero : code
| succ : code
| left : code
| right : code
| pair : code → code → code
| comp : code → code → code
| prec : code → code → code
| find' : code → code
```

We can define the semantics of a code via an "evaluation" function that takes a code and an input value in $\mathbb{N}$ and produces a partial $\mathbb{N}$ value.

```
def eval : code → ℕ →. ℕ
| zero         := pure 0
| succ         := succ
| left         := λ n, n.unpair.1
| right        := λ n, n.unpair.2
| (pair cf cg) := λ n,
  eval cf n >>= λ a, eval cg n >>= λ b, pure (mkpair a b)
| (comp cf cg) := λ n, eval cg n >>= eval cf
| (prec cf cg) := unpaired (λ a n,
  nat.rec_on n (eval cf a) (λ y IH, IH >>= λ i,
    eval cg (mkpair a (mkpair y i))))
| (find' cf)   := unpaired (λ a m, (λ i, i + m) <$>
  find (λ n, (λ m, m = 0) <$> eval cf (mkpair a (n + m))))
```

Then it is a simple consequence of the definition that f is partial recursive iff there exists a code $\hat{f}$ such that $f = \mathsf{eval}\ \hat{f}$.

Note. The find' constructor is a slightly modified version of find which is easier to use in evaluation:

$$\mathsf{find}'\ f\ (a, m) = (\mu n.\ f(a, n + m) = 0) + m,$$

which can be expressed in terms of find as:

$$\begin{aligned} \mathsf{find}\ f\ a &= \mathsf{find}'\ f\ (a, 0) \\ \mathsf{find}'\ f\ (a, m) &= \mathsf{find}\ (\lambda x.\ f\ (x_1, x_2 + m))\ a + m \end{aligned}$$

So we can pretend that partrec was defined with a case for find' instead of find since it yields the same class of functions.

Now the key fact is that code is denumerable. Concretely, we can encode it using a combination of the tricks we used to encode sums, products and option types, that is,

$$\begin{aligned} \mathsf{encode}\,(\mathsf{zero}) &= 0 \\ \mathsf{encode}\,(\mathsf{succ}) &= 1 \\ \mathsf{encode}\,(\mathsf{left}) &= 2 \\ \mathsf{encode}\,(\mathsf{right}) &= 3 \\ \mathsf{encode}\,(\mathsf{pair}\ c_1\ c_2) &= 4 \cdot (\mathsf{encode}\ c_1, \mathsf{encode}\ c_2) + 4 \\ \mathsf{encode}\,(\mathsf{comp}\ c_1\ c_2) &= 4 \cdot (\mathsf{encode}\ c_1, \mathsf{encode}\ c_2) + 5 \\ \mathsf{encode}\,(\mathsf{prec}\ c_1\ c_2) &= 4 \cdot (\mathsf{encode}\ c_1, \mathsf{encode}\ c_2) + 6 \\ \mathsf{encode}\,(\mathsf{find}'\ c) &= 4 \cdot (\mathsf{encode}\ c) + 7 \end{aligned}$$

where (m, n) is the pairing function from Figure 1. (We could have used a more permissive encoding, but this has the advantage that it is a bijection to $\mathbb{N}$, which makes the proof that this is a primcodable type trivial.)

Having shown that the type is primcodable we can now start to show that functions *on codes* are primitive recursive. In particular, all the constructors are primitive recursive, the recursion principle preserves primitive recursiveness and computability (not partial recursiveness, because of the as-yet unresolved problem with ifz), and we can prove that these simple functions on codes are primitive recursive:

$\mathsf{const} : \mathbb{N} \to \mathsf{code}$

$\mathsf{eval}\ (\mathsf{const}\ a)\ n = a$

$\mathsf{curry} : \mathsf{code} \to \mathbb{N} \to \mathsf{code}$

$\mathsf{eval}\ (\mathsf{curry}\ c\ m)\ n = \mathsf{eval}\ c\ (m, n)$

In particular, the rather understated fact that curry is primitive recursive is a form of the *s-m-n* theorem of recursion theory.

5.2 Resource-bounded evaluation

We have one more component before the universality theorem. We define a "resource-bounded" version of eval, namely $\mathsf{eval}_k : \mathsf{code} \to \mathbb{N} \to \mathsf{option}\ \mathbb{N}$ where $k : \mathbb{N}$. (In the formal text it is called `evaln`.) This function is total – we have a definite failure condition this time, unlike eval itself, which can diverge. There are multiple ways to define this function; the important part is that if $\mathsf{eval}\ c\ n = \bot$ then $\mathsf{eval}_k\ c\ n = \mathsf{none}$ for all k, and if $\mathsf{eval}\ c\ n = a$ is defined then $\mathsf{eval}_k\ c\ n = \mathsf{some}\ a$ for some k. Furthermore, it is convenient to ensure that eval_k is monotonic in k, and the domain of eval_k is contained in $[0, k]$, that is, if $n > k$ then $\mathsf{eval}_k\ c\ n = \mathsf{none}$.

The Lean definition of `evaln` is given in Figure 6. The details of the definition are not so important, but it is interesting to note that our "fuel" k for the computation only needs to decrease when we don't change the program code in the recursive call, namely in the prec and find' cases, thanks to Lean's pattern matcher (which compiles this definition into one by nested structural recursion). (You may wonder why we cannot use the fact that n is decreasing in the prec case to prove termination, but this is because the function is not defined by recursion on n, it is by recursion on k at all $n \le k$ simultaneously.)

Because $\mathsf{eval}_k\ c : \mathbb{N} \to \mathsf{option}\ \mathbb{N}$ has finite domain $n \in [0, k]$ outside which it is none, we can encode the whole function as a single $\mathsf{list}\ (\mathsf{option}\ \mathbb{N})$. Thus we can pack the function into the type $\mathbb{N} \times \mathsf{code} \to \mathsf{list}\ (\mathsf{option}\ \mathbb{N})$, and define this by strong recursion (using the theorem `nat_strong_rec` mentioned in Section 3), since in every case of the recursion, either k decreases and c remains fixed, or c decreases and k remains fixed.

Thus $\mathsf{evaln} : \mathbb{N} \to \mathsf{code} \to \mathbb{N} \to \mathsf{option}\ \mathbb{N}$ is primitive recursive (jointly in all arguments), and since $\mathsf{eval}\ c\ n = \mathsf{eval}_{k'}\ c\ n$ where $k' = \mu k.\ (\mathsf{eval}_{k'}\ c\ n \neq \mathsf{none})$, this shows that eval is partial recursive. This is Kleene's normal form theorem (in a different language) – eval is a universal partial recursive function.

5.3 Applications

The fixed point theorems are an easy consequence of universality. These have all been formalized; the formalized theorem names are given in parentheses.

▶ **Theorem 2** (`fixed_point`). *If* $f : \mathsf{code} \to \mathsf{code}$ *is computable, then there exists some code* c *such that* $\mathsf{eval}(f\ c) = \mathsf{eval}\ c$.

```
def evaln : ∀ k : ℕ, code → ℕ → option ℕ
| 0 _ := λ n, none
| (k+1) zero := λ n, guard (n ≤ k) >> pure 0
| (k+1) succ := λ n, guard (n ≤ k) >> pure (succ n)
| (k+1) left := λ n, guard (n ≤ k) >> pure (fst (unpair n))
| (k+1) right := λ n, guard (n ≤ k) >> pure (snd (unpair n))
| (k+1) (pair cf cg) := λ n, guard (n ≤ k) >>
  evaln (k+1) cf n >>= λ a, evaln (k+1) cg n >>= λ b, pure (mkpair a b)
| (k+1) (comp cf cg) := λ n, guard (n ≤ k) >>
  evaln (k+1) cg n >>= λ x, evaln (k+1) cf x
| (k+1) (prec cf cg) := λ n, guard (n ≤ k) >>
  unpaired (λ a m, nat.rec_on m
    (evaln (k+1) cf a)
    (λ y, evaln k (prec cf cg) (mkpair a y) >>= λ i,
          evaln (k+1) cg (mkpair a (mkpair y i)))) n
| (k+1) (find' cf)  := λ n, guard (n ≤ k) >>
  unpaired (λ a m,
    evaln (k+1) cf (mkpair a m) >>= λ x,
    if x = 0 then pure m else
      evaln k (find' cf) (mkpair a (m+1))) n
```

Figure 6 The definition of resource-bounded evaluation of partial recursive functions in Lean. *Notation note:* The » operator is monad sequencing, i.e. $a \gg b = a \gg= \lambda\_.\ b$, and `guard` p : option unit is the function that returns some () if p is true and none if p is false. Together they ensure that `evaln k c n = none` unless $n \le k$.

Proof. Consider the function $g : \mathbb{N} \to \mathbb{N} \nrightarrow \mathbb{N}$ defined by $g\ x\ y = \mathsf{eval}\ (\mathsf{eval}\ x\ x)\ y$ (using $\mathsf{decode} : \mathbb{N} \to \mathsf{code}$ to use natural numbers as codes in eval). This function is clearly partial recursive, so let $g = \mathsf{eval}\ \hat{g}$. Now let $F : \mathbb{N} \to \mathsf{code}$ such that $F\ x = f\ (\mathsf{curry}\ \hat{g}\ x)$; then F is computable so let $F = \mathsf{eval}\ \hat{F}$. Then for $c = \mathsf{curry}\ \hat{g}\ \hat{F}$ we have:

$$\begin{aligned}
\mathsf{eval}\ (f\ c)\ n &= \mathsf{eval}\ (f\ (\mathsf{curry}\ \hat{g}\ \hat{F}))\ n \\
&= \mathsf{eval}\ (F\ \hat{F})\ n \\
&= \mathsf{eval}\ (\mathsf{eval}\ \hat{F}\ \hat{F})\ n \\
&= g\ \hat{F}\ n \\
&= \mathsf{eval}\ \hat{g}\ (\hat{F}, n) \\
&= \mathsf{eval}\ (\mathsf{curry}\ \hat{g}\ \hat{F})\ n \\
&= \mathsf{eval}\ c\ n.
\end{aligned}$$

◀

▶ **Theorem 3** (`fixed_point₂`). *If* $f : \mathsf{code} \to \mathbb{N} \nrightarrow \mathbb{N}$ *is partial recursive, then there exists some code* c *such that* $\mathsf{eval}\ c = f\ c$.

Proof. Let $f = \mathsf{eval}\ \hat{f}$, and apply Theorem 2 to $\mathsf{curry}\ \hat{f}$ to obtain a c such that $\mathsf{eval}\ (\mathsf{curry}\ \hat{f}\ c) = \mathsf{eval}\ c$. Then

$$\begin{aligned}
\mathsf{eval}\ c\ n &= \mathsf{eval}\ (\mathsf{curry}\ \hat{f}\ c)\ n \\
&= \mathsf{eval}\ \hat{f}\ (c, n) \\
&= f\ c\ n.
\end{aligned}$$

◀

We can also finally solve the ifz problem. If f and g are partial recursive functions, then letting $f = \mathsf{eval}\ \hat{f}$ and $g = \mathsf{eval}\ \hat{g}$, the function

$$c(n) = \begin{cases} \hat{f} & \text{if } n = 0 \\ \hat{g} & \text{if } n \neq 0 \end{cases}$$

is primitive recursive (since both branches are just numbers now instead of computations that may not halt), and $\mathsf{ifz}[f, g](a, n) = \mathsf{eval}\ c(n)\ a$. More generally, this implies that we can evaluate conditionals where the condition is a computable function and the branches are partial functions. We can also construct a nondeterministic choice function:

▶ **Theorem 4** (`merge`). *If $f, g : \alpha \nrightarrow \beta$ are partial recursive functions, then there exists a function $h : \mathbb{N} \nrightarrow \mathbb{N}$ such that $h(a)$ is defined iff either $f(a)$ or $g(a)$ is defined, and if $x \in h(a)$ then $x \in f(a)$ or $x \in g(a)$.*

Proof. It is easy to reduce to the case where $f, g : \mathbb{N} \nrightarrow \mathbb{N}$. Let $f = \mathsf{eval}\ \hat{f}$ and $g = \mathsf{eval}\ \hat{g}$; then $h(n) = \mathsf{find}(\lambda k.\ \mathsf{eval}_k\ \hat{f}\ n\ \texttt{<|>}\ \mathsf{eval}_k\ \hat{g}\ n)$ works, where <|> is the alternative operator on option $\mathbb{N}$. ◀

A corollary is Post's theorem on the equivalence of computable and r.e. co-r.e. sets:

▶ **Theorem 5** (`computable_iff_re_compl_re`). *If $p : \alpha \to \mathsf{Prop}$ is a decidable predicate, then p is computable iff p is r.e. and $\lambda a.\ \neg p\ a$ is r.e.*

Proof. The forward direction is trivial. In the reverse direction, if $f, g : \alpha \nrightarrow \mathsf{unit}$ are chosen such that $f(a)$ is defined iff $p(a)$ and $g(a)$ is defined iff $\neg p(a)$, then by Theorem 4 there is a function $h : \alpha \nrightarrow \mathsf{bool}$ extending $\lambda a.\ f(a) \gg \mathsf{pure\ true}$ and $\lambda a.\ g(a) \gg \mathsf{pure\ false}$. This function has domain $\{a \mid p(a) \vee \neg p(a)\} = \alpha$ (because p is decidable) and is true when $p(a)$ is true and is false when $\neg p(a)$. Thus h is a computable indicator function for p. ◀

The assumption that p is decidable is not the tightest condition we could assert; it suffices p is stable, i.e. $\neg\neg p(a) \to p(a)$, or alternatively we could assume Markov's principle or LEM.

We conclude with Rice's theorem on the noncomputability of all nontrivial properties about computable functions:

▶ **Theorem 6** (`rice`). *Let $C \subseteq (\mathbb{N} \nrightarrow \mathbb{N})$ such that $\{c \mid \mathsf{eval}\ c \in C\}$ is computable. Then for any $f, g : \mathbb{N} \nrightarrow \mathbb{N}$, $f \in C$ implies $g \in C$ (so classically $C = \emptyset \vee C = \mathbb{N} \nrightarrow \mathbb{N}$).*

Proof. Apply Theorem 3 to the function $F\ c\ n =$ if $\mathsf{eval}\ c \in C$ then $g\ n$ else $f\ n$. to obtain a c such that $\mathsf{eval}\ c = F\ c$. (Note $\mathsf{eval}\ c \in C$ is decidable because it is computable.) Then if $\mathsf{eval}\ c \in C$, we have $F\ c\ n = g\ n$ for all n so $\mathsf{eval}\ c = F\ c = g$, hence $g \in C$. And if $\mathsf{eval}\ c \notin C$ then $\mathsf{eval}\ c = F\ c = f$ similarly which contradicts $f \in C$, $\mathsf{eval}\ c \notin C$. ◀

The undecidability of the halting problem is a trivial corollary:

▶ **Theorem 7** (`halting_problem`). *The set $\{c : \mathsf{code} \mid \mathsf{eval}\ c\ 0$ is defined$\}$ is not computable.*

Proof. Suppose it is; we can write it as $\{c \mid \mathsf{eval}\ c \in C\}$ where $C = \{f \mid f\ 0$ is defined$\}$, so applying Rice's theorem with $f = \lambda n.\ 0$ and $g = \lambda n.\ \bot$ we have a contradiction from $f \in C$ and $g \notin C$. ◀

File	Section	Line Count
`primrec`	Section 3	1338
`partrec`	Section 4	730
`partrec_code`	Section 5	918
`halting`	Section 5.3	354

Figure 7 Line counts (unadjusted) for the files in this formalization. Note that `primrec.lean` contains mostly endpoint theorems intended for presenting users with a convenient API for primitive recursion proofs.

6 Related Works

While this is the first formalization of computability theory in Lean, there are a variety of related formalizations in other theorem provers.

- Zammit (1997) [18] uses n-ary μ-recursive functions with an explicit big-step semantics. Although we believe we have reproduced all the theorems in this report and more, it should be noted that this predates all the others on this list by more than 10 years.
- Norrish achieves a substantial amount in [13], using the λ-calculus in HOL4, up to Rice's theorem and r.e. sets. The primary differences involve the differing model of computation and differences from working in a classical higher order logic system rather than a dependent type theory. (Lean is primarily focused on classical logic, but it permits working in intuitionistic logic, and there was no particular reason to assume LEM except in Theorem 5.)
- Asperti and Riccoti [1] have formalized the construction of a universal Turing machine in Matita, but do not go as far as the halting problem or recursively enumerable sets.
- "Mechanising turing machines and computability theory in Isabelle/HOL" by Xu, Zhang and Urban [17] builds from Turing machines, constructs a universal Turing machine, formalizes the halting problem, and relates them to abacus machines and recursive functions. But they acknowledge up front that formalizing TMs is a "daunting prospect," and their formalization is much longer (although direct comparisons are misleading at best).
- Forster and Smolka [7] formalize call-by-value λ-calculus in Coq, including Post's theorem and the halting problem, but they have a much greater focus on constructive mathematics and the exploration of choice principles such as Markov's principle. As Lean is not as focused on constructive type theory, we have instead chosen to focus on getting these core results with a minimum of fuss and with the most externally useful developments, so that they can perform well as an addition to Lean's standard library.
- In "Typing Total Recursive Functions in Coq" [11], Larchey-Wendling shows that all total recursive functions have function witnesses in Coq. From the point of view of our paper, at least concerning total recursive functions in the sense used in computability theory, this is a consequence of the definition - a computable function has a function witness by definition, as it is a predicate on functions. Similarly, we can evaluate a partial recursive function when it is defined because of the definition of $\mathsf{part}\ \alpha = \Sigma p, p \to \alpha$. The content of the theorem is then shifted to the construction of the function fix, which was not detailed here but reduces to $\mathsf{nat.find} : (\exists n : \mathbb{N}.\ P(n)) \to \{n \mid P(n)\}$, which ultimately relies on the same subsingleton elimination principle used in Coq.
- In "Formalization of the Undecidability of the Halting Problem for a Functional Language" by Ramos et. al. [15], the authors formalize a simplified version of PVS called PVS0 suitable for translating regular PVS definitions into PVS0 and proving termination, and

they also do some computability theory in this setting, including the fixed point theorem and Rice's theorem using an explicit PVS0 program. Our approach is much more abstract and generic, more suited to the mathematical theory than concrete execution models.

From our own work and the work in these alternative formalizations, we find the following "take-home messages":

- Although the standard formulation of μ-recursive functions uses n-ary functions, and both [18] and [11] use n-ary μ-recursive functions, it turns out that it is much simpler to work with unary μ-recursive functions and rely on the pairing function to get additional arguments. This simplifies the statement of composition and projection significantly and decreases the reliance on dependent types.
- There is not a significant difference between our formulation of partial recursive functions and the lambda calculus with de Bruijn variables, although we don't get the higher-order features until fairly late in the process. (Once we have eval and code we can use codes as higher order functions.) But it is less obvious how to get primitive recursion in the lambda calculus, and having a direct enumeration of all sets under consideration makes it easy to get things like option.map as primitive recursive functions early on.
- Building "synthetic computability" [5] into the types from the beginning makes it obvious that all computable functions are Lean-computable and all partial recursive functions can be evaluated on their domain. All the work is transferred to the single function fix, whose definition is independent of the computability library, and a complicated induction on partial recursive functions is avoided.
- Synthetic computability is convenient when applicable, but in the presence of a "proper" definition of computability, they are incompatible. It is not possible to prove that all synthetically computable functions (that is, all functions) are computable, and this statement is disprovable in classical logic, so we cannot assume it to be the case. (In fact, there is a diagonalization problem here as well; even in no-axioms Lean, we cannot take the assumption that all functions are computable as an axiom without making the axiom false.)

7 Future Work

7.1 Equivalences

The most obvious next step is to show the equivalence of other formulations of computable functions: Turing machines, λ-calculus, Minsky register machines, C... the space of options is very wide here and it is easy to get carried away. Furthermore, if one holds to the thesis that partial recursive functions are the quickest lifeline out of the Turing tarpit, then one must acknowledge that this is to jump right back in, where the hardest part of the translation is fiddling with the intricacies of the target language. We are still looking for ways to do this in a more abstract way that avoids the pain. Forster and Larchey-Wendling [6, 12] have recently made some progress in this direction, connecting Turing machines to Minsky register machines and Diophantine equations.

7.2 Complexity theory

This project was in part intended to set up the foundations of complexity theory. One of the often stated reasons for choosing Turing machines over other models of computation like primitive recursion is because they have a better time model. We would argue that this is not true at fine grained notions of complexity, because there is often a linear multiplicative

overhead for running across the tape compared to memory models. Moreover, in the other direction we find that, at least in the case of polynomial time complexity, there are methods such as bounded recursion on notation [9] that generalize primitive recursion methods to the definition of polynomial time computable functions, which can be used to define **P**, **NP**, and **NP**-hardness at least; we are hopeful that these methods can extend to other classes, possibly by hybridizing with other models of computation as well.

References

1. Andrea Asperti and Wilmer Ricciotti. Formalizing turing machines. In *International Workshop on Logic, Language, Information, and Computation*, pages 1–25. Springer, 2012.
2. Andrej Bauer. First Steps in Synthetic Computability Theory. *Electronic Notes in Theoretical Computer Science*, 155:5–31, 2006.
3. Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.
4. 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.
5. Yannick Forster, Dominik Kirst, and Gert Smolka. On synthetic undecidability in Coq, with an application to the Entscheidungsproblem. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 38–51. ACM, 2019.
6. Yannick Forster and Dominique Larchey-Wendling. Certified undecidability of intuitionistic linear logic via binary stack machines and Minsky machines. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 104–117. ACM, 2019.
7. Yannick Forster and Gert Smolka. Weak Call-by-Value Lambda Calculus as a Model of Computation in Coq. In *ITP*, 2017.
8. Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für mathematik und physik*, 38(1):173–198, 1931.
9. Martin Hofmann. Programming languages capturing complexity classes. *ACM SIGACT News*, 31(1):31–42, January 2000. doi:10.1145/346048.346051.
10. Stephen Cole Kleene. Recursive predicates and quantifiers. *Transactions of the American Mathematical Society*, 53(1):41–73, 1943.
11. Dominique Larchey-Wendling. Typing total recursive functions in Coq. In *International Conference on Interactive Theorem Proving*, pages 371–388. Springer, 2017.
12. Dominique Larchey-Wendling and Yannick Forster. Hilbert's Tenth Problem in Coq. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
13. Michael Norrish. Mechanised computability theory. In *International Conference on Interactive Theorem Proving*, pages 297–311. Springer, 2011.
14. Alan J Perlis. Special feature: Epigrams on programming. *ACM Sigplan Notices*, 17(9):7–13, 1982.
15. Thiago Mendonça Ferreira Ramos, César Muñoz, Mauricio Ayala-Rincón, Mariano Moscato, Aaron Dutle, and Anthony Narkawicz. Formalization of the Undecidability of the Halting Problem for a Functional Language. In *International Workshop on Logic, Language, Information, and Computation*, pages 196–209. Springer, 2018.
16. Alan M Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.
17. Jian Xu, Xingyuan Zhang, and Christian Urban. Mechanising turing machines and computability theory in Isabelle/HOL. In *International Conference on Interactive Theorem Proving*, pages 147–162. Springer, 2013.
18. Vincent Zammit. A proof of the S_n^m theorem in Coq. Technical Report 9-97, University of Kent, 1997.

Formal Proofs of Tarjan's Strongly Connected Components Algorithm in Why3, Coq and Isabelle

Ran Chen
Institute of Software of the Chinese Academy of Sciences, Beijing, China
chenr@ios.ac.cn

Cyril Cohen
Université Côte d'Azur, Inria, Sophia Antipolis, France
cyril.cohen@inria.fr

Jean-Jacques Lévy
Irif & Inria, Paris, France
jean-jacques.levy@inria.fr

Stephan Merz
Université de Lorraine, CNRS, Inria, LORIA, Nancy, France
stephan.merz@inria.fr

Laurent Théry
Université Côte d'Azur, Inria, Sophia Antipolis, France
laurent.thery@inria.fr

Abstract

Comparing provers on a formalization of the same problem is always a valuable exercise. In this paper, we present the formal proof of correctness of a non-trivial algorithm from graph theory that was carried out in three proof assistants: Why3, Coq, and Isabelle.

2012 ACM Subject Classification Software and its engineering → Formal software verification

Keywords and phrases Mathematical logic, Formal proof, Graph algorithm, Program verification

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.13

Supplement Material `http://www-sop.inria.fr/marelle/Tarjan/contributions.html`

1 Introduction

Graph algorithms are notoriously obscure in the sense that it is hard to grasp why exactly they work. Therefore, proofs of correctness are more than welcome in this domain. In this paper we consider Tarjan's algorithm [31] for computing the strongly connected components in a directed graph and present formal proofs of its correctness in three different systems: Why3, Coq and Isabelle/HOL. The algorithm is treated at an abstract level with a functional programming style manipulating finite sets, stacks and mappings, but it respects the linear time behaviour of the original presentation.

To our knowledge this is the first time that the formal correctness proof of a non-trivial program is carried out in three very different proof assistants: Why3 is based on a first-order logic with inductive predicates and automatic provers, Coq on an expressive theory of higher-order logic and dependent types, and Isabelle/HOL combines simply typed higher-order logic with automatic provers. Crucially for our comparison, the algorithm is defined at the same level of abstraction in all three systems, and the proof relies on the same arguments in the three formal systems. We deliberately decided not to base our representation of the algorithm on some specific infrastructure for program verification such as Lammich's monadic or imperative refinement frameworks [17, 19] for Isabelle/HOL or the existing encoding of Back and Morgan's refinement calculus in Coq [1], as doing so would make comparisons between

the systems less direct. Moreover, we do not target automatic generation of executable code. We claim that our proof is direct, readable, elegant, and that it follows Tarjan's presentation. Note that a similar exercise but for a much more elementary proof (the irrationality of square root of 2) and using many more proof assistants (17) was presented in [35].

Examples of formal and informal proofs of algorithms about graphs can be found in [27, 33, 18, 28, 14, 20, 32, 22, 30, 29, 16, 8], among others. Some of them are part of a larger library, others focus on the treatment of pointers or on concurrent algorithms. In particular, only Lammich and Neumann [20] gave an alternative formal proof of Tarjan's algorithm within their framework for verifying graph algorithms in Isabelle/HOL.

We expose here the key parts of the proofs. The interested reader can access the details of the proofs and run them on the web [9, 10, 23]. In this paper, we recall the principles of the algorithm in Section 2; we describe the proofs in the three systems in Sections 3, 4, and 5 by emphasizing the differences induced by the logics which are used; we conclude in Sections 6 and 7 by commenting the developments and advantages of each proof system.

2 The algorithm

In a directed graph, two vertices x and y are strongly connected if there exists a path from x to y and a path from y to x. A strongly connected component (scc) is a maximal set of vertices where all pairs of vertices are strongly connected. The vertices reached by a depth-first search (DFS) traversal in a directed graph form a spanning forest. A fundamental property relates sccs and DFS traversal: each scc is a prefix of a single subtree in the spanning forest (see Figure 1c). Its root is called the base of the scc. Tarjan's algorithm [31] relies on the detection of these bases and collects the sccs in a pushdown stack. It performs a single DFS traversal of the graph assigning a serial number $num[x]$ to any vertex x in the order of the visit. It computes the following function for every vertex x:[1]

$$LOWLINK(x) = \min\{num[y] \mid x \stackrel{*}{\Longrightarrow} z \hookrightarrow y \ \wedge \ x \text{ and } y \text{ are strongly connected}\}$$

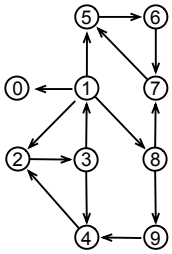
The relation $x \Longrightarrow z$ means that z is a son of x in the spanning forest, the relation $\stackrel{*}{\Longrightarrow}$ is its transitive and reflexive closure, and $z \hookrightarrow y$ means that there is a back edge from z to some node y of the spanning forest (a back edge is an edge of the graph which is not an edge in the spanning forest). In Figure 1c, $\Longrightarrow$ is drawn in thick lines and $\hookrightarrow$ in dotted lines; in Figure 1b the table of the values of the $LOWLINK$ function is shown. The minimum of the empty set is assumed to be $+\infty$ (this is a slight simplification w.r.t. the original algorithm).

The base x of an scc is found when $LOWLINK(x) \geq num[x]$, and the component is formed by the nodes of the subtree in the spanning forest rooted at x and pruned of the sccs already discovered in that subtree. As illustrated by vertices 8 or 9 in Figure 1c, notice that $LOWLINK(x)$ need not be the lowest serial number of a vertex accessible from x, nor of an ancestor of x in the spanning forest. The DFS traversal sets to $+\infty$ the serial numbers of vertices in already discovered sccs. This allows us to compute $LOWLINK$ as:

$$LOWLINK(x) = \min\{num[y] \mid x \stackrel{*}{\Longrightarrow} z \hookrightarrow y\}$$

Our implementation of graphs uses an abstract type *vertex* for vertices, a constant *vertices* for the finite set of all vertices in the graph, and a *successors* function from vertices to their adjacency set. The algorithm maintains an environment e implemented as a record of type *env* with four fields: a stack $e.stack$, a set $e.sccs$ of strongly connected components, a fresh serial number $e.sn$, and a function $e.num$ from vertices to serial numbers.

[1] This definition relies on knowing whether nodes are strongly connected, which the algorithm is intended to discover. This apparent circularity will be broken below.

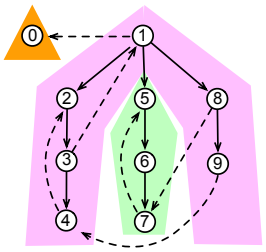


(a) A graph example.

$x =$

0	1	2	3	4	5	6	7	8	9
$LOWLINK(x) =$									
$+\infty$	1	1	1	2	5	5	5	4	4

(b) *LOWLINK*.



(c) Spanning forest.

Figure 1 The vertices are numbered and pushed onto the stack in the order of their visit by the recursive function *dfs1*. When the first component $\{0\}$ is discovered, vertex 0 is popped; similarly when the second component $\{5, 6, 7\}$ is found, its vertices are popped; finally all vertices are popped when the third component $\{1, 2, 3, 4, 8, 9\}$ is found. Notice that there is no back edge to a vertex with a number less than 5 when the second component is discovered. Similarly in the first component, there is no edge to a vertex with a number less than 0. In the third component, there is no edge to a vertex less than 1 since we have set the serial number of vertex 0 to $+\infty$ when 0 was popped.

```
type vertex
constant vertices: set vertex
function successors vertex : set vertex
type env = {stack: list vertex; sccs: set (set vertex); sn: int; num: map vertex int}
```

The DFS traversal is organized as two mutually recursive functions *dfs1* and *dfs*. The function *dfs1* visits a new vertex x and computes $LOWLINK(x)$. Furthermore it adds a new scc when x is the base of a new scc. The function *dfs* takes as argument a set r of roots and an environment e. It calls *dfs1* on non-visited vertices in r and returns a pair consisting of an integer and the modified environment. The integer is the minimum of the values computed by *dfs1* on non-visited vertices in r and the serial numbers of already visited vertices in r. If the set of roots is empty, the returned integer is $+\infty$.

The main procedure *tarjan* initializes the environment with an empty stack, an empty set of sccs, the fresh serial number 0 and the constant function giving the number -1 to each vertex. The result is the set of components returned by the function *dfs* called on all vertices in the graph.

```
let rec dfs1 x e =
  let n0 = e.sn in
  let (n1, e1) = dfs (successors x) (add_stack_incr x e) in
  if n1 < n0 then (n1, e1) else
    let (s2, s3) = split x e1.stack in
    (+∞, {stack = s3; sccs = add (elements s2) e1.sccs;
       sn = e1.sn; num = set_infty s2 e1.num})
with dfs r e = if is_empty r then (+∞, e) else
  let x = choose r in let r' = remove x r in
  let (n1, e1) = if e.num[x] ≠ -1 then (e.num[x], e) else dfs1 x e in
  let (n2, e2) = dfs r' e1 in (min n1 n2, e2)
let tarjan () =
  let e = {stack = Nil; sccs = empty; sn = 0; num = const (-1)} in
  let (_, e') = dfs vertices e in e'.sccs
```

In the body of *dfs1*, the auxiliary function *add_stack_incr* updates the environment by pushing x on the stack, assigning it the current fresh serial number, and incrementing that number in view of future calls. The function *dfs1* performs a recursive call to *dfs* for the adjacent vertices of x as roots and the updated environment. If the returned integer value *n1* is less than the number *n0* assigned to x, the function simply returns *n1* and the current

environment. Otherwise, the function declares that a new scc has been found, consisting of all vertices that are contained on top of x in the current stack. These vertices are popped from the stack, stored as a new set in *e.sccs*, and their serial numbers are all set to $+\infty$, ensuring that they do not interfere with future calculations of min values. The auxiliary functions *split* and *set_infty* are used to carry out these updates.

```
let add_stack_incr x e = let n = e.sn in
  {stack = Cons x e.stack; sccs = e.sccs; sn = n+1; num = e.num[x ← n]}
let rec set_infty s f = match s with Nil → f
  | Cons x s' → (set_infty s' f)[x ← +∞]  end
let rec split x s = match s with Nil → (Nil, Nil)
  | Cons y s' → if x = y then (Cons x Nil, s')
    else let (s1', s2) = split x s' in (Cons y s1', s2)  end
```

Figure 1 illustrates the behavior of the algorithm by an example. We presented the algorithm as a functional program, using data structures available in the Why3 standard library [4]. For lists we have the constructors *Nil* and *Cons*; the function *elements* returns the set of elements of a list. For finite sets, we have the empty set *empty*, and the functions *add* to add an element to a set, *remove* to remove an element from a set, *choose* to pick an arbitrary element[2] in a (non-empty) set, and *is_empty* to test for emptiness. We also use maps with functions *const* denoting the constant function, `_[_]` to access the value of an element, and `_[_ ← _]` for creating a map obtained from an existing map by setting an element to a given value.

For a correspondence between our presentation and the imperative programs used in standard textbooks, the reader is referred to [8]. The present version can be directly translated into Coq or Isabelle functions, and it respects the linear running time of the algorithm for straightforward implementations of the elements that we choose to leave abstract in our presentation. Indeed, vertices could be represented by integers, $+\infty$ by the cardinal of *vertices*, finite sets by lists of integers and mappings by mutable arrays (see for instance [9]).

Thus for each environment e in the algorithm, the working stack *e.stack* corresponds to a cut of the spanning forest where strongly connected components to its left are pruned and stored in *e.sccs*. In this stack, any vertex can reach any vertex higher in the stack. And if a vertex is a base of an scc, no back edge can reach some vertex lower than this base in the stack, otherwise that last vertex would be in the same scc with a strictly lower serial number.

Our proofs of the algorithm make these arguments formal. To maintain these invariants we will distinguish, as is common for DFS algorithms, three sets of vertices: white vertices are the non-visited ones, black vertices are those that are already fully visited, and gray vertices are those that are still being visited. Clearly, these sets are disjoint and white vertices can be considered as forming the complement in *vertices* of the union of the gray and black ones.

The previously mentioned invariant properties can now be expressed for vertices in the stack: no such vertex is white, any vertex can reach all vertices higher in the stack, any vertex can reach some gray vertex lower in the stack. Moreover, vertices in the stack respect the numbering order, i.e. a vertex x is lower than y in the stack if and only if the number assigned to x is strictly less than the number assigned to y.

3 The proof in Why3

The Why3 system comprises the programming language WhyML used in the previous section and a many-sorted first-order logic with inductive data types and inductive predicates to express the logical assertions. The system generates proof obligations w.r.t. the assertions,

[2] This is the only non-deterministic operation used in our development. We handle it by underspecification, effectively showing that any implementation is correct.

pre- and post-conditions and lemmas inserted in the WhyML program. The system is interfaced with off-the-shelf automatic provers and interactive proof assistants.

From the Why3 library, we use pre-defined theories for integer arithmetic, polymorphic lists, finite sets and mappings. There is also a small theory for paths in graphs. Here we define graphs, paths and sccs as follows.

```
axiom successors_vertices: ∀x. mem x vertices → subset (successors x) vertices
predicate edge (x y: vertex) = mem x vertices ∧ mem y (successors x)
inductive path vertex (list vertex) vertex =
  | Path_empty: ∀x: vertex. path x Nil x
  | Path_cons: ∀x y z: vertex, l: list vertex.
      edge x y → path y l z → path x (Cons x l) z
predicate reachable (x y: vertex) = ∃l. path x l y
predicate in_same_scc (x y: vertex) = reachable x y ∧ reachable y x
predicate is_subscc (s: set vertex) =  ∀x y. mem x s → mem y s → in_same_scc x y
predicate is_scc (s: set vertex) = not is_empty s
  ∧ is_subscc s ∧ (∀s'. subset s s' → is_subscc s' → s == s')
```

The predicates *mem* and *subset* denote membership and the subset relation for finite sets.

We add two ghost fields in environments for the black and gray sets of vertices. These fields are used in the proofs but not used in the calculation of the sccs, which is checked by the type-checker of the language.

```
type env = {ghost black: set vertex; ghost gray: set vertex;
            stack: list vertex; sccs: set (set vertex); sn: int; num: map vertex int}
```

Defining a new function *add_black* turning a vertex from gray to black and redefining *add_stack_incr* for adding a new gray vertex with a fresh serial number to the current stack, the functions now become:

```
let add_black x e =
  {black = add x e.black; gray = remove x e.gray;
   stack = e.stack; sccs = e.sccs; sn = e.sn; num = e.num}
let add_stack_incr x e =
  let n = e.sn in
  {black = e.black; gray = add x e.gray;
   stack = Cons x e.stack; sccs = e.sccs; sn = n+1; num = e.num[x ←n]}
let rec dfs1 x e =
  let n0 = e.sn in
  let (n1, e1) = dfs (successors x) (add_stack_incr x e) in
  if n1 < n0 then (n1, add_black x e1) else
    let (s2, s3) = split x e1.stack in
    (+∞, {black = add x e1.black; gray = e.gray; stack = s3;
       sccs = add (elements s2) e1.sccs; sn = e1.sn; num = set_infty s2 e1.num})
with dfs r e = ... (* unmodified *)
let tarjan () =
  let e = {black = empty; gray = empty;
           stack = Nil; sccs = empty; sn = 0; num = const (-1)} in
  let (_, e') = dfs vertices e in e'.sccs
```

The main invariant ($\mathcal{I}$) of our program states that the environment is well-formed:

```
predicate wf_env (e: env) =
  let {stack = s; black = b; gray = g} = e in
  wf_color e ∧ wf_num e ∧ simplelist s ∧ no_black_to_white b g ∧
  (∀x y. lmem x s → lmem y s → e.num[x] ≤ e.num[y] → reachable x y) ∧
  (∀y. lmem y s → ∃x. mem x g ∧ e.num[x] ≤ e.num[y] ∧ reachable y x) ∧
  (∀cc. mem cc e.sccs ↔ subset cc b ∧ is_scc cc)
```

where *lmem* stands for membership in a list. The well-formedness property is the conjunction of seven clauses. The two first clauses express elementary conditions about the colored sets of vertices and the numbering function (see [9, 8] for a detailed description). The third clause

states that there are no repetitions in the stack, and the fourth that there is no edge from a black vertex to a white vertex. The next two clauses formally express the property already stated above: any vertex in the stack reaches all higher vertices and any vertex in the stack can reach a lower gray vertex. The last clause states that the *sccs* field is the set of all sccs all of whose vertices are black.

Since at the end of the *tarjan* function, all vertices are black, the *sccs* field will contain exactly the set of all strongly connected components. We formally express this by adding a post-condition to the definition of the function.

```
let tarjan () = returns {r → ∀cc. mem cc r ↔ subset cc vertices ∧ is_scc cc}
  let e = {black = empty; gray = empty;
           stack = Nil; sccs = empty; sn = 0; num = const (-1)} in
  let (_, e') = dfs vertices e in assert {subset vertices e'.black};
    e'.sccs
```

Our functions *dfs1* and *dfs* modify the environment in a monotonic way. Namely, they augment the set of visited vertices (the black ones); they keep invariant the set of the ones currently under visit (the gray set); they increase the stack with new black vertices; they also discover new sccs and they keep invariant the serial numbers of vertices in the stack,

```
predicate subenv (e e': env) =
  subset e.black e'.black ∧ e.gray == e'.gray
  ∧ (∃s. e'.stack = s ++ e.stack ∧ subset (elements s) e'.black)
  ∧ subset e.sccs e'.sccs ∧ (∀x. lmem x e.stack → e.num[x] = e'.num[x])
```

Once these invariants are expressed, it remains to locate them in the program text and to add assertions which help to prove them. The pre-conditions of *dfs1* are quite natural: the vertex x must be a white vertex of the graph, and it must be reachable from all gray vertices. Moreover invariant ($\mathcal{I}$) must hold. The post-conditions of *dfs1* are the following. Firstly, ($\mathcal{I}$) and the monotonicity property *subenv* hold of the resulting environment. Second, vertex x is black at the end of *dfs1*. Finally we express properties of the integer value n returned by this function, which is indeed $LOWLINK(x)$, as noted previously. Formally, we give three properties for characterizing n. The returned value is never higher than the number of x in the final environment. Also, the returned value is either $+\infty$ or the number of a vertex in the stack reachable from x. Finally, if there is a (back) edge from a vertex y' in the new part of the stack to a vertex y in its old part, the returned value n must be lower or equal to the serial number of y.

```
let rec dfs1 x e  =
(* pre-condition *)
requires {mem x vertices ∧ not mem x (union e.black e.gray)}
requires {∀y. mem y e.gray → reachable y x}
requires {wf_env e} (* I *)
(* post-condition *)
returns {(_, e') → wf_env e' ∧ subenv e e'}
returns {(_, e') → mem x e'.black}
returns {(n, e') → n ≤ e'.num[x]}
returns {(n, e') → n = +∞ ∨ num_of_reachable_in_stack n x e'}
returns {(n, e') → ∀y. xedge_to e'.stack e.stack y → n ≤ e'.num[y]}
```

The auxiliary predicates used above are formally defined in the following way.

```
predicate num_of_reachable_in_stack (n: int) (x: vertex)(e: env) =
     ∃y. lmem y e.stack ∧ n = e.num[y] ∧ reachable x y
predicate xedge_to (s1 s3: list vertex) (y: vertex) =
     (∃s2. s1 = s2 ++ s3 ∧ ∃y'. lmem y' s2 ∧ edge y' y) ∧ lmem y s3
```

Notice that the definition of *xedge_to* fits the definition of *LOWLINK* when the back edge ends at a vertex residing in the stack before the call of *dfs1*. The pre- and post-conditions for the function *dfs* are quite similar up to a generalization to sets of vertices considered as the roots of the algorithm (see [9]).

We now add seven assertions in the body of the *dfs1* function to help the automatic provers. In contrast, the function *dfs* needs no extra assertions in its body. In *dfs1*, when the number *n0* of x is strictly greater than the number *n1* resulting from the call to its successors, the first assertion states that *n1* cannot be $+\infty$; it helps in proving the next assertion. The second assertion states that a lower gray vertex is reachable from x and that thus the scc of x is not fully black at the end of *dfs1*. In that assertion the inequality $y \neq x$ is redundant, but helps showing the *sccs* constraint at the end of *dfs1*. The first four assertions in the "else" branch show that the vertices on top of x in the current stack form a strongly connected component and that x is the base of that scc. The final assertion helps proving that the coloring constraint is preserved at the end of *dfs1*.

```
let n0 = e.sn in
let (n1, e1) = dfs (successors x) (add_stack_incr x e) in
if n1 < n0 then begin
  assert {n1 ≠ +∞};
  assert {∃y. y ≠ x ∧ mem y e1.gray ∧ e1.num[y] < e1.num[x] ∧ in_same_scc x y};
  (n1, add_black x e1) end
else
  let (s2, s3) = split x e1.stack in
  assert {is_last x s2 ∧ s3 = e.stack ∧ subset (elements s2) (add x e1.black)};
  assert {is_subscc (elements s2)};
  assert {∀y. in_same_scc y x → lmem y s2};
  assert {is_scc (elements s2)};
  assert {inter e.gray (elements s2) == empty};
  (+∞, {black = add x e1.black; gray = e.gray; stack = s3;
        sccs = add (elements s2) e1.sccs; sn = e1.sn; num = set_infty s2 e1.num})
```

The function *inter* denotes set intersection, and *is_last* is defined below.

```
predicate is_last (x: α) (s: list α) = ∃s'. s = s' ++ Cons x Nil
```

All proofs are discovered by the automatic provers except for two proofs carried out interactively in Coq. One is the proof of the black extension of the stack (from predicate *subenv* in the post-condition of *dfs1*) in case $n1 < n0$. The provers could not find a suitable witness for the existential quantifier, although the Coq proof is quite short. The second Coq proof is the fifth assertion in the body of *dfs1*, which asserts that any y in the scc of x belongs to *s2*. It is a maximality assertion which states that the set *elements s2* is a complete scc. The proof of that assertion is by contradiction. If y is not in *s2*, there must be an edge from x' in *s2* to some y' not in *s2* such that x reaches x' and y' reaches y. There are three cases, depending on the position of y'. Case 1 is when y' is in *sccs*: this is not possible since x would then be in *sccs* which contradicts x being gray. Case 2 is when y' is an element of *s3*: the serial number of y' is strictly less than the one of x which is *n0*. If $x' \neq x$, the back edge from x' to y' contradicts $n1 \geq n0$ (post-condition 5); if $x' = x$, then y' is a successor of x and again it contradicts $n1 \geq n0$ (post-condition 3). Case 3 is when y' is white, then $x' \neq x$ is impossible since x' is then black in *s2* and would be the origin of a black-to-white edge to y'; if $x' = x$, then y' is not white by post-condition 2 of *dfs*.

Some quantitative information about the Why3 proof is listed in Table 1. Alt-Ergo 2.3 and CVC4 1.5 proved the bulk of the proof obligations.[3] The proof uses 49 lemmas that were all proved automatically, but with an interactive interface providing hints to apply inlining,

[3] In addition to the results reported in the table, Spass was used to discharge one proof obligation.

Table 1 Performance results with provers in Why3-0.88.3 (in seconds, on a 3.3 GHz Intel Core i5 processor). Total time is 341.15 seconds. The two last columns contain the numbers of verification conditions and proof obligations. Notice that there may be several VCs per proof obligation.

provers	Alt-Ergo	CVC4	E-prover	Z3	#VC	#PO
49 lemmas	1.91	26.11	3.33		70	49
split	0.09	0.16			6	6
add_stack_incr	0.01				1	1
add_black	0.02				1	1
set_infty	0.03				1	1
dfs1	77.89	150.2	19.99	13.67	79	20
dfs	4.71	3.52		0.26	58	25
tarjan	0.85				15	5
total	85.51	179.99	23.32	13.93	231	108

splitting, or induction strategies. This includes 13 lemmas on sets, 16 on lists, 5 on lists without repetitions, 3 on paths, 5 on sccs and 7 very specialized lemmas directly involved in the proof obligations of the algorithm. The lemma *xpath_xedge* states a critical condition on paths, reducing a predicate on paths to a predicate on edges. In fact, most of the Why3 proof works on edges, which are handled more robustly by the automatic provers than paths. Another important lemma is *subscc_after_last_gray*, which shows that the elements of the stack on top of the last gray vertex form a subset of an scc. This means that a variant of the program with the *split* call before the if-statement would have a simpler proof, but its time complexity would be non-linear. The two Coq proofs are 9 and 81 lines long (the Coq files of 677 and 680 lines include preambles that are automatically generated during the translation from Why3 to Coq). The interested reader is refered to [9] where the full proof is available.

The proof explained so far only showed the partial correctness of the algorithm. But after adding two lemmas about union and difference for finite sets, termination is automatically proved by the following lexicographic ordering on the number of white vertices and roots.

```
let rec dfs1 x e  = variant {cardinal (diff vertices (union e.black e.gray)), 0}
with dfs r e = variant {cardinal (diff vertices (union e.black e.gray)), 1, cardinal r}
```

4 The proof in Coq

Coq is based on type theory and the calculus of constructions, a higher order lambda-calculus, to express formulae and proofs. Some basic notions of graph theory are provided by the Mathematical Components Library [21]. Our formalization is parameterized by a finite type V for the vertices and the function *successors* such that *successors* x is the adjacency set of any vertex x. The boolean *gconnect* x y indicates that a path connects the vertex x to the vertex y. The function *equivalence_partition* of the library creates a partition of a set with respect to an equivalence relation. The set *gsccs* of the sccs of a graph is defined as follows, based on *gconnect* and the set of all vertices *[set: V]*:

```
Definition gsymconnect x y := gconnect x y && gconnect y x.
Definition gsccs := equivalence_partition gsymconnect [set: V].
```

Components are represented as sets of sets *{set {set V}}*. Various operations on sets are available. In this proof, we use singletons *[set x]*, unions $S_1 \cup S_2$, differences $S_1 \setminus S_2$, complements $\sim: S$ and unions of all sets in a set of sets *cover* S.

Coq offers several mechanisms for combining properties (boolean conjunction, propositional conjunction, record, inductive family), all of which have their own idiosyncracies. In order to make the presentation more readable for a non-Coq expert, we write them all with the n-ary propositional conjunction [∧ P$_1$, P$_2$, … & P$_n$]. We refer to [10] for the actual code.

The Coq formalization differs from the one in Why3: it uses natural numbers only and does not mention colors (white, gray and black). In particular, the number ∞ is defined as the cardinality of V, vertices with $\infty + 1$ as serial number correspond to the white vertices of the previous section and the environment is defined as a record with only two fields, a set of sccs and the mapping assigning serial numbers to vertices:

```
Record env := Env {esccs : {set {set V}}; num: {ffun V → nat}}.
```

Given an environment e, the set of visited vertices is *visited e* (the vertices with serial number less or equal to ∞), the current fresh serial number is *sn e* (the cardinality of the set of visited vertices), and the stack is *stack e* (the list of elements x which satisfy *num e x* $<$ *sn e*, sorted by increasing serial number).

Another difference with the Why3 algorithm is the definition of *dfs1* and *dfs* as two separate, rather than two mutually recursive functions. As in the Why3 program, *dfs1* takes a vertex x, and *dfs* a set of vertices *roots*, in addition to an environment e. In order to prepare for the combination of the two functions, they also take function parameters that will represent the recursive calls.

```
Definition dfs1 dfs x e :=
  let: (n1, e1) as res := dfs (successors x) (visit x e) in
  if n1 < sn e then res else (∞, store (stack e1 \ stack e) e1).
Definition dfs dfs1 dfs (roots : {set V}) e :=
 if [pick x in roots] isn't Some x then (∞, e) else
 let: (n1, e1) := if num e x ≤ ∞ then (num e x, e) else dfs1 x e in
 let: (n2, e2) := dfs (roots \ [set x]) e1 in (minn n1 n2, e2).
```

The expression *visit x e* represents the environment where x gets the next serial number, and *store* produces an environment that contains an additional strongly connected component.

Then, the two functions are glued together in a recursive function *rec* where the parameter k controls the maximal recursive height.

```
Fixpoint rec k r e := if k is k'.+1 then dfs (dfs1 (rec k')) (rec k') r e else (∞, e).
```

If k is not zero (i.e. it is a successor of some *k'*), *rec* calls *dfs* taking care that its parameters can only use recursive calls to *rec* with a smaller recursive height, here *k'*. This ensures termination. A dummy value is returned in the case where k is zero. Finally, the top level *tarjan* calls *rec* with the proper initial arguments.

```
Definition tarjan := let: (_, e) := rec (∞ * (∞.+2)) V (Env ∅ [ffun⇒ ∞.+1]) in esccs e.
```

Initially, the roots are all the vertices (V) and the environment has no component and all vertices are not visited (their number is $\infty + 1$). As both *dfs* and *dfs1* cannot be applied more than the number of vertices, the value $\infty * (\infty + 2)$ encodes the lexicographic product of the two maximal heights. It gives *rec* enough fuel to never encounter the dummy value so *tarjan* correctly terminates the computation. This allows us to separate the proof of the termination from the algorithm itself, and this last statement is of course proved formally later as theorem *rec_terminates*.

The invariants of the Coq proof are usually shorter than in the Why3 proof since they do not mention colors. We first define well-formed environments and their valid extension:

```
Definition wf_env e := [∧ esccs e ⊆ gsccs,
  ∀x, num e x < ∞ → num e x < sn e,
  ∀x, (num e x = ∞) = (x ∈ cover (esccs e)) &
  ∀x y, num e x ≤ num e y < sn e → gconnect x y].
Definition subenv e1 e2 := [∧ esccs e1 ⊆ esccs e2,
  ∀x, num e1 x < ∞ → num e2 x = num e1 x &
  ∀x, num e2 x < sn e1 → num e1 x < sn e1].
```

Then we state that new visited vertices are the ones reachable by paths accessible from roots with non-visited vertices (i.e. by white paths in the colored setting). The function *nexts* such that *nexts D X* returns the set of vertices reachable from the set X by a path which only contains vertices in D except maybe the last one.

```
Definition outenv (roots : {set V}) (e e' : env) := [∧
  ∀x y, x ∈ stack e' \ stack e → y ∈ stack e' \ stack e → gconnect x y,
  ∀x, x ∈ stack e' \ stack e → ∃y, y ∈ stack e ∧ gconnect x y &
  visited e' = visited e ∪ nexts (∼: visited e) roots ].
```

The post-condition is the conjunction of these three properties and the characterization of the output rank:

```
Definition dfs_spec (ne' : nat * env) (roots : {set V}) e := let: (n, e') := ne' in
  [∧ n = \min_(x in nexts (∼: visited e) roots) inord (num e' x),
     wf_env e', subenv e e' & outenv roots e e']].
```

Here, the argument *ne'* is the result of a *dfs*. The output rank n is the minimum of the serial numbers of the vertices which can be reached from the roots through a path where all the vertices except maybe the last one were not already visited. Note that this characterization differs from the notion of *LOWLINK*, which requires that the last vertex was visited.

Finally, we express correctness as the implication between pre- and post-conditions:

```
Definition dfs_correct dfs (roots : {set V}) e := wf_env e →
  (∀x y, x ∈ stack e → y ∈ roots → gconnect x y) → dfs_spec (dfs roots e) roots e.
Definition dfs1_correct dfs1 x e := wf_env e → x ∉ visited e →
  (∀x y, x ∈ stack e → y ∈ [set x] → gconnect x y) → dfs_spec (dfs1 x e) [set x] e.
```

Although these invariants are expressed differently from the formulation in Why3, they reflect essentially the same ideas. Compared to a first version of the formal model that was more closely aligned with the Why3 representation, this more abstract version made it possible to reduce by approximately 50% the size of the Coq proofs. The two central theorems are:

```
Lemma dfsP dfs1 dfsrec (roots : {set V}) e : (∀x, x ∈ roots → dfs1_correct dfs1 x e) →
   (∀x, x ∈ roots → ∀e1, subenv e e1 → dfs_correct dfsrec (roots \ [set x]) e1) →
  dfs_correct (dfs dfs1 dfsrec) roots e.
Lemma dfs1P dfs x e : dfs_correct dfs (successors x) (visit x e) →
  dfs1_correct (dfs1 dfs) x e.
```

They state that *dfs* and *dfs1* are correct if their respective recursive calls are correct. The proof of the first lemma is straightforward since *dfs* simply iterates on a list. It mostly requires book-keeping between what is known and what needs to be proved. This is done in about 54 lines. The second one is more intricate and requires 124 lines. Gluing these two theorems together and proving termination gives us an extra 12 lines to prove, and the correctness of *tarjan* then follows directly in 19 lines of straightforward proof.

Table 2 Distribution of the numbers of lines of the 43 proofs in the file *tarjan_nocolors*.

Number of lines	1	2	3	4	5	6	11	12	16	19	54	124
Number of proofs	19	7	5	2	1	2	2	1	1	1	1	1

```
Theorem rec_terminates k (roots : {set V}) e :
  k ≥ #|~: visited e| * (∞.+1) + #|roots| → dfs_correct (rec k) roots e.
Theorem tarjan_correct : tarjan = gsccs.
```

We now provide some quantitative information. The Coq contribution consists of two files. Module *extra_nocolors* defines the *bigmin* operator and some notions of graph theory that we intend to add to Mathematical Components. This file is 294 lines long. The main module is *tarjan_nocolors* and is 605 lines long. It is compiled in 12 seconds with a memory footprint of 800 Mb (3/4 of which are resident) on a Intel® i7 2.60GHz quad-core laptop running Linux. The proofs are performed in the SSReflect proof language [15] with very little automation. The proof script is mostly procedural, alternating book-keeping tactics (*move*) with transformational ones (mostly *rewrite* and *apply*), but often intermediate steps are explicitly declared with the *have* tactic. There are more than fifty of such intermediate steps in the 320 lines of proof of the file *tarjan_nocolors*. Table 2 gives the distribution of the numbers of lines of these proofs. Most of them are very short (26 are at most 2 lines long) and the only complicated proof is the one corresponding to the lemma *dfs1P*.

5 The proof in Isabelle/HOL

Isabelle/HOL [24] is the encoding of simply typed higher-order logic in the logical framework Isabelle [26]. Unlike Why3, it is not primarily intended as an environment for program verification and does not contain specific syntax for stating pre- and post-conditions or intermediate assertions in function definitions. Although logics and formalisms for program verification have been developed within Isabelle/HOL (e.g., [19]), we decided to express our formalization in plain Isabelle/HOL: our main objective is to compare a proof of a significant algorithm in different proof assistants. The constructions that we use are universally available and do not rely on any elaborate infrastructure that would make comparisons more difficult.

We start by introducing a *locale*, fixing parameters and assumptions for the remainder of the proof. Unlike in Why, where the *set* type constructor represents finite sets, we explicitly assume that the set of vertices is finite.

```
locale graph =
  fixes vertices :: ν set  and  successors :: ν ⇒ ν set
  assumes finite vertices  and  ∀v ∈ vertices. successors v ⊆ vertices
```

We introduce reachability using an inductive predicate definition, rather than via an explicit reference to paths as in Why3. Isabelle then generates appropriate induction theorems.

```
inductive reachable where
  reachable x x
| ⟦y ∈ successors x; reachable y z⟧ ⟹ reachable x z
```

The definition of strongly connected components mirrors that used in Why3. The following lemma states that SCCs are disjoint; its one-line proof is found automatically using *Sledgehammer* [3], which heuristically selects suitable lemmas from the set of available facts (including Isabelle's library), invokes several automatic provers, and finally reconstructs a proof that is checked by the Isabelle kernel.

```
definition is_subscc where
  is_subscc S ≡ ∀x ∈ S. ∀y ∈ S. reachable x y
definition is_scc where
  is_scc S ≡ S ≠ {} ∧ is_subscc S ∧ (∀S'. S ⊆ S' ∧ is_subscc S' ⟶ S' = S)
lemma scc_partition:
  assumes is_scc S and is_scc S' and x ∈ S ∩ S'
  shows   S = S'
```

Environments are represented by records with the same components as in Why3, and the definition of the well-formedness predicate is also essentially identical to Why3.[4]

```
record ν env =
  black :: ν set      gray :: ν set
  stack :: ν list     sccs :: ν set set      sn :: nat      num :: ν ⇒ int
definition wf_env where wf_env e ≡
  wf_color e ∧ wf_num e ∧ distinct (stack e) ∧ no_black_to_white e
∧ (∀x y. y ⪯ x in (stack e) ⟶ reachable x y)
∧ (∀y ∈ set (stack e). ∃g ∈ gray e. y ⪯ g in (stack e) ∧ reachable y g)
∧ sccs e = { C . C ⊆ black e ∧ is_scc C }
```

We now define the two mutually recursive functions *dfs1* and *dfs* that expect as arguments a vertex x and a set of vertices *roots*, as well as an environment.

```
function (domintros) dfs1 and dfs where
  dfs1 x e =
   (let (n1,e1) = dfs (successors x) (add_stack_incr x e) in
     if n1 < int (sn e) then (n1, add_black x e1)
     else (let (l,r) = split_list x (stack e1) in
       (+∞, ⦇ black = insert x (black e1), gray = gray e,
              stack = r, sn = sn e1, sccs = insert (set l) (sccs e1),
              num = set_infty l (num e1) ⦈ ))) and
  dfs roots e =
   (if roots = {} then (+∞, e)
    else (let x = SOME x. x ∈ roots;
        res1 = (if num e x ≠ -1 then (num e x, e) else dfs1 x e);
        res2 = dfs (roots - {x}) (snd res1)
        in  (min (fst res1) (fst res2), snd res2) ))
```

The **function** keyword introduces the definition of a recursive function. Isabelle checks that the definition is well-formed and generates appropriate simplification and induction theorems. Because HOL is a logic of total functions, two proof obligations are introduced: the first one requires the user to prove that the cases in the function definitions cover all type-correct arguments; this holds trivially for the above definitions. The second obligation requires exhibiting a well-founded ordering on the function parameters that ensures the termination of recursive function invocations, and Isabelle provides a number of heuristics that work in many cases. However, the functions defined above will in fact not terminate for arbitrary calls, in particular for environments that assign the serial number -1 to non-white vertices. The *domintros* attribute instructs Isabelle to consider these functions as "partial". More precisely, it introduces an auxiliary predicate that represents the domains for which the functions are defined. This "domain condition" appears as a hypothesis in the simplification rules that mirror the function definitions. In particular, a function call can be replaced by the right-hand side of the definition only if the domain predicate holds. Isabelle also introduces (mutually inductive) rules for proving when the domain condition is known (or assumed) to hold. Our first objective is therefore to establish sufficient conditions that ensure

[4] We use the infix operator $\preceq$ to denote precedence in lists.

the termination of the two functions. Assuming the domain condition, we prove that the functions never decrease the set of colored vertices and that vertices are never explicitly assigned the number -1 by our functions. Denoting the union of gray and black vertices as *colored*, we show that the algorithm only colors vertices and never assigns them -1. We then prove that the triples

```
(vertices - colored e, {x}, 1)
(vertices - colored e, roots, 2)
```

for the arguments of *dfs1* and *dfs*, respectively, decrease w.r.t. lexicographical ordering on finite subset inclusion and $<$ on natural numbers across recursive function calls, provided that *colored_num* holds when the function is called and that x is a white vertex. These conditions are therefore sufficient to ensure that the domain condition holds:[5]

```
theorem dfs1_dfs_termination:
  ⟦x ∈ vertices - colored e; colored_num e⟧ ⟹ dfs1_dfs_dom (Inl(x,e))
  ⟦roots ⊆ vertices; colored_num e⟧ ⟹ dfs1_dfs_dom (Inr(roots,e))
```

The proof of partial correctness follows the same ideas as the proof presented for Why3. We define the pre- and post-conditions of the two functions as predicates in Isabelle. For example, the two predicates for *dfs1* are defined as follows:

```
definition dfs1_pre where dfs1_pre e ≡
  wf_env e ∧ x ∈ vertices ∧ x ∉ colored e ∧ (∀g ∈ gray e. reachable g x)
definition dfs1_post where dfs1_post x e res ≡
  let n = fst res; e' = snd res
  in  wf_env e' ∧ subenv e e' ∧ roots ⊆ colored e'
    ∧ (∀x ∈ roots. n ≤ num e' x)
    ∧ (n = +∞ ∨ (∃x ∈ roots. ∃y in set (stack e'). num e' y = n ∧ reachable x y))
```

We now prove the following theorems:

- The pre-condition of each function establishes the pre-condition of every recursive call appearing in the body of that function. For the second recursive call in the body of *dfs* we also assume the post-condition of the first recursive call.
- The pre-condition of each function, plus the post-conditions of each recursive call in the body of that function, establishes the post-condition of the function.

Combining these results, we establish partial correctness:

```
theorem dfs_partial_correct:
  ⟦dfs1_dfs_dom (Inl(x,e)); dfs1_pre x e⟧ ⟹ dfs1_post x e (dfs1 x e)
  ⟦dfs1_dfs_dom (Inr(roots,e)); dfs_pre roots e⟧ ⟹ dfs_post roots e (dfs roots e)
```

We now define the initial environment and the overall function. It is then trivial to show that the arguments to the call of *dfs* in the definition of *tarjan* satisfy the pre-condition of *dfs*. Putting together the theorems establishing termination and partial correctness, we obtain the desired total correctness results.

```
definition init_env where init_env ≡
  (| black = {},  gray = {},  stack = [],  sccs = {},  sn = 0,  num = λ_. -1 |)
definition tarjan where tarjan ≡
  sccs (snd (dfs vertices init_env))
theorem dfs_correct:
  dfs1_pre x e ⟹ dfs1_post x e (dfs1 x e)
  dfs_pre roots e ⟹ dfs_post roots e (dfs roots e)
theorem tarjan_correct:
  tarjan = { C . is_scc C ∧ C ⊆ vertices }
```

[5] Observe that Isabelle introduces a single predicate *dfs1_dfs_dom* corresponding to the two mutually recursive functions whose domain is the disjoint sum of the domains of both functions.

Table 3 Distribution of interactions in the Isabelle proofs.

$i = 1$	$i \leq 5$	$i \leq 10$	$i \leq 20$	$i \leq 30$	$i = 35$	$i = 43$	$i = 48$
28	8	4	1	2	1	1	1

The intermediate assertions that were inserted in the Why3 code guided the overall proof in Isabelle: they are established either as separate lemmas or as intermediate steps within the proofs of the above theorems. Similarly to the Coq proof, the overall induction proof was explicitly decomposed into individual lemmas as laid out above. In particular, whereas Why3 identifies the predicates that can be used from the function code and its annotation with pre- and post-conditions, these assertions appear explicitly in the intermediate lemmas used in the proof of theorem *dfs_partial_correct*. The induction rules that Isabelle generated from the function definitions were helpful for finding the appropriate decomposition of the overall correctness proof.

We extensively used *sledgehammer* in the development of these proofs for invoking automatic back-end provers, including the superposition provers E, Spass, and Vampire, and the SMT solvers CVC4 and Z3. Nevertheless, we found that in comparison to Why3, significantly more user interactions were necessary in order to guide the proof. On several occasions, the external back-end reported finding a proof, but the subsequent attempt to reconstruct a proof in Isabelle failed: sledgehammer mainly records the lemmas that are used by the automatic back-end and then invokes proof tools (such as *metis*) that can generate a detailed proof that can be certified by the Isabelle kernel, but these tools may not find a proof even when the original automatic prover succeeded. When automatic proof fails, the user has to decompose the proof into smaller steps. Although decomposition was often straightforward, a few steps were not so obvious and required designing a rather detailed proof strategy. Table 3 indicates the distribution of the number of interactions used for the proofs of the 46 lemmas the theory contains. These numbers cannot be compared directly to those shown in Table 2 for the Coq proof because an Isabelle interaction is typically much coarser-grained than a line in a Coq proof. As in the case of Why3 and Coq, the proofs of partial correctness of *dfs1* (split into two lemmas following the case distinction) required the most effort. It took about one person-month to carry out the case study, starting from an initial version of the Why3 proof. Processing the entire Isabelle theory on a laptop with a 2.7 GHz Intel® Core i5 (dual-core) processor and 8 GB of RAM takes 35 CPU seconds.

6 General comments about the proof

Our formal proofs refer to colors, finite sets, and the stack, although the informal correctness argument is about properties of strongly connected components in spanning trees. The algorithmician would explain the algorithm with spanning trees as in Tarjan's article. It would be nice to extract a program from such a proof, but the program does not explicitly manipulate spanning trees, and the proof should be given in terms of variables and data that appear in the program.

A first version of the formal proof used *ranks* in the working stack and a flat representation of environments by adding extra arguments to functions for the black, gray, scc sets and the stack. The automatic provers of Why3 worked very well with this representation. But after remodelling the proof in Coq and Isabelle/HOL, it appeared to be cleaner to gather these extra arguments in records and have a single extra argument for environments. Also *ranks* disappeared in favor of the *num* function and the precedence relation, which are easier to understand. Why3's automatic provers have more difficulties with the inlining of environments, but with a few hints they could still succeed.

Proving the correctness of Tarjan's algorithm requires surprisingly few, and entirely elementary, concepts of finite graphs. With the exception of the use of the Mathematical Components library for Coq, we therefore did not use existing libraries formalizing advanced concepts of graph theory [12, 25].

When designing a formal representation of an algorithm, one has to decide at what level of abstraction the algorithm should be modeled. For example, the Coq formalization shows that one can represent Tarjan's algorithm and proof using just serial numbers and the set of strongly connected components found so far, and that the stack used in the algorithm and the colors used in the proof can be reconstructed. The Why3 and Isabelle representations make these elements explicit: coloring of vertices is frequently used when reasoning about graph algorithms, and including the stack allows us to capture the linear time complexity of the algorithm.

There is always a tension between the concision of the proof, its clarity and its relation to the real program. Our presentation aimed at comparing different proof assistants, and we have allowed for a few redundancies while staying at the algorithmic level rather than capturing an implementation.

7 Conclusion

The formal proof expressed in this article was initially designed and implemented in Why3 [8] as the result of a long process, nearly a two-year half-time work with many attempts of proofs about various graph algorithms (depth first search, Kosaraju strong connectivity, bi-connectivity, articulation points, minimum spanning tree). Why3 has a clear separation between programs and the logic. It makes the correctness proof quite readable for a programmer. Also first-order logic is easy to understand. Moreover, one can prove partial correctness without caring about termination.

Another important feature of Why3 is its interface with various off-the-shelf theorem provers (mainly SMT provers). Thus the system benefits from the current technology in theorem provers. Clerical sub-goals can be delegated to these provers, which makes the overall proof shorter and easier to understand. Although the proof must be split in more elementary pieces, this has the benefit of improving its readability. Several hints about inlining or induction reasoning are still needed, and two Coq proofs were used. Technically, the automatic provers and the translations from the Why3 representation to their input languages are part of the trusted code base: proofs are not checked independently. The system records sessions and facilitates incremental proofs. However, the automatic provers are sometimes no longer able to handle a proof obligation after seemingly minor modifications to the formulation of the algorithm or the predicates, making the proof somewhat unstable.

The Coq and Isabelle proofs were inspired by the Why3 proof. Their development therefore required much less time although their text is longer. We do not know if the final proof would have been significantly different had the initial proof been developed in another system than Why3. The Coq proof uses SSReflect and the Mathematical Components library, which helps reduce the size of the proof compared to classical Coq. The proof also uses the bigops library and several other higher-order features which makes it more abstract.

In Coq, one could prove termination using well-foundedness [2, 5], but because of nested recursion the `Function` command fails, and both `Equations` and `Program Fixpoint` require the addition of an extra proof argument to the function. Instead, we define the functionals *dfs1* and *dfs* and recombine them in *rec* and *tarjan* by recursion on a natural number used as fuel. We prove partial correctness on functionals and postpone termination on *rec*.

Table 4 Characteristics of the three formal systems for our case study.

	Why3	Coq	Isabelle/HOL
expressivity	–	+	+
readability	+	–	+
stability w.r.t changes	–	+	+
ease of use	–	–	–
automation	+	–	+
partial correctness vs. termination	+	–	–
trusted base	–	+	+
lines of automatic proof	395	0	314 ui
lines of manual proof	90	898	1690

Our Coq proof does not use significant automation.[6] All details are explicitly expressed, but many of them were already present in the Mathematical Components library. Moreover, a proof certificate is produced and a functional program could in principle be extracted. The absence of automation makes the system very stable to use since the proof script is explicit, but it requires a higher degree of expertise from the user.

The Isabelle/HOL proof can be seen as a mid-point between the Why3 and Coq proofs. It uses higher order logic and the level of abstraction is close to the one of the Coq proof, although more readable in this case study. The proof makes use of Isabelle's extensive support for automation. In particular, *sledgehammer* [3] was very useful for finding individual proof steps. It heuristically selects lemmas and facts available in the context and then calls automatic provers (SMT solvers and superposition-based provers for first-order logic). When one of these provers finds a proof, sledgehammer attempts to find a proof that can be certified by the Isabelle kernel, using various proof methods such as combinations of rewriting and first-order reasoning (blast, fastforce etc.), calls to the *metis* prover or reconstruction of SMT proofs through the *smt* proof method. Unlike in Why3, the automatic provers used to find the initial proof are not part of the trusted code base because ultimately the proof is checked by the kernel. The price to pay is that the degree of automation in Isabelle is still significantly lower compared to Why3. Adapting the proof to modified definitions was fast: the Isabelle/jEdit GUI eagerly processes the proof script and quickly indicates those steps that require attention.

The Isabelle proof also faces the termination problem to achieve general consistency. We chose to delay handling termination, using the *domintros* attribute. The proofs of termination and of partial correctness are independent; in particular, we obtain a weaker predicate ensuring termination than the one used for partial correctness. Although the basic principle of the termination proof is similar to the Coq proof and relies on considering functionals of which the recursive functions are fixpoints, its technical formulation based on an appropriate well-founded order is closer to informal arguments that programmers would give and avoids low-level reasoning about fuel.

One strong point of Isabelle/HOL is its nice LaTeX output and the flexibility of its parser, supporting mathematical symbols. Combined with the hierarchical Isar proof language [34], the proof is in principle understandable without actually running the system.

[6] Hammers exist for Coq [11, 13] but unfortunately they currently perform badly when used in conjunction with the Mathematical Components library.

In the end, the three systems Why3, Coq, and Isabelle/HOL are mature, and each one has its own advantages w.r.t. readability, expressivity, stability, ease of use, automation, partial-correctness, code extraction, trusted base and length of proof (a subjective assessment appears in Table 4). Coming up with invariants that are both strong enough and understandable was by far the hardest part in this work. This effort requires creativity and understanding, although proof assistants provide some help: missing predicates can be discovered by understanding which parts of the proof fail. We think that formalizing the proof in all three systems was very rewarding and helped us better understand the state of the art in computer-aided deductive program verification. We would welcome further comparisons based on implementations of this quite challenging case study in other formal systems.[7]

Our work has not considered how the systems that we have used could have helped us generate an executable implementation of this algorithm, leave alone an efficient one, to imperative programs and concrete data structures. Formal refinement enables proceeding from the high-level correctness argument down to actually executable code, and there is support for verifying imperative programs in general-purpose proof assistants (e.g., [6, 7, 19]). However, the existing frameworks differ significantly, making comparisons quite difficult.

A final and totally different remark is about teaching of algorithms. Do we want students to formally prove algorithms, or to present algorithms with assertions, pre- and post-conditions, and make them prove these assertions informally as exercises? In both cases, we believe that our work could make a useful contribution.

References

1 João Alpuim and Wouter Swierstra. Embedding the refinement calculus in Coq. *Sci. Comput. Program.*, 164:37–48, 2018.

2 G. Barthe, J. Forest, D. Pichardie, and V. Rusu. Defining and reasoning about recursive functions: a practical tool for the Coq proof assistant. In *Functional and LOgic Programming Systems (FLOPS'06)*, volume 3945 of *LNCS*, pages 114–129, Fuji Susono, Japan, April 2006.

3 Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending Sledgehammer with SMT Solvers. *J. Automated Reasoning*, 51(1):109–128, 2013.

4 François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. *The Why3 platform, version 0.86.1*. LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.86.1 edition, May 2015. Available at `why3.lri.fr/download/manual-0.86.1.pdf`.

5 Ana Bove, Alexander Krauss, and Matthieu Sozeau. Partiality and recursion in interactive theorem provers - an overview. *Mathematical Structures in Computer Science*, 26(1):38–88, 2016.

6 Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In *Proc. 16th ACM SIGPLAN Intl. Conf. Functional Programming*, pages 418–430, Tokyo, Japan, 2011. ACM.

7 Arthur Charguéraud. Higher-order Representation Predicates in Separation Logic. In *Proc. 5th ACM SIGPLAN Conf. Certified Programs and Proofs*, CPP 2016, pages 3–14, New York, NY, USA, 2016. ACM.

8 Ran Chen and Jean-Jacques Lévy. A Semi-automatic Proof of Strong connectivity. In *VSTTE 2017*, volume 10712 of *LNCS*, pages 49–65. Springer, 2017.

9 Ran Chen and Jean-Jacques Lévy. Full scripts of Tarjan SCC Why3 proof. Technical report, Iscas and Inria, 2017. `jeanjacqueslevy.net/why3`.

[7] We have set up a Web page `http://www-sop.inria.fr/marelle/Tarjan/contributions.html` in order to collect formalizations.

10 Cyril Cohen and Laurent Théry. Full script of Tarjan SCC Coq/ssreflect proof, 2017. Available at `https://www-sop.inria.fr/marelle/Tarjan/`.

11 Łukasz Czajka and Cezary Kaliszyk. Hammer for Coq: Automation for Dependent Type Theory. *J. Autom. Reasoning*, 61(1-4):423–453, 2018.

12 Christian Doczkal, Guillaume Combette, and Damien Pous. A Formal Proof of the Minor-Exclusion Property for Treewidth-Two Graphs. In *ITP 2018*, volume 10895 of *LNCS*, pages 178–195. Springer, 2018.

13 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 *CAV (2)*, volume 10427 of *LNCS*, pages 126–133. Springer, 2017.

14 Jean-Christophe Filliâtre et al. The Why3 gallery of verified programs. Technical report, CNRS, Inria, U. Paris-Sud, 2015. `toccata.lri.fr/gallery/why3.en.html`.

15 Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in Coq. *J. Formalized Reasoning*, 3(2):95–152, 2010.

16 Aquinas Hobor and Jules Villard. The Ramifications of Sharing in Data Structures. In *Proc. 40th Ann. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, POPL '13, pages 523–536, New York, NY, USA, 2013. ACM.

17 Peter Lammich. Refinement for Monadic Programs. *Archive of Formal Proofs*, 2012. URL: `https://www.isa-afp.org/entries/Refine_Monadic.shtml`.

18 Peter Lammich. Verified Efficient Implementation of Gabow's Strongly Connected Component Algorithm. In *ITP 2015*, volume 8558 of *LNCS*, pages 325–340, Vienna, Austria, 2014. Springer.

19 Peter Lammich. Refinement to Imperative/HOL. *J. Automated Reasoning*, 2019.

20 Peter Lammich and René Neumann. A Framework for Verifying Depth-First Search Algorithms. In *Proc. 4th ACM SIGPLAN Conf. Certified Programs and Proofs*, CPP '15, pages 137–146, New York, NY, USA, 2015. ACM.

21 Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Available at: `https://math-comp.github.io/mcb/`, 2016.

22 Farhad Mehta and Tobias Nipkow. Proving Pointer Programs in Higher-Order Logic. In *CADE*, 2003.

23 Stephan Merz. Formalization of Tarjan's Algorithm in Isabelle, 2018. Available at `https://members.loria.fr/SMerz/projects/tarjan/index.html`.

24 Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. Number 2283 in Lecture Notes in Computer Science. Springer Verlag, 2002.

25 Lars Noschinski. A Graph Library for Isabelle. *Mathematics in Computer Science*, 9(1):23–39, 2015.

26 Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

27 Christopher M. Poskitt and Detlef Plump. Hoare-Style Verification of Graph Programs. *Fundamenta Informaticae*, 118(1-2):135–175, 2012.

28 François Pottier. Depth-First Search and Strong Connectivity in Coq. In *Journées Francophones des Langages Applicatifs (JFLA 2015)*, January 2015.

29 Azalea Raad, Aquinas Hobor, Jules Villard, and Philippa Gardner. Verifying Concurrent Graph Algorithms. In *APLAS 2016*, volume 10017 of *LNCS*, pages 314–334, Hanoi, Vietnam, 2016. Springer.

30 Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized Verification of Fine-grained Concurrent Programs. In *Proc. 36th ACM SIGPLAN Conf. Programming Language Design and Implementation*, PLDI '15, pages 77–87, New York, NY, USA, 2015. ACM.

31 Robert Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.

32 Laurent Théry. Formally-proven Kosaraju's algorithm. Inria report, Hal-01095533, 2015.

33 Ingo Wengener. A Simplified Correctness Proof for a Well-Known Algorithm Computing Strongly Connected Components. *Information Processing Letters*, 83(1):17–19, 2002.

34 Markus Wenzel. Isar – A Generic Interpretative Approach to Readable Formal Proof Documents. In *TPHOLS'99*, volume 1690 of *LNCS*, pages 167–184, Nice, France, 1999. Springer.

35 Freek Wiedijk. *The Seventeen Provers of the World*, volume 3600 of *LNCS*. Springer, Berlin, Heidelberg, 2006.

First-Order Guarded Coinduction in Coq

Łukasz Czajka
Faculty of Informatics, TU Dortmund University, Germany
lukaszcz@mimuw.edu.pl

Abstract

We introduce two coinduction principles and two proof translations which, under certain conditions, map coinductive proofs that use our principles to guarded Coq proofs. The first principle provides an "operational" description of a proof by coinduction, which is easy to reason with informally. The second principle extends the first one to allow for direct proofs by coinduction of statements with existential quantifiers and multiple coinductive predicates in the conclusion. The principles automatically enforce the correct use of the coinductive hypothesis. We implemented the principles and the proof translations in a Coq plugin.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Logic and verification

Keywords and phrases coinduction, Coq, guardedness, corecursion

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.14

Related Version A full version of the paper including the appendix is available at `https://www.mimuw.edu.pl/~lukaszcz/focoind.pdf`.

Supplement Material The Coq plugin is available at `https://github.com/lukaszcz/coinduction`.

1 Introduction

Coinduction has been studied for several decades now, and is being used increasingly often in practice. Most formal coinduction principles are based on the lattice-theoretic Knaster-Tarski fixpoint theorem [19, 18], on category theory [13, 16], on a syntactic description of legal proofs [5, 10], or on corecursors [15, 9]. Arguably, these principles are not well-suited for informal reasoning, and complex coinductive arguments are difficult to verify without a formalisation or a tedious reformulation.

Induction is dual to coinduction and it has dual lattice-theoretic and category-theoretic formulations, but informal proofs by induction normally follow an "operational" understanding of how to apply the inductive hypothesis: an argument to the inductive hypothesis must decrease in an appropriate sense. This informal understanding is reflected in Coq's induction principles and associated tactics. We propose a formal coinduction principle based on a dual (in an informal sense) "operational" understanding of how to use the coinductive hypothesis: the result must increase in an appropriate sense. This principle overcomes a weakness of Coq's current setup, where proofs built automatically by run-of-the-mill tactics may later be rejected by the type-checker on the grounds that they are not guarded.

A reader familiar with research in coinduction will probably notice a similarity between our first coinduction principle and some prior work, e.g., the principle from [14, 4.10] or the work on sized types [3, 2, 17, 1, 11] (see Remark 3.1). A contribution of this paper is to show that a principle of this kind may, to some extent, be already implemented in Coq's type theory, with the proofs translated directly to guarded Coq proof terms. From this point of view, Coq's guardedness criterion turns out to essentially be a syntactic description of the shape of normal forms of proofs obtainable using our principle. Gimenez [10, Theorem 8] already showed that his guardedness criterion is equivalent, in terms of definable functions, to corecursors in the style of [15, 9], but these are not convenient to use directly.

We also propose a second coinduction principle which extends the first one to allow for direct coinductive proofs of statements with existential quantifiers and multiple coinductive predicates in the conclusion.

The first coinduction principle may be implemented in Coq relatively seamlessly, with only small restrictions of limited practical significance. The situation is less satisfactory with the second principle. Significantly stronger restrictions are required, and the theoretical guarantees are weak. Nonetheless, the implementation is still useful. It covers a common pattern of proofs of existential statements that occur, e.g., in proofs about infinitary lambda-calculus [6]. Moreover, the difficulties with the implementation of the second principle seem to be caused by the limitations of Coq's type theory rather than by some more fundamental problems (see Remark 4.10).

The Paco library [12] achieves similar practical objectives to the first coinduction principle from our Coq plugin, but its methods are orthogonal to ours. It is based on parameterised coinduction – an extension of the common lattice-theoretic coinduction principle. It replaces Coq's `cofix` and requires the user to reformulate the definitions of their coinductive predicates using constructs from the library. In contrast, our approach is to translate the proofs obtained with our principle directly to guarded Coq proofs, which does not require any reformulation of the coinductive predicates. The translation approach has some practical disadvantages (e.g. Coq still wastes time on doing the guardedness checks), but our contribution is more in proposing a principle which may be considered an approximate semantic counterpart to Coq's syntactic guardedness check, thus opening an interesting line for future work.

Our principles are partly inspired by the explanations in [7] of how to elaborate proofs by coinduction to non-coinductive proofs in set theory.

2 Informal description

In this section, we informally state two coinduction principles and illustrate their use with a few examples of coinductive proofs. In the rest of this paper, we investigate to what extent and under which assumptions the principles may be implemented in Coq. The purpose of this section is to give an informal, illustrative introduction.

A (co)inductive type is given by its constructors, presented as, e.g.,

$$\texttt{Stream}(A : *) : * := \texttt{cons} : A \to \texttt{Stream}\, A \to \texttt{Stream}\, A$$

where $*$ denotes the sort of types. Above A is a *parameter* and $* \to *$ is the *arity* of `Stream`. The types of constructors implicitly quantify over the parameters, i.e., the type of `cons` above is $\forall A : *. A \to \texttt{Stream}\, A \to \texttt{Stream}\, A$. In the presentation we leave the parameter A implicit. Intuitively, a coinductive type consist of all possibly infinite objects built using the constructors, while an inductive type consists only of the finite ones.

Statements (logical formulas) are represented by dependent types. (Co)inductive predicates are represented by dependent (co)inductive types, e.g., the coinductive type

$$\begin{aligned}&\texttt{EqSt}(A : *) : \texttt{Stream}\, A \to \texttt{Stream}\, A \to * :=\\ &\quad \texttt{eqst} : \forall x : A. \forall s_1, s_2 : \texttt{Stream}\, A.\\ &\qquad\qquad \texttt{EqSt}\, A\, s_1\, s_2 \to \texttt{EqSt}\, A\, (\texttt{cons}\, x\, s_1)\, (\texttt{cons}\, x\, s_2)\end{aligned}$$

defines equality (bisimilarity) on streams. We use the words "statement" and "predicate" when we want to emphasise the logical interpretation of dependent types.

To state the coinduction principles, for each coinductive type I we need to define two associated types: the red type I^r and the green type I^g. Here we only informally describe them. The types I^r and I^g have the same parameters and the same arity as I and satisfy the following two properties. Below, we assume $I\, s_1 \ldots s_k : *$.

- The red type I^r is a fresh type symbol such that any value in $Is_1 \dots s_k$ or in $I^g s_1 \dots s_k$ may be (implicitly) converted into a value in $I^r s_1 \dots s_k$.
- The green type I^g is an inductive type such that for every constructor

  $$c : \forall x_1 : \tau_1 \dots \forall x_n : \tau_n . I s_1 \dots s_k$$

  of I there is a corresponding green constructor

  $$c^g : \forall x_1 : \tau_1[I^r/I] \dots \forall x_n : \tau_n[I^r/I] . I^g s_1 \dots s_k .$$

Nothing else is known about I^r and I^g. In particular, case analysis on values in $I^r s_1 \dots s_k$ is not possible. Note that any value v in $Is_1 \dots s_k$ may be converted into a value in $I^g s_1 \dots s_k$, by doing case analysis on v, in each case converting subterms of type $Is'_1 \dots s'_k$ to values in $I^r s'_1 \dots s'_k$, and then applying the corresponding green constructor.

▶ **Example 2.1.** For the type of streams `Stream` the green type $\mathtt{Stream}^g$ is:

$$\mathtt{Stream}^g (A : *) : * := \mathtt{cons}^g : A \to \mathtt{Stream}^r\ A \to \mathtt{Stream}^g\ A$$

For the bisimilarity `EqSt` on streams the green type $\mathtt{EqSt}^g$ is:

$$\begin{aligned} &\mathtt{EqSt}^g (A : *) : \mathtt{Stream}\ A \to \mathtt{Stream}\ A \to * := \\ &\quad \mathtt{eqst}^g : \forall x : A . \forall s_1, s_2 : \mathtt{Stream}\ A . \\ &\qquad\qquad \mathtt{EqSt}^r\ A\ s_1\ s_2 \to \mathtt{EqSt}^g\ A\ (\mathtt{cons}\ x\ s_1)\ (\mathtt{cons}\ x\ s_2) \end{aligned}$$

In a type $\varphi = \forall x_1 : \tau_1 \dots \forall x_n : \tau_n . I s_1 \dots s_k$ the type $Is_1 \dots s_k$ is the *target* and I is the *target (co)inductive predicate*. We write $\varphi(I')$ for φ with the target (co)inductive predicate replaced by I'. So $\varphi(I') = \forall x_1 : \tau_1 \dots \forall x_n : \tau_n . I' s_1 \dots s_k$. Note that the substitution of I' in $\varphi(I')$ leaves the occurrences of I in $\tau_1, \dots, \tau_n$ intact.

We restrict our coinduction principles to first-order statements and first-order (co)inductive types. First-order types will be defined precisely in the next section. Essentially, we need to disallow quantification over types and type constructors, excepting parameters of (co)inductive types. Also, for the actual implementation in Coq some further restrictions are needed, especially for the second principle.

▶ **Principle 1** (First coinduction principle – informal). *Let I be a coinductive type and $\varphi(I)$ a first-order statement. If $\varphi(I^r)$ implies $\varphi(I^g)$ then $\varphi(I)$ holds.*

The statement $\varphi(I^r)$ is the *coinductive hypothesis*, and $\varphi(I^g)$ is the *coinductive claim*. Hence, a proof by coinduction shows the coinductive claim under the assumption of the coinductive hypothesis. Intuitively, the red type I^r in the antecedent of the implication $\varphi(I^r) \to \varphi(I^g)$ ensures that a proof of $I^r s_1 \dots s_k$ obtained from the coinductive hypothesis cannot be analysed in any way, or used with previously proven lemmas about I. The green type I^g in the succedent ensures that a constructor must always be applied to a proof obtained from the coinductive hypothesis, i.e., it ensures productivity and prohibits concluding with the coinductive hypothesis directly. In this way, we ensure semantic guardedness of any proof of $\varphi(I^r) \to \varphi(I^g)$, i.e., the guarded use of the coinductive hypothesis $\varphi(I^r)$. Such a proof may then be translated into a guarded coinductive proof of $\varphi(I)$.

▶ **Example 2.2.** We show that the bisimilarity `EqSt` on streams is an equivalence relation. We write $s_1 \approx s_2$ instead of $\mathtt{EqSt}\ A\ s_1\ s_2$, and analogously with $\approx^r$ and $\approx^g$. We omit the type parameter A when irrelevant.

Using the first coinduction principle, we prove by coinduction that $\approx$ is reflexive. The coinductive hypothesis is: $s \approx^r s$ for all streams s. We need to show $s \approx^g s$ for all streams s. Let s be a stream. We have $s = \mathtt{cons}\, x\, s'$. By the coinductive hypothesis $s' \approx^r s'$. Hence $\mathtt{cons}\, x\, s' \approx^g \mathtt{cons}\, x\, s'$ by the definition of $\approx^g$.

We now prove by coinduction that $\approx$ is symmetric. The coinductive hypothesis is: for all streams s_1, s_2, if $s_1 \approx s_2$ then $s_2 \approx^r s_1$. Let s_1, s_2 be streams such that $s_1 \approx s_2$. Then $s_1 = \mathtt{cons}\, x\, s_1'$ and $s_2 = \mathtt{cons}\, x\, s_2'$ with $s_1' \approx s_2'$, by the definition of $\approx$. By the coinductive hypothesis $s_2' \approx^r s_1'$. Hence $\mathtt{cons}\, x\, s_2' \approx^g \mathtt{cons}\, x\, s_1'$ by the definition of $\approx^g$.

Finally, we prove transitivity of $\approx$ by coinduction. Let s_1, s_2, s_3 be streams such that $s_1 \approx s_2$ and $s_2 \approx s_3$. Then $s_1 = \mathtt{cons}\, x\, s_1'$, $s_2 = \mathtt{cons}\, x\, s_2'$ and $s_3 = \mathtt{cons}\, x\, s_3'$ with $s_1' \approx s_2'$ and $s_2' \approx s_3'$, by the definition of $\approx$. By the coinductive hypothesis $s_1' \approx^r s_3'$. Hence $\mathtt{cons}\, x\, s_1' \approx^g \mathtt{cons}\, x\, s_3'$ by the definition of $\approx^g$.

The first coinduction principle requires the target of the statement being proved to be a single coinductive predicate. This is in line with most previous work on coinduction. We will now informally state the second coinduction principle which enables direct coinductive proofs of statements with more complex targets.

Conjunction (product) $\wedge$, usually written in infix notation, may be defined by:

$$\wedge(A : *)(B : *) : * := \mathtt{conj} : A \to B \to A \wedge B$$

Existential quantification (dependent sum) may also be defined as an inductive type:

$$\mathtt{ex}(A : *)(P : A \to *) : * := \mathtt{ex\_intro} : \forall x : A. \forall p : Px. \mathtt{ex}AP$$

We usually write $\exists x : A.t$ instead of $\mathtt{ex}A(\lambda x : A.t)$.

We consider statements

$$\varphi = \forall x_1 : \tau_1 \ldots \forall x_m : \tau_m. \exists y : I t_1 \ldots t_p. I_1 s_1^1 \ldots s_{k_1}^1 y \wedge \ldots \wedge I_n s_1^n \ldots s_{k_n}^n y$$

where y does not occur in s_i^j. Thus the target is a single existential quantification on a value of a coinductive type followed by a conjunction of n coinductive predicates ($n \geq 1$) which depend on the existentially quantified variable. We write $\varphi(I'; I_1', \ldots, I_n')$ for φ with $I, I_1, \ldots, I_n$ in the target replaced by $I', I_1', \ldots, I_n'$ respectively (other occurrences of $I, I_1', \ldots, I_n'$ in $\tau_1, \ldots, \tau_m$ are not affected). For the sake of simplicity, we require y to always be the last argument of I_i, but the extension to the general case is straightforward.

The problem now is that changing the type of y will result in the whole statement being no longer well-typed. We thus introduce dependent red and green types by modifying the definitions of the red and the green types. We replace the last coinductive type in the arity with the corresponding red or green type, respectively, and for green types we also modify the types of the constructors accordingly. The definition of dependent red and green types works only for certain coinductive types. The precise conditions will be given later.

▶ **Example 2.3.** For the bisimilarity `EqSt` on streams the dependent red type has the type

$$\mathtt{EqSt}^r : \forall A : *. \mathtt{Stream}\, A \to \mathtt{Stream}^r A \to *$$

and the dependent green type $\mathtt{EqSt}^g$ is:

$$\begin{aligned} &\mathtt{EqSt}^g(A : *) : \mathtt{Stream}\, A \to \mathtt{Stream}^g A \to * := \\ &\quad \mathtt{eqst}^g : \forall x : A. \forall s_1 : \mathtt{Stream}\, A. \forall s_2^r : \mathtt{Stream}^r A. \\ &\qquad \mathtt{EqSt}^r\, A\, s_1\, s_2^r \to \mathtt{EqSt}^g\, A\, (\mathtt{cons}\, x\, s_1)\, (\mathtt{cons}^g\, x\, s_2^r) \end{aligned}$$

We can now informally state the second coinduction principle for statements with existential quantification in the target. When we write $\varphi(I^r; I_1^r, \ldots, I_n^r)$ we assume I^r is an (ordinary) red type and $I_1^r, \ldots, I_n^r$ are dependent red types. Analogously with $\varphi(I^g; I_1^g, \ldots, I_n^g)$.

▶ **Principle 2** (Second coinduction principle – informal). *Let $I, I_1, \ldots, I_n$ be coinductive types and $\varphi(I; I_1, \ldots, I_n)$ a first-order statement. If $\varphi(I^r; I_1^r, \ldots, I_n^r)$ implies $\varphi(I^g; I_1^g, \ldots, I_n^g)$ then $\varphi(I; I_1, \ldots, I_n)$ holds.*

▶ **Example 2.4.** Consider the following coinductive type of infinite terms.

$$\texttt{term} : * := C : \texttt{nat} \to \texttt{term} \mid A : \texttt{term} \to \texttt{term} \mid B : \texttt{term} \to \texttt{term} \to \texttt{term}$$

We define a parallel reduction relation $\Rightarrow$ on such terms, written in infix notation.

$$\begin{aligned} \Rightarrow : \texttt{term} \to \texttt{term} \to * \quad := \quad & r_C : \forall i : \texttt{nat}. Ci \Rightarrow Ci \\ & \mid r_A : \forall tt'. t \Rightarrow t' \to At \Rightarrow At' \\ & \mid r_B : \forall ss'tt'. s \Rightarrow s' \to t \Rightarrow t' \to Bst \Rightarrow Bs't' \\ & \mid r_{AB} : \forall tt_1t_2. t \Rightarrow t_1 \to t \Rightarrow t_2 \to At \Rightarrow Bt_1t_2 \end{aligned}$$

Using the second coinduction principle, we show that $\Rightarrow$ is confluent, i.e., if $s \Rightarrow t$ and $s \Rightarrow t'$ then there exists u such that $t \Rightarrow u$ and $t' \Rightarrow u$. The coinductive hypothesis is: for all terms s, t, t', if $s \Rightarrow t$ and $s \Rightarrow t'$ then there exists a red term u^r (i.e., an element of $\texttt{term}^r$) such that $t \Rightarrow^r u^r$ and $t' \Rightarrow^r u^r$.

Let s, t, t' be such that $s \Rightarrow t$ and $s \Rightarrow t'$. We need to show that there exists a green term u^g such that $t \Rightarrow^g u^g$ and $t' \Rightarrow^g u^g$. We do case analysis on the definitions of $s \Rightarrow t$ and $s \Rightarrow t'$. There are the following possibilities.

- $s = t = t' = Ci$. Then take $u^g = C^g i$.
- $s = As_1$ and $t = At_1$ and $t' = At_1'$ with $s_1 \Rightarrow t_1$ and $s_1 \Rightarrow t_1'$. By the coinductive hypothesis we obtain u^r (in $\texttt{term}^r$) such that $t_1 \Rightarrow^r u^r$ and $t_1' \Rightarrow^r u^r$. Take $u^g = A^g u^r$. Then $t = At_1 \Rightarrow^g A^g u^r$ and $t' = At_1' \Rightarrow^g A^g u^r$.
- $s = As_1$ and $t = At_1$ and $t' = Bt_1't_2'$ with $s_1 \Rightarrow t_1$ and $s_1 \Rightarrow t_1'$ and $s_1 \Rightarrow t_2'$. By the coinductive hypothesis we obtain u_1^r, u_2^r such that $t_1 \Rightarrow^r u_1^r$, $t_1' \Rightarrow^r u_1^r$, $t_1 \Rightarrow^r u_2^r$, $t_2' \Rightarrow^r u_2^r$. Take $u^g = B^g u_1^r u_2^r$. Then $s = As_1 \Rightarrow^g B^g u_1^r u_2^r$ and $t' = Bt_1't_2' \Rightarrow^g B^g u_1^r u_2^r$.
- Other cases are analogous to the ones already considered.

The rest of this paper is devoted to precisely stating the two coinduction principles in the logic of Coq, and investigating under which assumptions proofs using these principles may be automatically translated into guarded Coq proofs of the original statement.

3 Formal principles

In this section, we give a precise statement of the two coinduction principles. For this purpose, we define a type system in which our coinductive proofs will be represented. In the next section we define an extension of this type system which will be the target of our translations. Both systems are simplified subsets of the logic of Coq.

The language of our system consists of terms and (co)inductive declarations. First, we present the possible forms of terms together with a brief intuitive explanation of their meaning. The terms are essentially simplified terms of Coq. Below by t, s, u, τ, σ, etc., we denote terms, by c, c', etc., we denote constructors, and by x, y, z, etc., we denote variables. We use $\vec{t}$ for a sequence of terms $t_1 \ldots t_n$ of an unspecified length n, and analogously for a sequence of variables $\vec{x}$. For instance, $s\vec{y}$ stands for $sy_1 \ldots y_n$, where n is not important or implicit in the context. Analogously, we use $\lambda\vec{x} : \vec{\tau}.t$ for $\lambda x_1 : \tau_1.\lambda x_2 : \tau_2. \ldots \lambda x_n : \tau_n.t$, with n implicit or unspecified.

A *term* is a sort $*$, a constructor c, an inductive or a coinductive type I, an application $t_1 t_2$, an abstraction $\lambda x : t_1.t_2$, a dependent product $\forall x : t_1.t_2$, or a case expression $\texttt{case}(t, \lambda\vec{a} : \vec{\alpha}.\lambda x : I\vec{q}\vec{a}.\tau, \lambda\vec{x_1} : \vec{\sigma_1}.s_1 \mid \ldots \mid \lambda\vec{x_k} : \vec{\sigma_k}.s_k)$. In a case expression, t is the term matched on, I is a (co)inductive type, the type of t has the form $I\vec{q}\vec{u}$ where $\vec{q}$ are the values of the parameters, the type $\tau[\vec{u}/\vec{a}, t/x]$ is the return type, i.e., the type of the whole case expression, and $s_i[\vec{v}/\vec{x}]$ is the value of the case expression if the value of t is $c_i\vec{q}\vec{v}$.

For simplicity, we consider only one impredicative sort $*$ of types. If x does not occur free in t_1 then we abbreviate $\forall x : t_1.t_2$ to $t_1 \to t_2$.

A *(co)inductive declaration*

$$I(\vec{p} : \vec{\rho}) : \sigma := c_1 : \sigma_1 \mid \ldots \mid c_n : \sigma_n$$

declares a (co)inductive type I with *parameters* $\vec{p}$ and *arity* $\forall\vec{p} : \vec{\rho}.\sigma$ with n constructors $c_1, \ldots, c_n$ having types $\sigma_1, \ldots, \sigma_n$ respectively. We require:

- $\sigma = \forall\vec{a} : \vec{\alpha}.*$.
- $\sigma_i = \forall x_i^1 : \tau_i^1. \ldots \forall x_i^{k_i} : \tau_i^{k_i}.I\vec{p}\vec{u_i}$.
- I occurs only strictly positively in each σ_i, i.e., I does not occur in $\vec{u_i}$, and for each $j = 1, \ldots, k_i$ either I does not occur in τ_i^j or $\tau_i^j = \forall\vec{y} : \vec{\gamma}.I\vec{s}$ where I does not occur in $\vec{s}$ or $\vec{\gamma}$ ($\vec{s}, \vec{\gamma}$ depend on i, j).

The *arity* of a constructor c_i is $\forall\vec{p} : \vec{\rho}.\sigma_i$, denoted $c_i : \forall\vec{p} : \vec{\rho}.\sigma_i$. For the constructor $c_i : \forall\vec{p} : \vec{\rho}.\forall x_i^1 : \tau_i^1. \ldots \forall x_i^{k_i} : \tau_i^{k_i}.I\vec{p}\vec{u_i}$, the set $\mathcal{R}(c_i)$ of *recursive positions* is the set of all those j for which $\tau_i^j = \forall\vec{y} : \vec{\gamma}.I\vec{s}$.

We have the following reductions:

$$\begin{array}{rcl} (\lambda x : \tau.t)s & \to_\beta & t[s/x] \\ \texttt{case}(c_i\vec{p}\vec{v}, \lambda\vec{a} : \vec{\alpha}.\lambda x : I\vec{p}\vec{a}.\tau, \lambda\vec{x_1} : \vec{\tau_1}.s_1 \mid \ldots \mid \lambda\vec{x_k} : \vec{\tau_k}.s_k) & \to_\iota & s_i[\vec{v}/\vec{x_i}] \end{array}$$

An *environment* is a list of (co)inductive declarations. We write $I \in E$ if a declaration of a (co)inductive type I occurs in the environment E. Analogously, we write $(I : \tau) \in E$ and $(c : \tau) \in E$, if a declaration of I with arity τ occurs in E, or a constructor $c : \tau$ with arity τ in a declaration in E, respectively. A *context* Γ is a list of pairs $x : \tau$ with x a variable and τ a term. A *sort* is $*$ or $\Box$ (note that $\Box$ is not a term, but $*$ is). A *typing judgement* has the form $E; \Gamma \vdash t : \tau$ with t a term and τ a term or a sort. A term t is well-typed and has type τ in the context Γ and environment E if $E; \Gamma \vdash t : \tau$ may be derived using the rules from Figure 1. We denote an empty list by $\langle\rangle$.

In Figure 1 we assume s, s_1, s_2 are sorts. We also assume that the environment E is *well-formed*, which is defined inductively: an empty environment is well-formed, and an environment $E, I(\vec{p} : \vec{\rho}) : \tau := c_1 : \tau_1 \mid \ldots \mid c_n : \tau_n$ (denoted E, I) is well-formed if E is and:

- the constructors $c_1, \ldots, c_n$ are pairwise distinct and distinct from any constructors occurring in the declarations in E;
- $E; \langle\rangle \vdash \forall\vec{p} : \vec{\rho}.\tau : \Box$ and $E; I : \forall\vec{p} : \vec{\rho}.\tau, \vec{p} : \vec{\rho} \vdash \tau_i : *$.

When E, Γ are clear or irrelevant, we write $t : \tau$ instead of $E; \Gamma \vdash t : \tau$.

The type system is a subsystem of the Calculus of Inductive Constructions, so $\beta\iota$-reduction is confluent and strongly normalising on well-typed terms [21]. We usually implictly consider types to be in $\beta\iota$-normal form, without mentioning this every time. An η-expansion changes a term t of type $\forall x : \tau.\sigma$, which is not a λ-abstraction and is such that $x \notin \mathrm{FV}(t)$, into the term $\lambda x : \tau.tx$. The *η-long* form of a term is obtained by η-expanding as much as possible without creating β-redexes.

$$\frac{}{E;\langle\rangle \vdash * : \Box} \qquad \frac{(I:\tau)\in E}{E;\Gamma\vdash I:\tau} \qquad \frac{(c:\tau)\in E}{E;\Gamma\vdash c:\tau}$$

$$\frac{E;\Gamma\vdash A:s \quad x\notin\Gamma}{E;\Gamma,x:A\vdash x:A} \qquad \frac{E;\Gamma\vdash A:B \quad E;\Gamma\vdash C:s \quad x\notin\Gamma}{E;\Gamma,x:C\vdash A:B}$$

$$\frac{E;\Gamma\vdash F:\forall x:A.B \quad E;\Gamma\vdash t:A}{E;\Gamma\vdash Ft:B[t/x]} \qquad \frac{E;\Gamma,x:A\vdash t:B \quad E;\Gamma\vdash(\forall x:A.B):s}{E;\Gamma\vdash(\lambda x:A.t):\forall x:A.B}$$

$$\frac{E;\Gamma\vdash A:s_1 \quad E;\Gamma,x:A\vdash B:s_2}{E;\Gamma\vdash(\forall x:A.B):s_2} \qquad \frac{E;\Gamma\vdash A:B \quad E;\Gamma\vdash B':s \quad B=_{\beta\iota}B'}{E;\Gamma\vdash A:B'}$$

$$\frac{\begin{array}{c}E;\Gamma\vdash t:I\vec{q}\vec{u} \qquad E;\Gamma\vdash(\lambda\vec{a}:\vec{\alpha}.\lambda x:I\vec{q}\vec{a}.\tau):\forall\vec{a}:\vec{\alpha}.I\vec{q}\vec{a}\to *\\ (I(\vec{p}:\vec{\rho}):\forall\vec{a}:\vec{\alpha}.*:=c_1:\forall\vec{x_1}:\vec{\tau_1}.I\vec{p}\vec{u_1}\mid\ldots\mid c_k:\forall\vec{x_k}:\vec{\tau_k}.I\vec{p}\vec{u_k})\in E\\ E;\Gamma\vdash(\lambda\vec{x_i}:\vec{\sigma_i}.s_i):\forall\vec{x_i}:\vec{\sigma_i}.\tau[\vec{w_i}/\vec{a},c_i\vec{q}\vec{x_i}/x] \quad \vec{\sigma_i}=\vec{\tau_i}[\vec{q}/\vec{p}] \quad \vec{w_i}=\vec{u_i}[\vec{q}/\vec{p}]\end{array}}{E;\Gamma\vdash\mathtt{case}(t,\lambda\vec{a}:\vec{\alpha}.\lambda x:I\vec{q}\vec{a}.\tau,\lambda\vec{x_1}:\vec{\sigma_1}.s_1\mid\ldots\mid\lambda\vec{x_k}:\vec{\sigma_k}.s_k):\tau[\vec{u}/\vec{a},t/x]}$$

Figure 1 Typing rules.

A term τ is *first-order* if $*$ does not occur in it. A context Γ is first-order if for every $(x:\tau)\in\Gamma$ the type τ is first-order. A (co)inductive type

$$I(\vec{p}:\vec{\rho}):\sigma:=c_1:\sigma_1\mid\ldots\mid c_n:\sigma_n$$

is first-order if:

- $\sigma,\sigma_1,\ldots,\sigma_n$ are first-order;
- each parameter type ρ_i has the form $\forall\vec{x}:\vec{\tau}.*$ with all $\vec{\tau}$ first-order;
- if $\sigma_i=\forall x_1:\tau_1\ldots\forall x_m:\tau_m.I\vec{p}\vec{u}$ and I occurs in τ_i then $x_i\notin\mathrm{FV}(\tau_{i+1},\ldots,\tau_m,\vec{u})$.

An environment E is first-order if all (co)inductive types in E are. Note that we allow $*$ in the types of parameters of (co)inductive types.

Let $I(\vec{p}:\vec{\rho}):\forall\vec{a}:\vec{\alpha}.*:=c_1:\forall\vec{x_1}:\vec{\tau_1}.I\vec{p}\vec{u_1}\mid\ldots\mid c_k:\forall\vec{x_k}:\vec{\tau_k}.I\vec{p}\vec{u_k}$ be a coinductive declaration. The *red type declaration* $\mathrm{Decl}^r(I)$ for I is

$$I^r:\forall\vec{p}:\vec{\rho}.\forall\vec{a}:\vec{\alpha}.*,\quad \iota_I:\forall\vec{p}:\vec{\rho}.\forall\vec{a}:\vec{\alpha}.I\vec{p}\vec{a}\to I^r\vec{p}\vec{a},\quad \iota_I^g:\forall\vec{p}:\vec{\rho}.\forall\vec{a}:\vec{\alpha}.I^g\vec{p}\vec{a}\to I^r\vec{p}\vec{a}.$$

The *green type declaration* $\mathrm{Decl}^g(I)$ for I is

$$I^g(I^r:\tau_{I^r})(\vec{p}:\vec{\rho}):\forall\vec{a}:\vec{\alpha}.*:=c_1^g:\forall\vec{x_1}:\vec{\tau_1}[I^r/I].I^gI^r\vec{p}\vec{u_1}\mid\ldots\mid c_k^g:\forall\vec{x_k}:\vec{\tau_k}[I^r/I].I^gI^r\vec{p}\vec{u_k}$$

where $\tau_{I^r}=\forall\vec{p}:\vec{\rho}.\forall\vec{a}:\vec{\alpha}.*$ is the arity of the red type I^r. The type I is *admissible for* $E;\Gamma$ if $I^r,\iota_I,\iota_I^g\notin\Gamma$ and $I^g,c_1^g,\ldots,c_k^g\notin E$. Note that I^g need not be first-order, because τ_{I^r} might not have the required form for a parameter type.

We assume two new term forms: $\mathtt{cofix}_1(t)$ and $\mathtt{cofix}_2(t)$.

▶ **Principle 1** (First coinduction principle). *Let $\varphi=\forall\vec{x}:\vec{\tau}.z\vec{u}$ be a first-order type with $z\notin\mathrm{FV}(\vec{\tau},\vec{u})$ free. Let Γ be a first-order context and E a first-order environment. Let $(I:\forall\vec{p}:\vec{\rho}.\forall\vec{a}:\vec{\alpha}.*)\in E$ be a coinductive type admissible for $E;\Gamma$. If*

$$E,\mathrm{Decl}^g(I);\Gamma,\mathrm{Decl}^r(I)\vdash t:\varphi[I^r/z]\to\varphi[I^gI^r/z]$$

then

$$E; \Gamma \vdash \texttt{cofix}_1(t') : \varphi[I/z]$$

where $t' = t[I/I^r, \texttt{id}/\iota_I, \texttt{id}/\iota_I^g, (\lambda I^r.I)/I^g, (\lambda I^r.c_1)/c_1^g, \ldots, (\lambda I^r.c_k)/c_k^g]$ *and* $\texttt{id} = \lambda\vec{p} : \vec{\rho}.\lambda\vec{a} : \vec{\alpha}.\lambda x : I\vec{p}\vec{a}.x$ *and* $c_1, \ldots, c_k$ *are the only constructors of* I*.*

The first coinduction principle could be simply added to our type theory as a typing rule. We conjecture that, even without the first-order restriction, the resulting system would be reasonable and enjoy logical consistency and strong normalisation. In this paper we do not study the meta-theoretical properties of such a system, leaving this for future work. Instead, we investigate to what extent the principle may be implemented in the existing type theory of Coq. It turns out that in addition to the assumptions already stated, we need only minor restrictions on the proof t which have limited practical significance.

▶ Remark 3.1. We believe that the first-order restriction is not necessary for the soundness of the first coinduction principle. It is necessary to enable a translation to guarded Coq proofs. If we allow quantification over types, then some proofs obtained using the principle are not directly translatable, but we believe them to be still valid. For instance, if $I : * := c : I \to I$ and $R : I \to * := r : \forall x : I.Rx \to R(cx)$ are coinductive types and the context contains $F : \forall A : *.A \to A$ then $\texttt{cofix}_1(\lambda f : \forall y.Ry.\lambda y.\texttt{case}(y, \lambda y.Ry, \lambda x.rx(F(Rx)(fx))))$ may be obtained using the first coinduction principle. This proof is not syntactically guarded, but seems valid. Since F is parametric in the type argument A, it cannot inspect its second argument in any way. Hence, the proof is semantically guarded.

In fact, when restricted to streams the first coinduction principle is essentially a degenerate case of the principle from [17] based on sized types. The two colors (red and green) may be seen as two sizes: with green the successor of red. In a proof by coinduction the size needs to increase from red to green. One could extend our principle by introducing an arbitrary number of "colors" corresponding to natural numbers. The resulting system would be very similar to systems based on sized types [3, 2].

The reader may check that the counterexamples to a more relaxed syntactic guardedness criterion from [10, p. 53] do not translate to our principle. Nonetheless, the interaction of the first coinduction principle with impredicative polymorphism and fixpoints is not obvious. We leave for future work the rigorous investigation of the general soundness of our principle without the first-order restriction.

We now proceed to state the second coinduction principle. For this purpose, we need to introduce the definitions of dependent red and green types, as indicated in Section 2. We consider a coinductive type I with the declaration

$$I(\vec{p} : \vec{\rho}) : \forall\vec{a} : \vec{\alpha}.\forall b : J\vec{w}.* := c_1 : \forall\vec{x_1} : \vec{\tau_1}.I\vec{p}\vec{u_1}(d_1\vec{w_1}) \mid \ldots \mid c_k : \forall\vec{x_k} : \vec{\tau_k}.I\vec{p}\vec{u_k}(d_k\vec{w_k})$$

such that each d_i is a constructor of the coinductive type J, and if I occurs in τ_i^j then $\tau_i^j = I\vec{u}v$ and $v = x$ is a variable. We define $\mathrm{Var}_i(I)$ as the set of all variables which appear as the last argument to some occurrence of I in $\vec{\tau_i}$.

A coinductive type I of the above form is *J-admissible* if:

- for every $x \in \mathrm{Var}_i(I)$: x can only occur in $\vec{\tau_i}$ as the last argument of some occurrence of I in $\vec{\tau_i}$, and $x \notin \mathrm{FV}(\vec{u_i})$.
- if $d_i : \forall\vec{y} : \vec{\sigma}.J\vec{t}$ then for every j either $w_i^j = x \in \mathrm{Var}_i(I)$ and $\sigma_j = J\vec{s_j}$, or w_i^j does not contain any variables from $\mathrm{Var}_i(I)$ and σ_j does not contain J.

▶ **Example 3.2.** Let $I : * := c : I \to I$. The type $R : I \to I \to * := r : \forall x, y : I.Rxy \to R(cx)(cy)$ is I-admissible, but $R_1 : I \to I \to * := r_1 : \forall x, y : I.R_1xy \to R_1(cx)y$ and $R_2 : I \to I \to * := r_2 : \forall x, y : I.R_2xy \to R_2(cy)(cy)$ and $R_3 : I \to I \to * := r_3 : \forall x, y : I.R_3yy \to R_3(cx)(cy)$ are not.

The *dependent red type declaration* $\mathrm{Decl}^r_d(I)$ for I is:

$$I^r : \forall \vec{p} : \vec{\rho}.\forall \vec{a} : \vec{\alpha}.\forall b : J^r\vec{w}.*, \quad \iota_I : \forall \vec{p} : \vec{\rho}.\forall \vec{a} : \vec{\alpha}.\forall b : J\vec{w}.I\vec{p}\vec{a}b \to I^r\vec{p}\vec{a}(\iota_J\vec{w}),$$
$$\iota^g_I : \forall \vec{p} : \vec{\rho}.\forall \vec{a} : \vec{\alpha}.\forall b : J^gJ^r\vec{w}.I^gJ^rI^r\vec{p}\vec{a}b \to I^r\vec{p}\vec{a}(\iota^g_J\vec{w}).$$

The *dependent green type declaration* $\mathrm{Decl}^g_d(I)$ for I is:

$$I^g(J^r : \tau_{J^r})(I^r : \tau_{I^r})(\vec{p} : \vec{\rho}) : \forall \vec{a} : \vec{\alpha}.\forall b : J^gJ^r\vec{w}.* :=$$
$$c^g_1 : \forall \vec{x_1} : \vec{\sigma_1}.I^gJ^rI^r\vec{u_1}(d^g_1J^r\vec{w_1}) \mid \ldots \mid c^g_k : \forall \vec{x_k} : \vec{\sigma_k}.I^gJ^rI^r\vec{u_k}(d^g_kJ^r\vec{w_k})$$

where:

- $\sigma^j_i = \tau^j_i[I^r/I]$ if $x^j_i \notin \mathrm{Var}_i(I)$;
- $\sigma^j_i = J^r\vec{v}$ if $x^j_i \in \mathrm{Var}_i(I)$ and $\tau^j_i = J\vec{v}$;
- τ_{J^r} is the arity of the non-dependent red type J^r, and τ_{I^r} is the arity of the dependent red type I^r.

If I is J-admissible then I^g is well-formed.

▶ Remark 3.3. The definition of dependent green types could be relaxed at the cost of additional complexity. For example, we could parameterise the definition by ι_J and allow all constructors of the form $c_i : \forall \vec{x_i} : \vec{\tau_i}.I\vec{p}\vec{u_i}w$ where $I \notin \mathrm{FV}(\vec{\tau_i})$.

▶ **Principle 2** (Second coinduction principle). *Let $\varphi = \forall \vec{x} : \vec{\tau}.\exists y : z\vec{u}.z_1\vec{u_1}y \wedge \ldots \wedge z_n\vec{u_n}y$ be a first-order type with $z, z_1, \ldots, z_n \notin \mathrm{FV}(\vec{\tau}, \vec{u}, \vec{u_1}, \ldots, \vec{u_n})$ free. Let Γ be a first-order context and E a first-order environment. Let $I, I_1, \ldots, I_n \in E$ be coinductive types admissible for $E; \Gamma$, and such that $I_1, \ldots, I_n$ are I-admissible. If*

$$E, \mathrm{Decl}^g(I), \mathrm{Decl}^g_d(I_1), \ldots, \mathrm{Decl}^g_d(I_n); \Gamma, \mathrm{Decl}^r(I), \mathrm{Decl}^r_d(I_1), \ldots, \mathrm{Decl}^r_d(I_n) \vdash$$
$$t : \varphi[I^r/z, I^r_1/z_1, \ldots, I^r_n/z_n] \to \varphi[I^gI^r/z, I^g_1I^r_1I^r_1/z_1, \ldots, I^g_nI^r_nI^r_n/z_n]$$

then $E; \Gamma \vdash \mathtt{cofix}_2(t') : \varphi[I/z, I_1/z_1, \ldots, I_n/z_n]$ where

$$t' = t[I/I^r, \mathtt{id}/\iota_I, \mathtt{id}/\iota^g_I, (\lambda I^r.I)/I^g,$$
$$I_1/I^r_1, \mathtt{id}_1/\iota_{I_1}, \mathtt{id}_1/\iota^g_{I_1}, (\lambda I^rI^r_1.I_1)/I^g_1, \ldots, I_n/I^r_n, \mathtt{id}_n/\iota_{I_n}, \mathtt{id}_n/\iota^g_{I_n}, (\lambda I^rI^r_n.I_n)/I^g_n,$$
$$(\lambda I^r.c_1)/c^g_1, \ldots, (\lambda I^r.c_k)/c^g_k, (\lambda I^rI^r_1.c_{1,1})/c^g_{1,1}, \ldots, (\lambda I^rI^r_1.c_{k,1})/c^g_{k_1,1}, \ldots,$$
$$(\lambda I^rI^r_n.c_{1,n})/c^g_{1,n}, \ldots, (\lambda I^rI^r_n.c_{k_n,n})/c^g_{k_n,n}]$$

and $\mathtt{id}, \mathtt{id}_1, \ldots, \mathtt{id}_n$ are functions of appropriate types which return their last argument, and $c_1, \ldots, c_k$ are the only constructors of I, and $c_{1,j}, \ldots, c_{k_j,j}$ are the only constructors of I_j.

Above we assume all $I^r_1, \ldots, I^r_n$ to be distinct, even if some of the $I_1, \ldots, I_n$ are identical. This weakens the principle slightly in comparison to its informal presentation in Section 2. The types $\exists y : A.B$ and $A \wedge B$ are defined like in Section 2.

As in Section 2, for simplicity we allow only one existential quantifier and we require the existential variable to always be the last argument to a coinductive predicate. The extension to the general case is straightforward but tedious.

4 Proof translations

In this section, we define the two translations which map proofs that use our principles into guarded Coq proofs. The target type system of the translations is the system of the previous section extended with `cofix`.

▶ **Definition 4.1.** We add a new term form to the terms from Section 3: if t is a term and I a coinductive type, then $\texttt{cofix}(\lambda f : \forall \vec{x} : \vec{\tau}.I\vec{u}.t)$ is a term. We extend the type system from Section 3 by the reduction rule

$$\begin{aligned}&\texttt{case}(\texttt{cofix}(\lambda f : \forall \vec{x} : \vec{\tau}.I\vec{u}.t), r, s_1 \mid \ldots \mid s_k) \to_\iota \\ &\qquad \texttt{case}(t[\texttt{cofix}(\lambda f : \forall \vec{x} : \vec{\tau}.I\vec{u}.t)/f], r, s_1 \mid \ldots \mid s_k)\end{aligned}$$

and the typing rule

$$\frac{E; \Gamma \vdash (\lambda f : \forall \vec{x} : \vec{\tau}.I\vec{u}.t) : (\forall \vec{x} : \vec{\tau}.I\vec{u}) \to (\forall \vec{x} : \vec{\tau}.I\vec{u}) \quad \mathcal{G}(f,t)}{E; \Gamma \vdash \texttt{cofix}(\lambda f : \forall \vec{x} : \vec{\tau}.I\vec{u}.t) : \forall \vec{x} : \vec{\tau}.I\vec{u}}$$

where I is a coinductive type, and $\mathcal{G}(f,t)$ states that f is guarded in t, as defined below.

Following [10], we define two predicates $\mathcal{G}_h(f,t)$ for $h = 0, 1$. The predicate $\mathcal{G}_h(f,t)$ holds if one of the following is satisfied:

- $t = \lambda x : \tau.t'$ and $f \notin \mathrm{FV}(\tau)$ and $\mathcal{G}_h(f,t')$;
- $t = \texttt{case}(u, r, s_1 \mid \ldots \mid s_k)w_1 \ldots w_n$ with $n \geq 0$ and $f \notin \mathrm{FV}(u, r, w_1, \ldots, w_n)$ and $\mathcal{G}_h(f, s_i)$ for $i = 1, \ldots, k$;
- $t = ct_1 \ldots t_n$ and for $j = 1, \ldots, n$ we have: either $j \in \mathcal{R}(c)$ is a recusive position and $\mathcal{G}_1(f, t_j)$, or $f \notin \mathrm{FV}(t_j)$;
- $t = f\vec{u}$ and $h = 1$ and $f \notin \mathrm{FV}(\vec{u})$;
- $f \notin \mathrm{FV}(t)$.

We set $\mathcal{G} = \mathcal{G}_0$. If $\mathcal{G}(f,t)$ then f is *guarded in* t.

To avoid confusion, we denote the typability relation in the extended system by $\vdash_e$. We reserve $\vdash$ for the system without `cofix`.

The above syntactic guardedness criterion is more liberal than what is described in [10], but it is closer to the criterion actually implemented in Coq. In [10] terms of the form $\texttt{case}(u, r, s_1 \mid \ldots \mid s_k)w_1 \ldots w_n$ with $n \geq 1$ are not considered. Such terms are often generated by the `destruct` and `inversion` tactics, and Coq's guardedness checker does accept them.

▶ **Example 4.2.** Let $I : * := c : I \to I$. The variable f is guarded in $\texttt{case}(x, \lambda x : I.I \to I, \lambda y : I.\lambda a : I.c(fy))z$ but not in $\texttt{case}(x, \lambda x : I.I \to I, \lambda y : I.\lambda a : I.cy)(fz)$.

▶ **Definition 4.3** (The first translation)**.** Let $\varphi = \forall \vec{x} : \vec{\tau}.z\vec{u}$ be a first-order type with $z \notin \mathrm{FV}(\vec{\tau}, \vec{u})$ free. Let Γ be a first-order context and E a first-order environment. Let

$$I(\vec{p} : \vec{\rho}) : \forall \vec{a} : \vec{\alpha}.* := c_1 : \forall \vec{x_1} : \vec{\tau_1}.I\vec{p}\vec{u_1} \mid \ldots \mid c_k : \forall \vec{x_k} : \vec{\tau_k}.I\vec{p}\vec{u_k}$$

be a coinductive type in E admissible for $E; \Gamma$.

Assume $E, \mathrm{Decl}^g(I); \Gamma, \mathrm{Decl}^r(I) \vdash t : \varphi[I^r/z] \to \varphi[I^g I^r/z]$. The first translation of t, denoted $\mathrm{tr}_1(t)$, is defined as follows:

$$\mathrm{tr}_1(t) = \texttt{cofix}(t'[I/I^r, \texttt{id}/\iota_I, \texttt{id}/\iota_I^g, (\lambda I^r.I)/I^g, (\lambda I^r.c_1)/c_1^g, \ldots, (\lambda I^r.c_k)/c_k^g])$$

where t' is the η-long $\beta\iota$-normal form of t, and $\texttt{id} = \lambda \vec{p} : \vec{\rho}.\lambda \vec{a} : \vec{\alpha}.\lambda x : I\vec{p}\vec{a}.x$.

▶ **Example 4.4.** Let $I : * := c : I \to I$ and $R : I \to * := r : \forall x : I.Rx \to R(cx)$. Then a proof $t = \lambda f : (\forall x : I.R^r x).\lambda x : I.\mathtt{case}(x, \lambda x.R^g x, \lambda x'.r^g x'(fx'))$ for $\forall x : I.Rx$ gets translated to $\mathrm{tr}_1(t) = \mathtt{cofix}(\lambda f : (\forall x : I.Rx).\lambda x : I.\mathtt{case}(x, \lambda x.Rx, \lambda x'.rx'(fx')))$. For readability, we omit parameters to green types.

▶ **Definition 4.5.** A term t satisfies the *proper case restriction for* X, I if for every subterm of t of the form $\mathtt{case}(u, \lambda\vec{a} : \vec{\alpha}.x : J\vec{p}\vec{a}.\tau, s_1 \mid \ldots \mid s_k)$ the type τ is first-order, $J \neq I$, and X, I do not occur in τ. A term t satisfies the *weak case restriction for* X, I if:

- it satisfies the proper case restriction for X, I; or
- $t = \lambda x : \tau.t'$, and t' satisfies the weak case restriction for X, I; or
- $t = \mathtt{case}(u, \lambda\vec{a} : \vec{\alpha}.x : J\vec{p}\vec{a}.\forall\vec{y} : \vec{\beta}.\tau, s_1 \mid \ldots \mid s_k,)\vec{w}$, and $\tau, \vec{\beta}$ are first-order, and X, I do not occur in $\vec{\beta}$, and $J \neq I$, and $s_1, \ldots, s_k$ satisfy the weak case restriction for X, I, and $u, \vec{w}$ satisfy the proper case restriction for X, I.

The proper case restriction allows us to partially recover the subformula property for normal proofs of first-order statements, to the extent that we need it to conclude that the coinductive hypothesis does not occur in a proof of a statement with no occurrences of I^r. This is achieved in the following technical lemma, whose proof may be found in the appendix.

▶ **Lemma 4.6.** *Assume E is a first-order environment, Γ, Γ' is a first-order context, $X, I_X, f_1, \ldots, f_n$ do not occur in Γ, Γ', and u is an η-long $\beta\iota$-normal form satisfying the proper case restriction for X, I_X, and $E, I_X; \Gamma, X : \forall\vec{a} : \vec{\alpha}.*, f_1 : \forall\vec{x} : \vec{\sigma_1}.X\vec{v_1}, \ldots, f_n : \forall\vec{x} : \vec{\sigma_n}.X\vec{v_n}, \Gamma' \vdash u : \tau$.*

1. *If $\tau : *$ is a first-order type and $X, I_X, f_1, \ldots, f_n$ do not occur in τ, then $X, I_X, f_1, \ldots, f_n$ do not occur in u and u is first-order.*
2. *If $\tau = *$ and u is first-order and X, I_X do not occur in u, then $f_1, \ldots, f_n$ do not occur in u.*
3. *If $\tau = I\vec{h}$ and $I \neq I_X$ and either $u = \mathtt{case}(\ldots)\vec{w}$ or $u = y\vec{w}$, then τ is first-order and $X, I_X, f_1, \ldots, f_n$ do not occur in τ.*

▶ **Theorem 4.7.** *Under the assumptions of Definition 4.3, if the η-long $\beta\iota$-normal form of t additionally satisfies the weak case restriction for I^r, I^g, then $E; \Gamma \vdash_e \mathrm{tr}_1(t) : \varphi(I)$.*

Proof. We reason modulo $\beta\iota$-conversion in types. We also implicitly use standard meta-theoretical properties like the generation and subject reduction lemmas [4, 21]. The system is a simplification of the Calculus of Inductive Constructions, and these properties hold.

For any term u, by $\hat{u}$ we denote the η-long $\beta\iota$-normal form of

$$u[I/I^r, \mathtt{id}/\iota_I, \mathtt{id}/\iota_I^g, (\lambda I^r.I)/I^g, (\lambda I^r.c_1)/c_1^g, \ldots, (\lambda I^r.c_k)/c_k^g].$$

Without loss of generality assume t is in η-long $\beta\iota$-normal form. Then $\mathrm{tr}_1(t) = \mathtt{cofix}(\hat{t})$ and $t = \lambda f : \varphi(I^r/z).u$. It follows by induction on the derivation of $E, \mathrm{Decl}^g(I); \Gamma, \mathrm{Decl}^r(I) \vdash t : \varphi[I^r/z] \to \varphi[I^g I^r/z]$ that $E; \Gamma \vdash_e \hat{t} : \varphi[I/z] \to \varphi[I/z]$. Hence, it suffices to show that f is guarded in $\hat{u}$. Recall $\varphi = \forall\vec{x} : \vec{\tau}.z\vec{w}$ with $I^r, I^g, \iota_I, \iota_I^g, f$ not occurring in $\vec{\tau}, \vec{w}$ which are first-order. Because $\hat{u}$ is η-long, $u = \lambda\vec{x} : \vec{\tau}.r$. Hence $E, \mathrm{Decl}^g(I); \Gamma, \mathrm{Decl}^r(I), f : \varphi[I^r/z], \vec{x} : \vec{\tau} \vdash r : I^g I^r\vec{w}$. We need to show that $\mathcal{G}_0(f, \hat{r})$, i.e., f is guarded in $\hat{r}$.

By induction on u in η-long $\beta\iota$-normal form satisfying the weak case restriction for I^r, I^g, we show that if $E'; \Gamma' \vdash u : \sigma$ where $\sigma = I^g I^r\vec{w}'$ (resp. $\sigma = I^r\vec{w}'$), and $E' = E, \mathrm{Decl}^g(I)$, and $\Gamma' = \Gamma, \mathrm{Decl}^r(I), f : \varphi[I^r/z], \vec{x} : \vec{\tau}'$, and $\vec{\tau}', \vec{w}'$ are first-order, and $I^r, I^g, \iota_I, \iota_I^g, f$ do not occur in $\vec{\tau}', \vec{w}'$, then $\mathcal{G}_0(f, \hat{u})$ (resp. $\mathcal{G}_1(f, \hat{u})$).

First, assume $\sigma = I^g I^r\vec{w}'$. We consider possible forms of u.

- $u = xu_1 \dots u_n$. This is impossible, because for no $(x : \tau) \in \Gamma'$ the type τ has I^g or a bound variable at the head of the target.
- $u = cI^r q_1 \dots q_m u_1 \dots u_n$ $(m, n \geq 0)$ where $q_1, \dots, q_m$ are the parameters. Then $c : \forall I^r : \tau_{I^r}.\forall \vec{p} : \vec{\rho}.\forall \vec{x} : \vec{\gamma}.I^g I^r \vec{p} \vec{v}$, and $I^r \notin \mathrm{FV}(\rho)$, and $I^g, \iota_I, \iota_I^g, f$ do not occur in $\vec{\rho}, \vec{\gamma}$, and for each $i = 1, \dots, n$ either $I^r \notin \mathrm{FV}(\gamma_i)$ or $\gamma_i = \forall \vec{y} : \vec{\alpha_i}.I^r \vec{r_i}$ and $i \in \mathcal{R}(c)$ is a recursive position. Since $q_1, \dots, q_m$ occur in $\vec{w}$, $I^r, I^g, \iota_I, \iota_I^g, f$ do not occur in $q_1, \dots, q_m$, and thus $f \notin \mathrm{FV}(\hat{q_1}, \dots, \hat{q_m})$. Also $q_1, \dots, q_m$ are first-order, because $\vec{w}$ is. Let $\gamma_1' = \gamma_1[q_1/p_1] \dots [q_m/p_m]$. Then $I^g, \iota_I, \iota_I^g, f$ do not occur in γ_1', and γ_1' is first-order, and $E'; \Gamma' \vdash u_1 : \gamma_1'$. If $I^r \notin \mathrm{FV}(\gamma_1')$ then $I^r, I^g, \iota_I, \iota_I^g, f$ do not occur in u_1 and u_1 is first-order by Lemma 4.6. So $f \notin \mathrm{FV}(\hat{u_1})$. Otherwise $\gamma_1' = \forall \vec{y} : \vec{\alpha}.I^r \vec{r}$ with $\vec{\alpha}, \vec{r}$ first-order and $I^r, I^g, \iota_I, \iota_I^g, f$ do not occur in $\vec{\alpha}, \vec{r}$, and $1 \in \mathcal{R}(c)$ is a recursive position. Let $\Gamma'' = \Gamma', \vec{y} : \vec{\alpha}$. Because u_1 is η-long, $u_1 = \lambda \vec{y} : \vec{\alpha}.u_1'$ with $E'; \Gamma'' \vdash u_1' : I^r \vec{r}$. By the inductive hypothesis $\mathcal{G}_1(f, \hat{u_1}')$, so $\mathcal{G}_1(f, \hat{u_1})$ because $f \notin \mathrm{FV}(\vec{\alpha})$. Also, x_1 does not occur in $\gamma_2, \dots, \gamma_n$, because I is first-order. Hence, in any case, $\gamma_2' = \gamma_2[q_1/p_1] \dots [q_m/p_m][u_1/x_1]$ is first-order, and $I^g, \iota_I, \iota_I^g, f$ do not occur in γ_2', and $E'; \Gamma' \vdash u_2 : \gamma_2'$. Continuing this argument for $u_2, u_3, \dots, u_n$ we conclude that for each $i = 1, \dots, n$ either $f \notin \mathrm{FV}(\hat{u_i})$, or $\mathcal{G}_1(f, \hat{u_i})$ and $i \in \mathcal{R}(c)$. Since also $f \notin \mathrm{FV}(\hat{q_j})$ for $j = 1, \dots, m$, we conclude that $\mathcal{G}_0(f, \hat{u})$.
- $u = \mathtt{case}(t, r, s_1 \mid \dots \mid s_k) w_1 \dots w_n$ $(n \geq 0)$. Then $E', \Gamma' \vdash t : J \vec{p} \vec{q}$ and either $t = \mathtt{case}(\dots)\vec{h}$ or $t = y\vec{h}$. By the weak case restriction $J \neq I^g$ and t satisfies the proper case restriction. Hence, by Lemma 4.6, $J \vec{p} \vec{q}$ is first-order and $I^r, I^g, \iota_I, \iota_I^g, f$ do not occur in it. Using Lemma 4.6 again, we conclude that $I^r, I^g, \iota_I, \iota_I^g, f$ do not occur in t and t is first-order. Then by the weak case restriction the type $r \vec{q} t =_{\beta\iota} \forall \vec{x} : \vec{\beta}.\zeta$ of $\mathtt{case}(t, r, s_1 \mid \dots \mid s_k)$ must be first-order with I^r, I^g not occurring in $\vec{\beta}$. By point 2 in Lemma 4.6 also ι_I, ι_I^g, f do not occur in $\vec{\beta}$. Now using point 1 of Lemma 4.6 we conclude that $I^r, I^g, \iota_I, \iota_I^g, f$ do not occur in $\vec{w}$ and $\vec{w}$ are first-order, by an argument analogous to the one used in Lemma 4.6 for the case $u = xu_1 \dots u_m$ in the proof of point 1. To sum up, what we have shown so far implies $\mathcal{G}_0(f, \hat{t})$, $\mathcal{G}_0(f, \hat{r})$ and $\mathcal{G}_0(f, \hat{w_i})$ for $i = 1, \dots, n$. It remains to show $\mathcal{G}_0(f, \hat{s_i})$ for $i = 1, \dots, k$. Let the declaration of J be

  $$J(\vec{y} : \vec{\rho}) : \forall \vec{a} : \vec{\alpha}.* := c_1 : \forall \vec{x_1} : \vec{\tau_1}.J \vec{y} \vec{v_1} \mid \dots \mid c_k : \forall \vec{x_k} : \vec{\tau_k}.J \vec{y} \vec{v_k}$$

  We have $E', \Gamma' \vdash s_i : \xi$ where $\xi =_{\beta\iota} \forall \vec{x_i} : \vec{\tau_i}[\vec{p}/\vec{y}].r \vec{v_i}[\vec{p}/\vec{y}](c_i \vec{p} \vec{x_i}) =_{\beta\iota} \forall \vec{x_i} : \vec{\tau_i}[\vec{p}/\vec{y}].\forall \vec{b} : \vec{\beta_1}.I^g I^r \vec{v}$ (see Figure 1). Because $\vec{\tau}, \vec{v_i}, \vec{p}, \vec{\beta_1}$ are first-order and $I^r, I^g, \iota_I, \iota_I^g, f$ do not occur in them, also $\vec{\tau_i}[\vec{p}/\vec{y}], \vec{\beta_1}$ are first-order and $I^r, I^g, \iota_I, \iota_I^g, f$ do not occur in them. Also $\vec{v} = \vec{v_0}[\vec{v_i}[\vec{p}/\vec{y}]/\vec{a}, (c_i \vec{p} \vec{x_i})/x]$. Because $\vec{w}'$ are substitution instances of $\vec{v_0}$, the terms $\vec{v_0}$ must be first-order and $I^r, I^g, \iota_I, \iota_I^g, f$ cannot occur in them. This implies that $\vec{v}$ are first-order and $I^r, I^g, \iota_I, \iota_I^g, f$ do not occur in them. Because s_i is in η-long $\beta\iota$-normal form, $s_i = \lambda \vec{b} : \vec{\beta_1}.s_i'$. Let $\Gamma'' = \Gamma', \vec{x_i} : \vec{\tau_i}[\vec{p}/\vec{y}], \vec{b} : \vec{\beta_1}$. We have $E'; \Gamma'' \vdash s_i' : I^g I^r \vec{v}$, and s_i' still satisfies the weak case restriction, and $\Gamma'', \vec{v}$ satisfy the requirements of the inductive hypothesis. Thus, by the inductive hypothesis we conclude $\mathcal{G}_0(f, \hat{s_i}')$. This shows $\mathcal{G}_0(f, \hat{s_i})$.

Now assume $\sigma = I^r \vec{w}'$. The proof proceeds as above, mutatis mutantdis, except that we have three additional cases.

- $u = fu_1 \dots u_n$. Then u satisfies the proper case restriction. Since $f : \forall \vec{x} : \vec{\tau}.I^r \vec{w}$ with $\vec{\tau}$ first-order and $I^r, I^g, \iota_I, \iota_I^g, f$ not occurring in $\vec{\tau}$, using Lemma 4.6 we may conclude that $f \notin \mathrm{FV}(u_1, \dots, u_n)$ by an inductive argument as in the case $u = xu_1 \dots u_m$ in the proof of Lemma 4.6. Hence $\mathcal{G}_1(f, \hat{u})$.
- $u = \iota_I u_1 \dots u_n$. The argument is then analogous to the case above, because $\iota_I : \forall \vec{p} : \vec{\rho}.\forall \vec{a} : \vec{\alpha}.I \vec{p} \vec{a} \to I^r \vec{p} \vec{a}$ with $\vec{\rho}, \vec{\alpha}$ first-order not containing $I^r, I^g, \iota_I, \iota_I^g, f$.

- $u = \iota^g_I u_1 \dots u_n u'$. We have $\iota^g_I : \forall \vec{p} : \vec{\rho}. \forall \vec{a} : \vec{\alpha}. I^g I^r \vec{p} \vec{a} \to I^r \vec{p} \vec{a}$ with $\vec{\rho}, \vec{\alpha}$ first-order not containing $I^r, I^g, \iota_I, \iota^g_I, f$. Like above, using Lemma 4.6 we conclude that $u_1, \dots, u_n$ are first-order and $I^g, I^r, \iota_I, \iota^g_I, f$ do not occur in them. Also $u' : I^g I^r u_1 \dots u_n$. Hence, by the inductive hypothesis $\mathcal{G}_0(f, \hat{u}')$, so $\mathcal{G}_1(f, \hat{u}')$. Thus $\mathcal{G}_1(f, \hat{u})$, because $\hat{u} = \hat{u}'$. ◀

▶ Remark 4.8. For the purposes of the above theorem, any lemmas used in the proof term must appear in the context Γ. Note that the theorem requires the context and the statement to be first-order, but not the proof term. This implies that if the statement is first-order and we recursively unfold the proofs of all lemmas, we obtain a proof term t which satisfies the requirements of the theorem in the empty context, even if the lemmas used in the proof term were not first-order; provided the weak case restriction holds for the η-long $\beta\iota$-normal form of t.

One important situation where this procedure fails is when using the setoid library for rewriting. Then the generated proof terms, after unfolding, often fail to satisfy the weak case restriction.

To ease working with equality on coinductive types, our plugin provides a `peek` tactic which forces reduction of a cofixpoint.

The idea with the second translation is to "split" a coinductive proof $t : \forall \vec{x} : \vec{\tau}. \exists y : I \vec{u}. I_1 \vec{u_1} y \wedge \dots \wedge I_n \vec{u_n} y$ into $n+1$ separate guarded proofs $t_0 : \forall \vec{x} : \vec{\tau}. I \vec{u}$ and $t_i : \forall \vec{x} : \vec{\tau}. I_i \vec{u_i}(t_0 \vec{x})$ for $i = 1, \dots, n$.

▶ **Definition 4.9** (The second translation)**.** Let $\varphi = \forall \vec{x} : \vec{\tau}. \exists y : z \vec{u}. z_1 \vec{u_1} y \wedge \dots \wedge z_n \vec{u_n} y$ be a first-order type with $z, z_1, \dots, z_n \notin \mathrm{FV}(\vec{\tau}, \vec{u}, \vec{u_1}, \dots, \vec{u_n})$ free. Let Γ be a first-order context and E a first-order environment. Let $I, I_1, \dots, I_n \in E$ be coinductive types admissible for $E; \Gamma$, and such that $I_1, \dots, I_n$ are I-admissible. Assume

$$E, \mathrm{Decl}^g(I), \mathrm{Decl}^g_d(I_1), \dots, \mathrm{Decl}^g_d(I_n); \Gamma, \mathrm{Decl}^r(I), \mathrm{Decl}^r_d(I_1), \dots, \mathrm{Decl}^r_d(I_n) \vdash$$
$$t : \varphi[I^r/z, I^r_1/z_1, \dots, I^r_n/z_n] \to \varphi[I^g I^r/z, I^g_1 I^r I^r_1/z_1, \dots, I^g_n I^r I^r_n/z_n].$$

The second translation of t, denoted $\mathrm{tr}_2(t)$, is defined as follows. We omit the parameters of `ex_intro` and `conj`.

1. We compute t' – the η-long $\beta\iota$-normal form of

$$\lambda f : \psi[I^r/z]. \lambda f_1 : \psi_1[I^r_1/z, f/y] \dots \lambda f_n : \psi_n[I^r_n/z, f/y].$$
$$t(\lambda \vec{x} : \vec{\tau}. \texttt{ex\_intro}(f \vec{x})(\texttt{conj}(f_1 \vec{x})(f_2 \vec{x})(\dots))).$$

where $\psi = \forall \vec{x} : \vec{\tau}. z \vec{u}$ and $\psi_i = \forall \vec{x} : \vec{\tau}. z \vec{u_i}(y \vec{x})$. In this way we "split" the coinductive hypothesis into $n+1$ hypotheses. Note that

$$t' : \forall f : \psi[I^r/z]. \psi_1[I^r_1/z, f/y] \to \dots \to \psi_n[I^r_n/z, f/y] \to$$
$$\varphi[I^g I^r/z, I^g_1 I^r I^r_1/z_1, \dots, I^g_n I^r I^r_n/z_n].$$

2. Inductively, for a term u we define $\mathrm{tr}^0_2(u)$, and for $i = 1, \dots, n$, a sequence of terms $\vec{w}$, and a term u, we define $\mathrm{tr}^i_2(\vec{w}; u)$.
   - If $u = \texttt{case}(x, \lambda x : J \vec{p}. \zeta, \lambda \vec{x_1} : \vec{\tau_1}. s_1 \mid \dots \mid \lambda \vec{x_k} : \vec{\tau_k}. s_k)$ where

$$\zeta = \exists y : I^g I^r \vec{v}. I^g_1 I^r I^r_1 \vec{v_1} y \wedge \dots \wedge I^g_n I^r I^r_n \vec{v_n} y$$

   and J is a (co)inductive type with no non-parameter arguments, and $c_1, \dots, c_k$ are the only constructors of J, then
   - $\mathrm{tr}^0_2(u) = \texttt{case}(x, \lambda x : J \vec{p}. I \vec{v}, \lambda \vec{x_1} : \vec{\tau_1}. \mathrm{tr}^0_2(s_1) \mid \dots \mid \lambda \vec{x_k} : \vec{\tau_k}. \mathrm{tr}^0_2(s_k))$.
   - $\mathrm{tr}^i_2(\vec{w}; u) = \texttt{case}(x, \lambda x : J \vec{p}. I_i \vec{v_i}(f \vec{w}), \lambda \vec{x_1} : \vec{\tau_1}. \mathrm{tr}^i_2(\vec{w}[c_1 \vec{p} \vec{x_1}/x]; s_1) \mid \dots \mid \lambda \vec{x_k} : \vec{\tau_k}. \mathrm{tr}^i_2(\vec{w}[c_k \vec{p} \vec{x_k}/x]; s_k))$ for $i > 0$.

- If $u = \texttt{ex\_intro}\, u_0\, (\texttt{conj}\, u_1\, (\texttt{conj}\, u_2\, (\ldots)))$ then
  - $\mathrm{tr}_2^0(u) = u_0[I/I^r, \texttt{id}/\iota_I, (\lambda I^r.I)/I^g, \ldots]$,
  - $\mathrm{tr}_2^i(\vec{w}; u) = u_i[I/I^r, \texttt{id}/\iota_I, (\lambda I^r.I)/I^g, \ldots]$,

  with the substitutions like in Principle 2.

  In other cases tr_2^i are undefined.
3. Since t' is in η-long $\beta\iota$-normal form,

$$t' = \lambda f : \psi[I^r/z].\lambda f_1 : \psi_1[I_1^r/z, f/y] \ldots \lambda f_n : \psi_n[I_n^r/z, f/y].\lambda \vec{x} : \vec{\tau}.t''.$$

   We define t_i for $i = 0, \ldots, n$ by:
   - $t_0 = \texttt{cofix}(\lambda f : \psi[I/z].\lambda \vec{x} : \vec{\tau}.\mathrm{tr}_2^0(t''))$,
   - $t_i = \texttt{cofix}(\lambda f_i : \psi_i[I_i/z, t_0/y].\lambda \vec{x} : \vec{\tau}.\mathrm{tr}_2^i(\vec{x}; t'')[t_0/f])$ for $i = 1, \ldots, n$.
4. Finally: $\mathrm{tr}_2(t) = \lambda \vec{x} : \vec{\tau}.\texttt{ex\_intro}(t_0\vec{x})(\texttt{conj}(t_1\vec{x})(\texttt{conj}(t_2\vec{x})(\ldots(\texttt{conj}(t_{n-1}\vec{x})(t_n\vec{x})))))$.

▶ Remark 4.10. The above translation is very restricted, essentially to proofs which do case analysis on variables followed by the use of the coinductive hypothesis. It is undefined for proof terms commonly generated by the `inversion` tactic. The problem here is that Coq's dependent matching "forgets" some equality information in the branches, which makes it difficult to automatically choose the arguments for the f function above in a way that satisfies the type-checker. More precisely, when $u : I\vec{q}\vec{w}$ and $c_i : \forall \vec{p} : \vec{\rho}.\forall \vec{x_i} : \vec{\tau_i}.I\vec{p}\vec{v_i}$ is the i-th constructor of I, the equalities $u = c_i\vec{p}\vec{v_i}$ and $v_i^j = w_i^j$ are not available to the Coq's type-checking algorithm when checking the branch s_i in $\texttt{case}(u, r, s_1 \mid \ldots \mid s_k)$ (see Figure 1). In practice, we allow a broader class of proof terms. In particular, we handle proofs commonly generated by one inversion (but not multiple nested inversions in general). The details of this are very tedious and not particularly illuminating, so we do not describe them here. The restrictions in the actual implementation, while a bit ad-hoc and still significant, are weak enough to allow for reasonably convenient usage. Especially the restriction on nested inversions can usually be easily worked around. See Example 5.2.

▶ **Example 4.11.** Let $I : * := c : I \to I$ and $R : I \to I \to * := r : \forall x, y : I.Rxy \to R(cx)(cy)$ be coinductive types. Recall $I^r : *$, $R^r : I \to I^r \to *$, $I^g : * := c^g : I^r \to I^g$, $R^g : I \to I^g \to * := r^g : \forall x : I.\forall y : I^r.R^r xy \to R^g(cx)(c^g y)$. For readability, we omit the parameters to the green types. Consider the term

$$\begin{aligned} t \quad = \quad & \lambda f : (\forall x : I.\exists y : I^r.R^r xy).\lambda x : I.\texttt{case}(x, \lambda x.\exists y : I^g.R^g xy, \\ & \lambda x'.\texttt{case}(fx', \exists y : I^g.R^g(cx')y, \lambda y' : I^r.\lambda h : R^r x'y'.\texttt{ex\_intro}(c^g y')(r^g x'y'h))) \end{aligned}$$

which proves $\forall x : I.\exists y : I.Rxy$ by the second coinduction principle. After the first step of the second translation we obtain $t' = \lambda f : I \to I^r.\lambda f_1 : \forall x : I.R^r x(fx).\lambda x : I.t''$ where $t'' = \texttt{case}(x, \lambda x.\exists y : I^g.R^g xy, \lambda x'.\texttt{ex\_intro}(c^g(fx'))(r^g x'(fx')(f_1 x')))$. We have $\mathrm{tr}_2^0(t'') = \texttt{case}(x, \lambda x.I, \lambda x'.c(fx'))$ and $\mathrm{tr}_2^1(x; t'') = \texttt{case}(x, \lambda x.Rx(fx), \lambda x'.rx'(fx')(f_1 x'))$. Then $\mathrm{tr}_2(t) = \lambda x{:}I.\texttt{ex\_intro}(t_0 x)(t_1 x)$ with $t_0 = \texttt{cofix}\lambda f : I \to I.\lambda x{:}I.\mathrm{tr}_2^0(t'')$ and $t_1 = \texttt{cofix}\lambda f_1 : (\forall x : I.Rx(t_0 x)).\lambda x : I.\mathrm{tr}_2^1(x; t'')[t_0/f]$.

Note that in the example above $rx'(t_0 x')(f_1 x') : R(cx')(c(t_0 x'))$, but the branch should have type $R(cx')(t_0(cx'))$. We thus relax the ι-reduction for `cofix` to: $\texttt{cofix}(\lambda f.t) \to_\iota t[\texttt{cofix}(\lambda f.t)/f]$. Then $t_0(cx') =_{\beta\iota} c(t_0 x')$ and $\mathrm{tr}_2(t)$ type-checks. The typing relation of the system thus modified is denoted by $\vdash_{e+}$. This allows us to prove the theorem below, but makes type-checking undecidable. In practice, the implemented translation inserts appropriate equality proofs into the proof term to make it type-check. The details are again quite tedious and not very interesting.

▶ **Definition 4.12.** A term satisfies the *strong case restriction* if it does not contain any subterms $\mathtt{case}(\mathtt{case}(u, r', t_1 \mid \ldots \mid t_m)\vec{w}, r, s_1 \mid \ldots \mid s_k)$ or $\mathtt{case}(u, r, s_1 \mid \ldots \mid s_k)w_1 \ldots w_n$ with $n \geq 1$.

▶ **Theorem 4.13.** *Under the assumptions of Definition 4.9, if the η-long $\beta\iota$-normal form of t additionally satisfies the strong case restriction, and $\mathrm{tr}_2(t)$ is defined, and $E; \Gamma, f : \psi[I/z], f_i : \psi_i[I_i/z, t_0/y], \vec{x} : \vec{\tau} \vdash_{e+} \mathrm{tr}_2^i(\vec{x}; t'')[t_0/f] : I_i\vec{u_i}(t_0\vec{x})$ for $i = 1, \ldots, n$ (with $\psi, \psi_i, t_0, \ldots$ as in Definition 4.9), then $E; \Gamma \vdash_{e+} \mathrm{tr}_2(t) : \varphi(I; I_1, \ldots, I_n)$.*

Proof (sketch). The strong case restriction essentially recovers the subformula property for first-order statements, which allows us to show that the coinductive hypotheses do not appear in parts of the proof term whose types do not contain corresponding red or green types. The special form of the original proof term enforced by the second translation, and the typability assumptions for tr_2^i, guarantee that the result is well-typed. ◀

5 Coq plugin

We provide a proof-of-concept implementation of our principles in a Coq plugin (see the supplement material). The plugin introduces the `CoInduction` command which starts a proof by coinduction using one of our principles. The command defines the (dependent) green coinductive types, and adds the (dependent) red type declarations and the coinductive hypothesis to the context. At `Qed` the proof is translated to a guarded Coq proof. The coinduction principle is chosen automatically based on the form of the goal statement.

▶ **Example 5.1.** Using our plugin, the proofs from Example 2.2 may be formalised as follows.

```
CoInduction lem_refl : forall {A : Type} (s : Stream A), s == s.
Proof. ccrush. Qed.

CoInduction lem_sym :
   forall {A : Type} (s1 s2 : Stream A), s1 == s2 -> s2 == s1.
Proof. ccrush. Qed.

CoInduction lem_trans :
   forall {A : Type} (s1 s2 s3 : Stream A), s1 == s2 -> s2 == s3 -> s1 == s3.
Proof. destruct 1; ccrush. Qed.
```

The `ccrush` tactic is a generic proof search tactic, based on a tactic from CoqHammer [8]. Note that the user may just apply generic automated tactics without worrying too much about the guarded use of the coinductive hypothesis. In contrast, when using Coq's `cofix` directly, generic automated tactics are likely to use the coinductive hypothesis incorrectly so that the proof fails at `Qed`.

▶ **Example 5.2.** A direct formalisation of the confluence proof from Example 2.4 would require two nested inversions, and the second translation would fail at `Qed` (compare Remark 4.10). This may, however, be easily worked around by defining a predicate $\mathtt{Peak}\, s\, t\, t'$ which holds if $s \Rightarrow t$ and $s \Rightarrow t'$ do. We define all predicates into `Set` to get around Coq's restriction of case analysis on proofs. Note that for the proof of `lem_peak` below the first translation may be used, with no restrictions on nested destructions/inversions.

```
CoInductive Peak : term -> term -> term -> Set :=
| peak_C : forall i, Peak (C i) (C i) (C i)
| peak_A : forall s t t', Peak s t t' -> Peak (A s) (A t) (A t')
| peak_B : forall s t s1 t1 s2 t2, Peak s s1 s2 -> Peak t t1 t2 ->
                                   Peak (B s t) (B s1 t1) (B s2 t2)
| peak_AAB : forall s s' t1 t2, Peak s s' t1 -> Peak s s' t2 ->
                                Peak (A s) (A s') (B t1 t2)
| peak_ABA : forall s s' t1 t2, Peak s t1 s' -> Peak s t2 s' ->
                                Peak (A s) (B t1 t2) (A s')
| peak_ABB : forall s s1 s2 t1 t2, Peak s s1 t1 -> Peak s s2 t2 ->
                                   Peak (A s) (B s1 s2) (B t1 t2).

CoInduction lem_peak : forall s t t', s ==> t -> s ==> t' -> Peak s t t'.
Proof. destruct 1; inversion_clear 1; constructor; eauto. Qed.
```

Then the confluence proof looks as follows. It corresponds closely to the "pen-and-paper" proof presented in Example 2.4.

```
CoInduction lem_confl :
   forall s t t', Peak s t t' -> { s' & (t ==> s') * (t' ==> s') }.
Proof. intros s t t' H; inversion_clear H.
  - ccrush.
  - generalize (CH s0 t0 t'0 H0); intro.
    simp_hyps; eexists; split; constructor; eauto.
  - generalize (CH s0 s1 s2 H0); generalize (CH t0 t1 t2 H1); intros.
    simp_hyps; eexists; split; constructor; eauto.
  (...)
Qed.
```

Above `CH` refers to the coinductive hypothesis automatically introduced by `CoInduction`:

```
CH : forall s t t' : term, Peak s t t' ->
       {s' : term__r & Red__r__01 t s' * Red__r__00 t' s'}
```

At the beginning of the proof the goal is:

```
forall s t t' : term, Peak s t t' ->
   {s' : term__g term__r & Red__g__01 term__r Red__r__01 t s' *
                           Red__g__01 term__r Red__r__00 t' s'}
```

▶ **Example 5.3.** Using the first coinduction principle, we formalised most of the examples from [14]. The formalisation is in the `examples/practical.v` file in the plugin sources.

6 Conclusions and future work

We introduced two coinduction principles and corresponding proof translations which, under certain conditions, map proofs using our principles to guarded Coq proofs. In contrast to previous work on coinduction, the second principle allows to directly prove by coinduction statements with existential quantifiers and multiple coinductive predicates in the conclusion. The proof translations clarify the relationship between Coq's syntactic guardedness criterion and the shape of normal forms of proofs obtained using our principles. Implementating the first translation required only small restrictions on dependent matches occurring in proof

terms. While trying to implement the second translation, we encountered more difficulties, which necessitated introducing much heavier restrictions. These difficulties, however, do not seem to be fundamental, but rather stem from the limitations of Coq's type theory.

The restrictions on proof terms are needed because normal proofs in the Calculus of Inductive Constructions are not sufficiently normal in a proof-theoretical sense. And they cannot be normalised further using commutative conversions [20, Chapter 6], like in a natural-deduction system for first-order logic, because commutative conversions are not sound in general for dependent elimination with `case` as defined in the Calculus of Inductive Constructions. The lack of "good" normal forms is a consequence of the fact that not enough equality information is available to the type checker in the branches of dependent matches, which also makes it difficult to implement the second coinduction principle in full generality.

For a more complete implementation of the second coinduction principle, it would probably be better to use a target system with copatterns and sized types, or use negative coinductive types instead of positive ones. We leave this for future work. It also remains to investigate the properties of a type theory directly extended with our principles, and the effects of removing the first-order restriction.

References

1. A. Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006.
2. A. Abel and B. Pientka. Well-founded recursion with copatterns and sized types. *J. Funct. Program.*, 26:e2, 2016.
3. A. Abel, A. Vezzosi, and T. Winterhalter. Normalization by evaluation for sized dependent types. *PACMPL*, 1(ICFP):33:1–33:30, 2017.
4. H. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, volume 2, pages 118–310. Oxford University Press, 1992.
5. T. Coquand. Infinite Objects in Type Theory. In *TYPES 1993*, pages 62–78, 1993.
6. Ł. Czajka. A New Coinductive Confluence Proof for Infinitary Lambda Calculus. Submitted, 2018.
7. Ł. Czajka. Coinduction: an elementary approach. *arXiv*, 2019. `arXiv:1501.04354`.
8. Ł. Czajka and C. Kaliszyk. Hammer for Coq: Automation for Dependent Type Theory. *J. Autom. Reasoning*, 61(1-4):423–453, 2018.
9. H. Geuvers. Inductive and coinductive types with iteration and recursion. In *TYPES 1992*, pages 193–217, 1992.
10. E. Giménez. Codifying Guarded Definitions with Recursive Schemes. In *TYPES*, pages 39–59, 1994.
11. J. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *POPL 1996*, pages 410–423, 1996.
12. C.-K. Hur, G. Neis, D. Dreyer, and V. Vafeiadis. The power of parameterization in coinductive proof. In *POPL 2013*, pages 193–206, 2013.
13. B. Jacobs and J. Rutten. An introduction to (co)algebras and (co)induction. In *Advanced Topics in Bisimulation and Coinduction*, pages 38–99. Cambridge University Press, 2011.
14. D. Kozen and A. Silva. Practical coinduction. *Math. Struct. in Comp. Science*, 27(7):1132–1152, 2017.
15. N.P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51:159–172, 1991.
16. J. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, 2000.
17. J.L. Sacchini. Type-Based Productivity of Stream Definitions in the Calculus of Constructions. In *LICS 2013*, pages 233–242, 2013.

18 D. Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2012.

19 A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

20 A.S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, 1996.

21 B. Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Paris Diderot University, France, 1994.

Formalizing the Solution to the Cap Set Problem

Sander R. Dahmen
Department of Mathematics, Vrije Universiteit Amsterdam, The Netherlands
s.r.dahmen@vu.nl

Johannes Hölzl
Department of Computer Science, Vrije Universiteit Amsterdam, The Netherlands
johannes.hoelzl@posteo.de

Robert Y. Lewis
Department of Computer Science, Vrije Universiteit Amsterdam, The Netherlands
r.y.lewis@vu.nl

Abstract

In 2016, Ellenberg and Gijswijt established a new upper bound on the size of subsets of $\mathbb{F}_q^n$ with no three-term arithmetic progression. This problem has received much mathematical attention, particularly in the case $q = 3$, where it is commonly known as the *cap set problem*. Ellenberg and Gijswijt's proof was published in the *Annals of Mathematics* and is noteworthy for its clever use of elementary methods. This paper describes a formalization of this proof in the Lean proof assistant, including both the general result in $\mathbb{F}_q^n$ and concrete values for the case $q = 3$. We faithfully follow the pen and paper argument to construct the bound. Our work shows that (some) modern mathematics is within the range of proof assistants.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Type theory; Mathematics of computing → Number-theoretic computations; Computing methodologies → Combinatorial algorithms; Software and its engineering → Formal methods

Keywords and phrases formal proof, combinatorics, cap set problem, Lean

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.15

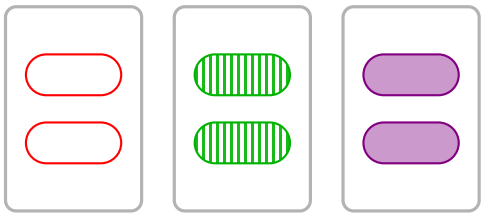
Supplement Material Links to our formalization and supporting documents are hosted at the URL `https://lean-forward.github.io/e-g/`.

Funding *Sander R. Dahmen*: NWO Vidi grant No. 639.032.613, New Diophantine Directions
Johannes Hölzl: ERC grant agreement No. 713999, Matryoshka
Robert Y. Lewis: ERC grant agreement No. 713999, Matryoshka

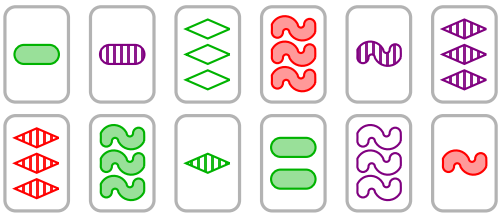
Acknowledgements We are grateful to the Lean `mathlib` maintainers and contributors on whose work this project is based. We thank Jeremy Avigad and Jasmin Blanchette for helpful comments on this paper, Dion Gijswijt for suggesting an improvement to our asymptotic bound argument, and Manuel Eberl for pointing us to related work.

1 Introduction

As proof assistants improve and their libraries grow, these tools are increasingly used to formalize results at the cutting edge of computer science. At some prestigious conferences such as *Principles of Programming Languages* (POPL), it is common for papers establishing new metatheoretical results about programming languages to be accompanied by formal proofs. In the field of mathematics, however, the picture looks very different. Even though early proof assistants were developed by and for mathematicians [10, 27], there are still very few mathematicians who use these tools in their work. With a small number of noteworthy exceptions (e.g. Gouëzel and Schur [21] and Hales, et al. [23]), no current work in pure



(a) A valid triple. Each card has the same shape and the same number of shapes. Each card has a different color and a different fill.



(b) A collection of twelve cards that contains no valid triple.

Figure 1 The cap set problem can be interpreted in the game Set, where it concerns an upper bound on the size of a collection of cards that contains no valid triple.

mathematics work gets formalized; most of the results formalized in papers at *Interactive Theorem Proving* (ITP) or *Certified Programs and Proofs* (CPP) have already made it into undergraduate or introductory graduate textbooks.

Researchers often point to the *depth* of mathematical theory to explain this difference. While programming language formalizations can be sprawling and difficult, they rarely depend on large background libraries, and often involve repetitive arguments that are amenable to automation. In comparison, mathematics builds upwards on centuries of earlier work, and one cannot formalize modern results without first formalizing the necessary foundation. The few existing formal developments of cutting-edge mathematics tend to focus on results that are difficult to verify by hand – justifying the effort needed to develop libraries – or fall in subfields of mathematics where the background theory is less intimidating.

The combinatorial proof described in this paper belongs in the latter category. Let G be an abelian group. A three-term *arithmetic progression* of elements of G is a sequence $a, a+g, a+g+g$ where $a, g \in G$ and g is nonzero. Let $r_3(G)$ denote the cardinality of a largest subset of G containing no three-term arithmetic progression. We will focus on the group $(\mathbb{Z}/3\mathbb{Z})^n = \{(a_1, \ldots, a_n) \mid a_i \in \{0, 1, 2\}\}$, where vector addition is pointwise and modulo 3; a subset of this group with no three-term arithmetic progression is known as a *cap set*. The *cap set problem* asks whether there is a constant $c < 3$ such that $r_3((\mathbb{Z}/3\mathbb{Z})^n)$ grows in n no faster than c^n.

Readers familiar with the card game Set (Figure 1) may understand the cap set problem in different terms. A card in Set has four features, where each feature has three possible values. (A card has one, two, or three copies of a shape; the shape is an oval, a diamond, or a squiggle; the shape is solid, striped, or empty; the shape is purple, red, or green.) A triple of cards is said to be *valid* if, for each feature, either all three cards have the same value or all three cards have different values. During game play, players search a collection of cards for valid triples. The number $r_3((\mathbb{Z}/3\mathbb{Z})^4)$ is the maximum size of a collection of distinct cards in which no valid triples can be found, and the cap set problem concerns the growth rate of this value as the number of features is increased.

The cap set problem is surprisingly difficult to analyze and has attracted attention over the past decades from leading combinatorialists. Croot, Lev, and Pach [9] solved a closely related problem in 2016. Building on their work, Ellenberg and Gijswijt soon showed that $r_3((\mathbb{Z}/3\mathbb{Z})^n)$ is $o(2.756^n)$, a major breakthrough. In fact, they proved a more general result about finite fields. Their 2017 paper in the *Annals of Mathematics* [18] is noteworthy in that the core of the proof does not use any complicated theoretical machinery. Rather, it relies on a clever shift of context, casting the problem in terms of polynomials of bounded degree.

While their final proof of the asymptotics does make use of relatively high-powered methods, Tao [30] and Zeilberger [33] indicate how these calculations can be made elementary. We also note that Tao [30] reformulates Ellenberg and Gijswijt's proof in a more symmetric way, using what is now called "slice rank." Although this is arguably a more natural way to express things, the underlying arguments are essentially the same.

This paper describes a formalization of Ellenberg and Gijswijt's argument, carried out in the Lean proof assistant. While unavoidably more verbose, our computation of an upper bound for $r_3((\mathbb{Z}/p\mathbb{Z})^n)$ faithfully follows Ellenberg and Gijswijt's proof. To verify the asymptotics, we work out a new elementary argument (inspired by Zeilberger's approach and a suggestion by Gijswijt). Ellenberg and Gijswijt use a technique known as the *polynomial method* to translate the problem to one about vector spaces of polynomials. We expect that our library contributions will be useful for proving other results that follow this approach.

A recent project begun at the Vrije Universiteit Amsterdam aims to bring together traditional mathematicians, formalizers, and tool developers to incorporate modern number theory into proof assistants.[1] The current paper shows that the goals of this project are within reach: we have formalized a paper published in the *Annals* less than two years ago.

The more general components of our formalization have been incorporated into the Lean mathematics library `mathlib`, which is available on GitHub.[2] The remainder of the formalization can be found with the supplementary material linked at the beginning of this paper. The code blocks presented below should be read as schematic, not literal. We sometimes change names, remove namespaces, omit universe levels, and swap implicit and explicit arguments for the sake of formatting and presentation.

2 Mathematical Background

Ellenberg and Gijswijt study a generalization of the cap set problem that holds for arbitrary finite fields (including $\mathbb{Z}/p\mathbb{Z}$ for any prime p). For the rest of this discussion, we fix a positive integer n and prime power q, and let $\mathbb{F}_q$ denote a finite field with cardinality q.

For $d \in \mathbb{R}$ with $0 \le d \le (q-1)n$, consider all n-variable monomials whose degree in each variable is at most $q-1$ and whose total degree is at most d, i.e.

$$M_n^d := \left\{ \prod_{i=1}^{n} x_i^{a_i} \in \mathbb{F}_q[x_1, \ldots, x_n] \;\middle|\; 0 \le a_i \le q-1 \text{ and } \sum_{i=1}^{n} a_i \le d \right\}.$$

Let $m_d := |M_n^d|$. Ellenberg and Gijswijt [18, Theorem 4] establish an upper bound for the size of generalized cap sets in terms of $m_{(q-1)n/3}$.

▶ **Theorem 1** (Ellenberg–Gijswijt). *Let $\alpha, \beta, \gamma \in \mathbb{F}_q$ such that $\alpha + \beta + \gamma = 0$ and $\gamma \neq 0$. Let A be a subset of $\mathbb{F}_q^n$ such that the equation $\alpha a_1 + \beta a_2 + \gamma a_3 = 0$ has no solutions with $a_1, a_2, a_3 \in A$ apart from those with $a_1 = a_2 = a_3$. Then $|A| \le 3m_{(q-1)n/3}$.*

If $(\alpha, \beta, \gamma) = (1, -2, 1)$, then the equation $\alpha a_1 + \beta a_2 + \gamma a_3 = 0$ is equivalent to $a_2 - a_1 = a_3 - a_2$; any solution to this, other than $a_1 = a_2 = a_3$, corresponds to a three term arithmetic progression.

To answer the cap set problem, it remains to determine good asymptotics for $m_{(q-1)n/3}$ as n tends to ∞.

[1] `https://lean-forward.github.io/`
[2] `https://github.com/leanprover-community/mathlib/`

▶ **Theorem 2.** *For every q there exists $c \in \mathbb{R}$ with $0 < c < q$ such that $m_{(q-1)n/3} = \mathcal{O}(c^n)$ as $n \to \infty$.*

Thus, with notation from Theorem 1, $|A| = \mathcal{O}(c^n)$ for some $0 < c < q$. For particular values of q we can write down explicit values of c. In the case of the original cap set problem, where $q = 3$ (and $\alpha = \beta = \gamma = 1$, also noting that $-2 = 1$ in $\mathbb{Z}/3\mathbb{Z}$), the proof method yields the following theorem; the exact value c already appears in Zeilberger [33].

▶ **Theorem 3.** *Let $c := \frac{3}{8}\sqrt[3]{207 + 33\sqrt{33}} < 2.755105$. Then $r_3\left((\mathbb{Z}/3\mathbb{Z})^n\right) \leq 3c^n$, and thus $r_3\left((\mathbb{Z}/3\mathbb{Z})^n\right) = o(2.755105^n)$ (as $n \to \infty$).*

The proof of Theorem 1 follows the *polynomial method*. (For a general introduction to the polynomial method, see e.g. Guth [22] or Tao [29].) Broadly speaking, this approach aims to analyze finite combinatorial objects by describing them through a system or space of polynomials. Techniques from algebraic geometry, or sometimes algebraic topology or simply linear algebra, can then be employed to study these polynomials; the results should translate back to properties of the original combinatorial objects of interest.

The polynomial method has been employed over the last decade to solve a large variety of open problems in arithmetic combinatorics and number theory. However, the scope and limitations of the method are still not well understood. In particular, its applicability to the cap set problem was unexpected, at least until the breakthrough of Croot, Lev, and Pach [9]. The main approach to the cap set problem for the previous half century was through Fourier theory methods.

We sketch here an overview of the proof of Theorem 1; more details can be found in Section 4. Let α, β, γ, and A be as stated in the theorem. We introduce the $\mathbb{F}_q$-vector space spanned by M_n^d, i.e.

$$S_n^d := \left\{ \sum_{m \in M_n^d} c_m m \;\middle|\; c_m \in \mathbb{F}_q \right\}.$$

Consider the $\mathbb{F}_q$-vector subspace V of S_n^d consisting of all polynomials $p \in S_n^d$ that vanish on the complement of $-\gamma A = \{-\gamma a \mid a \in A\}$ inside $\mathbb{F}_q^n$, i.e.

$$V := \{p \in S_n^d \mid \forall a \in \mathbb{F}_q^n \setminus (-\gamma A),\ p(a) = 0\}.$$

This is the setup of the polynomial method, the idea being that this space of polynomials V contains valuable information on $|-\gamma A| = |A|$ via $\dim(V)$. The strategy is to get good lower and upper bounds on $\dim(V)$. Namely, it holds that

$$\dim(V) \geq m_d - q^n + |A| \qquad \text{and} \qquad \dim(V) \leq 2m_{d/2}. \tag{1}$$

The lower bound is reasonably straightforward: it follows from rank-nullity and the remark that $|\mathbb{F}_q^n \setminus (-\gamma A)| = q^n - |A|$. The upper bound is more involved; the key to it is the following.

▶ **Proposition 4** (Proposition 2 from [18])**.** *Let $A \subseteq \mathbb{F}_q^n$ and $\alpha, \beta, \gamma \in \mathbb{F}_q$ with $\alpha + \beta + \gamma = 0$. Let $P \in S_n^d$ such that for all $a, b \in A$ with $a \neq b$ we have $P(\alpha a + \beta b) = 0$. Then*

$$|\{a \in A \mid P(-\gamma a) \neq 0\}| \leq 2m_{d/2}.$$

In addition, an elementary combinatorial argument gives us

$$q^n - m_d \leq m_{(q-1)n-d}. \tag{2}$$

Combining (1) and (2) and taking $d = 2(q-1)n/3$ gives us Theorem 1, i.e.

$$|A| \leq 3m_{(q-1)n/3}.$$

To establish the asymptotic behavior of this bound, Ellenberg and Gijswijt apply Cramér's theorem on large deviations. Tao [30] describes a more elementary approach via Stirling's approximation for the factorial function. Zeilberger [33] gives another even more elementary approach using recurrence sequences. Inspired by Zeilberger's paper, we worked out yet another approach, which lends itself very well to formalization in Lean. This was the initial approach we followed through; it is briefly described in Appendix A. Finally, thanks to a remark from Dion Gijswijt on our preprint, we arrive at a further significant simplification of the asymptotics proof, which we present below.

Our starting point is the combinatorial observation

$$m_d = \sum_{j=0}^{\lfloor d \rfloor} c_j^{(n)} \tag{3}$$

where $c_j^{(n)}$ is the coefficient of x^j in the polynomial $\left(1 + x + \ldots x^{q-1}\right)^n$. Let $r \in \mathbb{R}$ with $0 < r < 1$ and write $e := \lfloor (q-1)n/3 \rfloor$. Note that the $c_j^{(n)}$ are nonnegative and that $r^e \leq r^j$ for integers $0 \leq j \leq e$. Now

$$m_{(q-1)n/3} \cdot r^e = \sum_{j=0}^{e} c_j^{(n)} r^e \leq \sum_{j=0}^{e} c_j^{(n)} r^j \leq \sum_{j=0}^{(q-1)n} c_j^{(n)} r^j = \left(1 + r + \ldots + r^{q-1}\right)^n.$$

Dividing by $r^e \geq (r^{(q-1)/3})^n$ and defining

$$C_{r,q} := \frac{1 + r + \ldots + r^{q-1}}{r^{(q-1)/3}} = \frac{1 - r^q}{(1-r)r^{(q-1)/3}} \tag{4}$$

we arrive at our main asymptotics estimate

$$m_{(q-1)n/3} \leq C_{r,q}^n.$$

Elementary analysis gives us that for every $q > 1$ there exists some $0 < r < 1$ such that $C_{r,q} < q$, yielding Theorem 2. Specializing at $q = 3$ and $r = (\sqrt{33} - 1)/8$ gives the precise version of the cap set problem in Theorem 3. Similarly, minimizing $C_{r,q}$ for other values of q immediately leads to other growth rates, including those given by Zeilberger [33].

3 Lean and its Mathematics Library

The Lean proof assistant, developed principally by Leonardo de Moura, was first released in 2014 [11]. Lean implements a version of the calculus of inductive constructions (CIC) [8] with support for quotient types and classical reasoning. Since the release of Lean 3 in 2017 [17], there has been a concerted effort to develop `mathlib`, a comprehensive library for use in mathematics and computer science [4]. This library is built on the latest release of Lean, version 3.4.2. Some of the text in this section is adapted from a paper by the third author [26], which describes another formalization based on `mathlib`.

The datatypes available in `mathlib` include the concrete types commonly found in mathematics, among them $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$; finite sets and multisets over a base type; univariate and multivariate polynomials; and embeddings and isomorphisms between types.

```
class semigroup (α : Type) extends has_mul α :=
(mul_assoc : ∀ a b c : α, a * b * c = a * (b * c))

class monoid (α : Type) extends semigroup α, has_one α :=
(one_mul : ∀ a : α, 1 * a = a) (mul_one : ∀ a : α, a * 1 = a)

class group (α : Type) extends monoid α, has_inv α :=
(mul_left_inv : ∀ a : α, a⁻¹ * a = 1)

lemma one_inv (α : Type) [group α] : 1⁻¹ = (1 : α) :=
inv_eq_of_mul_eq_one (one_mul 1)
```

Figure 2 A sample of the bottom of the algebraic hierarchy. The lemma `one_inv` can be applied to any α for which Lean can infer an instance of `group` α.

The algebraic hierarchy of mathlib is designed using *type classes*, which endow a base type with extra structure in the forms of operations, properties, and notation [28, 32]. Lean's type class resolution mechanism automatically manages inheritance between type classes (Figure 2). If a type class `T'` extends (directly or by transitivity) a type class `T`, any theorem proved over `T` will apply to any type that instantiates `T'`. The algebraic hierarchy begins with semigroups and monoids and extends to rich structures including fields, Noetherian rings, and principal ideal domains. Van Doorn, von Raumer, and Buchholz [31] also explain how type classes are used to define an algebraic hierarchy in Lean.

The project described in this paper makes heavy use of the linear algebra and multivariate polynomial developments in `mathlib`. As with the algebraic hierarchy, these developments are built around type classes. The linear algebra theory in particular is modeled after the one found in Isabelle/HOL, reworked to use bundled submodules and bundled linear functions.

The fundamental type class in linear algebra is `module` α β, which assumes a ring structure on α and an abelian group structure on β, and endows β with a well-behaved scalar multiplication operation from α. When α is a field, this extends to the type class `vector_space` α β. Many of the typical theorems and constructions from linear algebra are defined over this type class, including the existence of bases, the rank-nullity theorem for linear maps, and the matrix representation of maps between finite-dimensional spaces. General instances establish that a family of vector spaces over an index type forms a vector space itself, and that a field α instantiates `vector_space` α α; combined, these allow us to consider the type of n-tuples of field elements, `fin n` $\to$ α, as a vector space over α.

Polynomials are another important instance of a vector space. Given a type σ used to index variables, we identify a monomial with a finitely supported function from σ to $\mathbb{N}$. A multivariate polynomial is a finitely supported function mapping monomials into a coefficient ring α. We use the infix notation $\to_0$ for functions of finite support.

```
def mv_polynomial (σ α : Type) [comm_semiring α] := (σ →₀ ℕ) →₀ α
```

When α is a field, this type forms a vector space over α. Important operations on polynomials include `eval`, which evaluates the polynomial in α given an assignment $\sigma \to \alpha$, and `total_degree`, which computes the maximum degree over all monomials in a polynomial.

Many contributions were made to `mathlib` in the course of this project. In addition to extending the linear algebra, polynomial, and finitely supported function theories, we added various results about big operators and series, finite sets and multisets, and orders of elements in finite groups (to show, for example, that $a^q = a$ for $a \in \mathbb{F}_q$).

Another type class that plays an important role in our formalization is `fintype α`, which provides functions for listing and counting the elements of α. The standard finite types instantiate this class, including the type `fin n` of natural numbers less than `n`. When α and β instantiate `fintype`, so does the function type `α → β`.

The `mathlib` library is designed with a focus on classical logic. Type-valued declarations are defined computably when possible, but classical logic is used freely in propositions. Our formalization is similarly classical.

Readers unused to Lean syntax should note that explicit arguments to declarations are enclosed in parentheses `()`, implicit arguments are enclosed in curly brackets `{}`, and type class arguments are enclosed in square brackets `[]`. Only explicit arguments are given by the user when applying a declaration. Implicit arguments are inferred from later arguments and the expected type, and type class arguments are inferred by type class resolution.

Another important feature of Lean syntax is its projection notation. As an example, let terms `F : polynomial α` and `a : α` be given. The operator

```
polynomial.eval : α → polynomial α → α
```

evaluates a polynomial at an argument. Because the head symbol of the type of `F` is `polynomial`, matching the namespace of `eval`, we can abbreviate `polynomial.eval a F` with the more concise `F.eval a`. This notation can be nested:

```
polynomial.eval a (polynomial.derivative F)
```

shortens to `F.derivative.eval a`.

4 The Cap Set Bound

As described in Section 2, Ellenberg and Gijswijt's solution to the cap set problem [18] proceeds in two parts. The first part establishes an upper bound on the size of a cap set in terms of the dimension of a vector space of polynomials; the second part shows the asymptotic behavior of this bound. Our formalization is similarly divided. This section describes the formal construction of the bound, and Section 5 explains the verification of the asymptotics. Our construction of the bound closely follows Ellenberg and Gijswijt's paper.

At the outset of our efforts, the first author produced a detailed paper proof[3] of the result, drawing from Ellenberg and Gijswijt and from Zeilberger [33] and adapting the asymptotics part significantly. The most recent approach to this part was added after initially submitting this paper, and was subsequently also formalized. The theorem names in the following sections match the corresponding statements in the paper proof.

The theorems here hold over an arbitrary finite field. We will take a fixed parameter `α : Type` instantiating the type classes `[fintype α]` and `[discrete_field α]`, and use `q` to abbreviate the cardinality `fintype.card α`. In this section, we also fix a parameter `n : ℕ`, representing the length of the tuples in the set whose cardinality we will bound.

The goal of this section, then, is to define a function `m` and prove the following theorem, which corresponds to the informal statement of Theorem 1 above:

[3] This writeup is available at `https://lean-forward.github.io/e-g/`.

```
theorem theorem_12_1 {α : Type} [discrete_field α] [fintype α]
  (n : ℕ) {a b c : α} (hc : c ≠ 0) (habc : a + b + c = 0)
  (hn : n > 0) {A : finset (fin n → α)}
  (ha : ∀ x y z ∈ A, a · x + b · y + c · z = 0 → x = y ∧ x = z) :
  A.card ≤ 3 * m α n (1 / 3 * ((card α − 1) * n))
```

Ellenberg and Gijswijt's key insight is to translate the question to one concerning vector spaces of multivariate polynomials. After setting up this translation, this bound will follow from a sequence of intermediate lemmas.

4.1 Setting Up the Polynomial Method

The type `mv_polynomial (fin n) α` forms a vector space, by results established in `mathlib` (Section 3). We will focus our attention on a particular subspace. We define `M` to be the set of monomials in `n` variables where the exponent of each variable is strictly less than `q`. This set is linearly independent with respect to `α`.

```
def M : finset (mv_polynomial (fin n) α) :=
(finset.univ.image
  (λ f : fin n →₀ fin q, f.map_range fin.val rfl)).image
    (λ d : fin n →₀ ℕ, monomial d (1:α))
```

For `d : ℚ`, we make the following definitions:

- `M'` is the subset of `M` whose elements have total degree at most `d`.
- `S'` is the span of `M'`; this is a subspace of `mv_polynomial (fin n) α`.
- `m` is the dimension of `S'`.

Since `M'` is linearly independent, it follows that the cardinality of `M'` is equal to `m`.

```
def M' (d : ℚ) : finset (mv_polynomial (fin n) α) :=
M.filter (λ m, d ≥ mv_polynomial.total_degree m)

def S' (d : ℚ) : submodule α (mv_polynomial (fin n) α) :=
submodule.span α ((M' d) : set (mv_polynomial (fin n) α))

def m (d : ℚ) : ℕ := (vector_space.dim α (S' d)).to_nat

lemma M'_card (d : ℚ) : (M' d).card = m d
```

Much of the following argument will be carried out in a subspace of `S'`. We first describe this subspace generically. Given a subspace of polynomials `T` and a set of vectors `A`, we define `zero_set T A` to be the set of polynomials in `T` that evaluate to 0 at all elements of `A`. By basic properties of polynomial evaluation, this set is a subspace of `T`.

```
parameters (T : subspace α (mv_polynomial (fin n) α))
           (A : finset (fin n → α))

def zero_set : set (mv_polynomial (fin n) α) :=
{p ∈ T.carrier | ∀ a ∈ A, mv_polynomial.eval a p = 0}

def zero_set_subspace : subspace α (mv_polynomial (fin n) α) :=
{ carrier := zero_set,
  zero := ⟨submodule.zero, by simp⟩,
  add := λ _ _ hx hy,
   ⟨submodule.add hx.1 hy.1, λ _ hp, by simp [hx.2 hp, hy.2 hp]⟩,
  smul := λ _ _ hp,
   ⟨submodule.smul hp.1, λ _ hx, by simp [hp.2 hx]⟩ }
```

Our target theorem takes as parameters `a b c : α` and `A : finset (fin n → α)` satisfying certain properties, in particular that `c ≠ 0`. Let these terms be given. We define `neg_cA` to be the image of `A` under multiplication by $-c$, and `V` to be the zero set of `S'` with respect to the complement of `neg_cA`.

```
def neg_cA : finset (fin n → α) := A.image (λ z, (-c) · z)

def V : subspace α (S' d) :=
zero_set_subspace (S' d) (finset.univ \ neg_cA)

def V_dim : ℕ := (vector_space.dim α V).to_nat
```

Our goal – an upper bound on the cardinality of `A`, in terms of `m` – will follow from a number of lemmas controlling the dimension of `V`.

4.2 Lemma 1: Bounding the Dimension from Below

The first lemma establishes a lower bound for the dimension of `V` in terms of `m`, `q`, and `A.card`. We prove this via a generic result that holds for every `zero_set_subspace` of a finite-dimensional space.

```
theorem lemma_9_2 (T : subspace α (mv_polynomial (fin n) α))
  (A : finset (fin n → α)) :
  (vector_space.dim α zero_set_subspace).to_nat + A.card ≥
    (vector_space.dim α T).to_nat
```

This lemma is an exercise in linear algebra. It follows quickly from the rank-nullity theorem. The formal proof takes little work with our additions to the linear algebra theory in `mathlib`.

We now set a parameter `d : ℚ` which will remain fixed until the end of this section. After specializing `lemma_9_2` and performing a cardinality computation, we obtain the following:

```
theorem lemma_12_2 : q^n + V_dim ≥ m d + A.card
```

The `mathlib` definition of `vector_space.dim` takes values in the type `cardinal`, since vector spaces are not restricted to finite dimensions. (Perhaps confusingly, `finset.card` and `fintype.card` take values in $\mathbb{N}$.) In our setting, the vector space `S'`, and hence its subspace `V`, is finite dimensional. The cast `cardinal.to_nat` is thus well behaved.

4.3 Lemmas 2 and 3: Bounding the Dimension from Above

Next we establish an upper bound for the dimension of `V`. It is conceptually clearest to achieve this via two lemmas, one which bounds the dimension above by an intermediate value, and one which bounds this value above by `m`.

To prove the first lemma, we define the support set of a polynomial to be the set of points on which it does not evaluate to 0:

```
def sup (p : mv_polynomial (fin n) α) : finset (fin n → α) :=
finset.univ.filter (λ x, p.eval x ≠ 0)
```

A general argument about finite sets shows that there is some polynomial in `V` with maximal support.

```
lemma exi_max_sup :
  ∃ P ∈ V, ∀ P' ∈ V, sup P ⊆ sup P' → sup P = sup P'
```

We define `P` to be this polynomial and `P_sup` to be `sup P`, allowing us to state the following:

```
theorem lemma_12_3 : P_sup.card ≥ V_dim
```

The proof of this lemma involves some algebraic manipulation of the evaluation function `mv_polynomial.eval`. It invokes yet another polynomial subspace, the zero set of `V` with respect to `P_sup`.

In order to relate `P_sup` to other more interesting constants, we must prove a second lemma:

```
theorem lemma_12_4 : P_sup.card ≤ 2 * m (d/2)
```

This lemma is a special case of Proposition 4 (Section 2), stated here in Lean:

```
theorem proposition_11_1 {p :  mv_polynomial (fin n) α}
  (A : finset (fin n → α)) : p ∈ S' n d →
  (∀ (x : fin n → α), x ∈ A → ∀ (y : fin n → α), y ∈ A →
    x ≠ y → p.eval (a · x + b · y) = 0) →
  (A.filter (λ x, p.eval (-c · x) ≠ 0)).card ≤ 2 * m (d / 2)
```

Proving this proposition requires the most intricate argument of our formalization. We note that this is in line with Ellenberg and Gijswijt's paper; their corresponding Proposition 2 makes up nearly a third of the non-expository content. Some of the intricacy comes from another shift of representation. Every student of linear algebra learns that linear transformations between finite-dimensional vector spaces can be represented by matrices, and it is standard in mathematics to conflate the two concepts. While our lemma (after unfolding the definition of `P_sup`) is stated in terms of the linear transformation `p.eval`, Ellenberg and Gijswijt's argument proceeds more naturally in the matrix setting. Formalizing their argument required significant library development to unify the treatment of linear transformations and matrices in Lean. We expect that this development will be reusable in future results that depend on linear algebra.

Briefly, the proof of `proposition_11_1` proceeds as follows. Given terms `a b : α`, `x y : fin n → α`, and `p : mv_polynomial (fin n) α` with `p ∈ S' d`, the term `p.eval (a · x + b · y)` can be written as a linear combination of evaluated monomials in `M' d`. We define an `A × A` matrix `B` such that `B x y = p.eval (a · x + b · y)`. In fact, we can factor the matrix `B` and express it in the following form:

```
lemma B_eq_sum_matrix : B =
  split_left.sum (λ _ _, matrix.vec_mul_vec _ _) +
  split_right.sum (λ  _ _, matrix.vec_mul_vec _ _)
```

(We direct interested readers to our formalization for the details of this computation.) Here, the cardinalities of the finite sets `split_left` and `split_right` are at most `m (d/2)`. Since the product of two vectors `matrix.vec_mul_vec` has rank 1, this implies that `B` has rank at most `2 * m (d / 2)`. But in fact, `B` is a diagonal matrix, from which we can infer that its rank is equal to the cardinality we wish to bound.

4.4 Lemma 4: A Combinatorial Calculation

Our next lemma, largely independent of the previous ones, relates different values of `m`.

```
theorem lemma_12_5 : q^n ≤ m ((q-1)*n - d) + m d
```

This lemma follows from a combinatorial argument on `fin n → fin q`, the type of `n`-tuples of natural numbers less than `q`. First, we define functions to map such a tuple to the monomial with corresponding exponents, and in reverse:

```
def monom : (fin n → fin q) → mv_polynomial (fin n) α
def monom_exps : mv_polynomial (fin n) α → (fin n → fin q)
```

Note that these functions are inverses when we restrict `fin n → fin q` to the subset `M`.

We define five terms of type `finset (fin n → fin q)`, including the universal set:

- `I := finset.univ`
- `B := {v ∈ I // (total_degree (monom v)) ≤ d}`
- `C := {v ∈ I // (total_degree (monom v)) > d}`
- `D := {v ∈ I // (total_degree (monom v)) < (q-1)*n - d}`
- `E := {v ∈ I // (total_degree (monom v)) ≤ (q-1)*n - d}`

There are a number of straightforward cardinality calculations that follow. Among them, we show that `B.card = m d`, since `B` is the image of `M' d` under `monom_exps`. It similarly holds that `E.card = m ((q-1)*n - d)`. The function sending the tuple $(a_1, \ldots, a_n)$ to $(q-1-a_1, \ldots, q-1-a_n)$ is a bijection and maps `C` to `D`; thus these sets have the same cardinality. Combining these calculations leads us to our goal.

Thanks to the large library of `finset` operations in `mathlib`, the proof of this lemma is basically frictionless. Indeed, the least pleasant part is checking that the bijection used is in fact a bijection, an argument that involves some trivial natural number arithmetic.

4.5 Lemma 5: Connecting These Lemmas

We have nearly achieved our goal for this section. Combining the previous four lemmas via linear arithmetic, we obtain the following:

```
theorem lemma_12_6 : A.card ≤ 2 * m (d/2) + m ((q-1)*n - d) :=
by linarith using [lemma_12_2, lemma_12_3, lemma_12_4, lemma_12_5]
```

Finally, abstracting the parameter `d` and instantiating it with `2/3*(q-1)*n` delivers our desired bound.

```
theorem theorem_12_1 : A.card ≤ 3*(m (1/3*((q-1)*n)))
```

5 Asymptotics

We have shown an upper bound for the cardinality of a cap set `A` in terms of `n`. To be precise, this bound is proportional to the number of monomials in `n` variables with total degree at most `(q-1)*n/3`, where `q` is the cardinality of the underlying finite field.

Our goal was to investigate the growth rate of this bound, in terms of `n`. In particular, we would like to show that it grows at a rate bounded above by `c^n`, for some `c < q`. Ellenberg and Gijswijt apply Cramér's theorem, a fairly deep result in probability theory (not to be confused with Cramer's rule), to derive this fact. But this detour is not necessary, and formalizing Cramér's theorem would be a significant undertaking on its own. We verify the growth rate of the size of `A` using more elementary methods. While the results of this section could be stated in terms of $\mathcal{O}$-notation [1], we favor a more explicit style, which allows us to state the $q = 3$ result in very concrete terms.

Our goal is the following general statement:

```
theorem general_cap_set {α : Type} [discrete_field α] [fintype α] :
  ∃ B C : ℝ, B > 0 ∧ C > 0 ∧ C < card α ∧
    ∀ {a b c : α} {n : ℕ} {A : finset (fin n → α)},
      c ≠ 0 → a + b + c = 0 →
      (∀ x y z ∈ A, a · x + b · y + c · z = 0 → x = y ∧ x = z) →
      A.card ≤ B * C ^ n
```

Our motivating example is concerned with the case where the underlying field is $\mathbb{Z}/3\mathbb{Z}$. In this case, we can be more explicit about the growth rate:

```
theorem cap_set {n : ℕ} {A : finset (fin n → ℤ/3ℤ)} :
  (∀ x y z ∈ A, x + y + z = 0 → x = y ∧ x = z) →
    A.card ≤ 3 * (((3/8) ^ 3 * (207 + 33 * sqrt 33)) ^ (1/3)) ^ n
```

Since we have that

$$\sqrt[3]{\left(\frac{3}{8}\right)^3 \left(207 + 33\sqrt{33}\right)} \approx 2.755,$$

this result answers the cap set problem in the affirmative.

To prove `general_cap_set`, we will show an alternate representation for `m` and develop an argument that bounds this value from above in terms of `n` and `d`. This argument involves some combinatorial calculations similar to those presented in Section 4.4.

In the previous section we worked with a fixed parameter `n`, the length of the vectors. It is now necessary to abstract over this parameter. (We will keep the base field α and its cardinality `q` fixed.) Note that `m` depends on both `n` and a rational input `d`.

5.1 Expressing `m` as a Sum of Coefficients

Our first lemma will show that we can write `m` as a sum of coefficients depending on n and d. On paper, we define

$$c_j^{(n)} := \left| \left\{ (a_1, \ldots, a_n) \;\middle|\; a_i \in \{0, 1, \ldots, q-1\} \text{ and } \sum_{i=1}^{n} a_i = j \right\} \right|.$$

We again face a choice of how to represent these values in Lean. In Section 4.4, we represented such tuples $(a_1, \ldots, a_n)$ with the type `fin n → fin q`. This type is very convenient when `n` is fixed, but a following lemma will proceed by induction on `n`, and the function representation is cumbersome in this kind of argument. We choose instead to represent these tuples with the type `vector (fin q) n`, defined to be the subtype of `list (fin q)` whose elements have fixed length `n`. To connect with earlier results stated using the function representation, we will show a bijection between the two types. Moving between representations like this is aided by library support for establishing bijections and showing that relevant properties are preserved, and with the right support, it is far easier to carry out arguments in the "natural" setting.

With this in mind, we define:

```
def sf (n j : ℕ) : finset (vector (fin q) n) :=
finset.univ.filter (λ f, (f.nat_sum = j))

def cf (n j : ℕ) : ℕ := (sf n j).card
```

Following the bijection between representations of tuples, and reusing some of the cardinality computations from Section 4.4, we show that `m n d` is equal to the sum of `cf q n j` for `0 ≤ j ≤ ⌊d⌋`:

```
theorem lemma_13_8 (n : ℕ) {d : ℚ} (hd : d ≥ 0) :
  m n d = (finset.range (⌊d⌋.nat_abs + 1)).sum (cf n)
```

To get a better handle on `m`, we would like a more algebraic representation of `cf`. As an intermediate step, we turn again to the setting of polynomials, this time univariate: we will show that for each j and n, $c_j^{(n)}$ is equal to the jth coefficient of the polynomial $(1 + x + \ldots + x^{q-1})^n$.

It is in this argument that we benefit from using the list representation for tuples, as we need to prove:

```
lemma cf_mul (n j : ℕ) : cf (n+2) j =
  (finset.range (j + 1)).sum (λ i, (cf 1 (j − i)) * cf (n + 1) i)
```

This combinatorial puzzle requires lifting $(n+1)$-tuples to $(n+2)$-tuples. Any $(n+2)$-tuple of natural numbers less than q whose values sum to j can be constructed by appending its last value k to an $(n+1)$-tuple whose values sum to $i = j - k$. The number of such $(n+2)$-tuples, then, is the sum of the number of such $(n+1)$-tuples where i ranges from 0 to $\max(q-1, j)$. Since `cf 1 k` is 0 when `k > q` and 1 otherwise, this sum is equal to the expression in `cf_mul`.

Counting arguments like this can make for entertaining puzzles on paper, but the pain of formalizing them can be compounded by using the wrong representation. We found that the lifting of tuples required for this argument was much more natural under the list representation for tuples; casts in the function representation became unwieldy.

With this identity, and proceeding by induction on `n`, we can define the polynomial $1 + x + \ldots + x^{q-1}$ and show our desired result:

```
def one_coeff_poly (m : ℕ) : polynomial ℕ :=
(finset.range m).sum (λ k, (polynomial.X : polynomial ℕ) ^ k)

theorem lemma_13_9 (hq : q > 0) :
  ∀ n j : ℕ, ((one_coeff_poly q) ^ n).coeff j = cf n j
```

5.2 Concrete Bounds on `m`

We can now write `m` in terms of the coefficients `cf`. We will use this representation to establish a concrete upper bound on the values of `m`. This upper bound will be in terms of another auxiliary value:

```
def crq (r : ℝ) (q : ℕ) :=
((one_coeff_poly q).eval₂ coe r) / r ^ ((q−1)/3)
```

Note that for `p : polynomial ℕ` and `r : ℝ`, `p.eval₂ coe r` embeds the coefficients of `p` into the real numbers and evaluates the resulting polynomial at `r`.

For every `r` between 0 and 1, `crq` bounds `m`:

```
theorem theorem_14_1 {r : ℝ} (hr : 0 < r) (hr2 : r < 1) :
  m ((q − 1)*n / 3) ≤ (crq r q) ^ n
```

This result is derived from `theorem_13_8` and `theorem_13_9`, with the additional fact that summing the monomials of a polynomial over its support is the same as evaluating the polynomial.

```
lemma finset_sum_range {r : ℝ} (hr : 0 < r) (hr2 : r < 1) :
  (finset.range ((q − 1) * n + 1)).sum (λ j, r ^ j * (cf q n j)) =
    ((one_coeff_poly q) ^ n).eval₂ coe r
```

Since `crq 1 q = q` and the derivative of `crq` with respect to `r` is positive at `r = 1`, we have from elementary calculus:

```
theorem lemma_13_15 : ∃ r : ℝ, 0 < r ∧ r < 1 ∧ crq r q < q
```

Instantiating `theorem_14_1` with this `r`, invoking `theorem_12_1`, and abstracting the type parameter α leads us to the theorem `general_cap_set` stated at the beginning of this section.

We finally return to the original cap set problem with `q = 3`. Pen and paper calculations show that `crq r 1` is minimized in `r` at `r := (real.sqrt 33 − 1) / 8`. Aided by the numeral and ring normalization tactics in `mathlib`, we establish that `0 < r < 1` and that `crq r 3 = ((3 / 8)^3 * (207 + 33*real.sqrt 33))^(1/3)`. We apply `theorem_14_1` to this `r` to conclude:

```
theorem cap_set {n : ℕ} {A : finset (fin n → ℤ/3ℤ)} :
  (∀ x y z ∈ A, x + y + z = 0 → x = y ∧ x = z) →
    A.card ≤ 3 * (((3/8) ^ 3 * (207 + 33 * sqrt 33)) ^ (1/3)) ^ n
```

6 Related Work

We are not aware of any existing formal developments that relate directly to the cap set problem or the polynomial method. Since the core library components of our proof are in combinatorics and number theory, linear algebra, and the theory of polynomials, we provide here a survey of formalizations in these areas. This incomplete list is meant to indicate the depth and flavor of such projects.

The combinatorial arguments we employ are fairly simple results about involutions and the cardinalities of finite sets; similar developments exist in the libraries of most modern proof assistants. Gonthier's proof of the four color theorem in Coq [19] includes some more sophisticated proofs. Dubois, Giorgetti, and Genestier [14] also provide a Coq library for enumerative combinatorics, again more sophisticated than what is needed in our proof.

While the result of Ellenberg and Gijswijt is most clearly characterized as combinatorics, it is also of interest in number theory. There has been recent attention toward formalizing results in this area, including Eberl's work on analytic number theory in Isabelle/HOL [16] and Lewis' work on the p-adic numbers in Lean [26]. Chyzak, Mahboubi, Sibut-Pinote, and Tassi's Coq proof that $\zeta(3)$ is irrational [7] is also relevant.

Finite fields play an important role in combinatorics and number theory and are needed to state our general result. Chan and Norrish's mechanization of the AKS algorithm [5] shows an approach to their study in HOL4, which makes for an interesting contrast with our approach in a dependently typed system. Their subsequent work [6] relates to ours in its study of polynomials over finite fields.

There are many formal proof developments of linear algebra. Our additions to `mathlib` were partially inspired by the impressive work of Gonthier in Coq [20], Lee [25] and Aransay and Divasón [2, 13] in Isabelle/HOL, and Harrison in HOL Light [24].

Our formalization focuses in particular on the vector space of polynomials, also seen in Divasón, Joosten, Thiemann, and Yamada [12]. As with linear algebra, polynomials are a fundamental object of study in mathematics, and they appear in most proof assistant libraries. Some recent results concerning polynomials include Bernard, Bertot, Rideau, and Strub [3] and Eberl [15].

7 Conclusion

We have formalized Ellenberg and Gijswijt's solution to the cap set problem, a recent and celebrated result in combinatorics. Our formalization is evidence that verifying certain cutting-edge mathematics is possible without enormous investments of time or resources. This effort was undertaken as part of the Lean Forward project, which aims to develop tools, tactics, and libraries to formalize modern results in number theory and related areas. Much of the background theory we have implemented will be of future use in this project.

At the outset of our efforts, the first author produced a detailed paper proof of the result, drawing from Ellenberg and Gijswijt and from Zeilberger [33] and adapting the asymptotics part significantly. We used this writeup as a blueprint for our formalization. It was heartening to see that the blueprint translated very directly to Lean. We were able to work at a similar level of abstraction as the original sources without any complications introduced by the proof assistant.

Our proof of the asymptotics is a significant simplification of the original arguments. While in principle this could have been found without any interactive theorem proving, it was ultimately due to the formalization process, including the necessity to explore alternative paths of this part of the proof and feedback from Gijswijt on an earlier version of this paper, that this simplification was established.

As usual, it is difficult to compare the length of formal proofs with their paper counterparts, since the background assumptions and level of detail differ significantly. Nevertheless, we can provide some approximate information. Ellenberg and Gijswijt's paper contains just over two pages of mathematical work. Our blueprint is seventeen pages long; the first six pages are preliminary material, and two pages correspond to an obsolete argument (Appendix A). The remaining nine pages correspond to around 2000 lines of our formalization. (This does not represent our entire effort: thousands more lines of general definitions and proofs were added to `mathlib` as part of this project.) The ratio of 2000 lines of formal proof to two pages of paper proof is perhaps misleading, since we take a more verbose approach to checking the asymptotic behavior of the upper bound. (Ellenberg and Gijswijt take only one paragraph to invoke Cramér's theorem.) A better comparison is the part of the proof described in Section 4: 900 formal lines subsume a page and a half of paper proof. The corresponding section of our detailed writeup is just under five pages.

This formalization, and `mathlib` more generally, rely heavily on hierarchies of type classes. In some sections of our proof – particularly those involving linear subspaces of the type of multivariate polynomials – we found that type class inference behaved erratically. The backtracking search performed by Lean's elaborator is sensitive to many features, and import order and additional instances can greatly affect the depth and speed of the search. We ended up revising the hierarchy in parts of `mathlib` to simplify this. A moral we have taken from this project is that "misleading" instances that lead the elaborator down a long and ultimately unsuccessful path can be nearly as dangerous as circular instances.

References

1 Reynald Affeldt, Cyril Cohen, and Damien Rouhling. Formalization Techniques for Asymptotic Reasoning in Classical Analysis. *Journal of Formalized Reasoning*, October 2018. URL: `https://hal.inria.fr/hal-01719918`.

2 Jesús Aransay and Jose Divasón. Formalization and Execution of Linear Algebra: From Theorems to Algorithms. In Gopal Gupta and Ricardo Peña, editors, *Logic-Based Program Synthesis and Transformation*, pages 1–18, Cham, 2014. Springer International Publishing.

3 Sophie Bernard, Yves Bertot, Laurence Rideau, and Pierre-Yves Strub. Formal Proofs of Transcendence for e and Pi As an Application of Multivariate and Symmetric Polynomials. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, pages 76–87, New York, NY, USA, 2016. ACM. doi:10.1145/2854065.2854072.

4 Mario Carneiro. The Lean 3 Mathematical Library (presentation), July 2018. URL: http://robertylewis.com/files/icms/Carneiro_mathlib.pdf.

5 Hing-Lun Chan and Michael Norrish. Mechanisation of AKS Algorithm: Part 1 – The Main Theorem. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving*, pages 117–136, Cham, 2015. Springer International Publishing.

6 Hing-Lun Chan and Michael Norrish. Proof Pearl: Bounding Least Common Multiples with Triangles. In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving*, pages 140–150, Cham, 2016. Springer International Publishing.

7 Frédéric Chyzak, Assia Mahboubi, Thomas Sibut-Pinote, and Enrico Tassi. A Computer-Algebra-Based Formal Proof of the Irrationality of $\zeta(3)$. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, pages 160–176, Cham, 2014. Springer International Publishing.

8 Thierry Coquand and Christine Paulin. Inductively defined types. In *COLOG-88 (Tallinn, 1988)*, volume 417 of *Lec. Notes in Comp. Sci.*, pages 50–66. Springer, Berlin, 1990. doi:10.1007/3-540-52335-9_47.

9 Ernie Croot, Vsevolod F. Lev, and Péter Pál Pach. Progression-free sets in $\mathbb{Z}_4^n$ are exponentially small. *Ann. of Math. (2)*, 185(1):331–337, 2017. doi:10.4007/annals.2017.185.1.7.

10 N. G. de Bruijn. AUTOMATH, a Language for Mathematics. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, pages 159–200. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983. doi:10.1007/978-3-642-81955-1_11.

11 Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover, 2014. URL: http://leanprover.github.io/files/system.pdf.

12 Jose Divasón, Sebastiaan Joosten, René Thiemann, and Akihisa Yamada. A Formalization of the Berlekamp-Zassenhaus Factorization Algorithm. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 17–29, New York, NY, USA, 2017. ACM. doi:10.1145/3018610.3018617.

13 Jose Divasón and Jesús Aransay. Rank-Nullity Theorem in Linear Algebra. *Archive of Formal Proofs*, January 2013. Formal proof development. URL: http://isa-afp.org/entries/Rank_Nullity_Theorem.html.

14 Catherine Dubois, Alain Giorgetti, and Richard Genestier. Tests and Proofs for Enumerative Combinatorics. In Bernhard K. Aichernig and Carlo A. Furia, editors, *Tests and Proofs*, pages 57–75, Cham, 2016. Springer International Publishing.

15 Manuel Eberl. Symmetric Polynomials. *Archive of Formal Proofs*, September 2018. Formal proof development. URL: http://isa-afp.org/entries/Symmetric_Polynomials.html.

16 Manuel Eberl. Nine Chapters of Analytic Number Theory in Isabelle/HOL. In *Interactive Theorem Proving*, 2019.

17 Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *Proceedings of the ACM on Programming Languages*, 1(ICFP):34, 2017.

18 Jordan S. Ellenberg and Dion Gijswijt. On large subsets of $\mathbb{F}_q^n$ with no three-term arithmetic progression. *Ann. of Math. (2)*, 185(1):339–343, 2017. doi:10.4007/annals.2017.185.1.8.

19 Georges Gonthier. The Four Colour Theorem: Engineering of a Formal Proof. In Deepak Kapur, editor, *Computer Mathematics*, pages 333–333, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

20 Georges Gonthier. Point-Free, Set-Free Concrete Linear Algebra. In Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving*, pages 103–118, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

21 Sébastien Gouëzel and Vladimir Shchur. A corrected quantitative version of the Morse lemma. *arXiv preprint*, 2018. arXiv:1810.04579.

22 Larry Guth. *Polynomial methods in combinatorics*, volume 64 of *University Lecture Series*. American Mathematical Society, Providence, RI, 2016.

23 Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Hoang Le Truong, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, et al. A formal proof of the Kepler conjecture. In *Forum of Mathematics, Pi*, volume 5. Cambridge University Press, 2017.

24 John Harrison. The HOL Light Theory of Euclidean Space. *J. Autom. Reason.*, 50(2):173–190, February 2013. doi:10.1007/s10817-012-9250-9.

25 Holden Lee. Vector Spaces. *Archive of Formal Proofs*, August 2014. Formal proof development. URL: http://isa-afp.org/entries/VectorSpace.html.

26 Robert Y. Lewis. A Formal Proof of Hensel's lemma over the p-adic Integers. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, pages 15–26, New York, NY, USA, 2019. ACM. doi:10.1145/3293880.3294089.

27 Roman Matuszewski and Piotr Rudnicki. Mizar: the first 30 years. *Mechanized Mathematics and Its Applications*, 4(1):3–24, March 2005.

28 Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21(4):795–825, 2011.

29 Terence Tao. Algebraic combinatorial geometry: the polynomial method in arithmetic combinatorics, incidence combinatorics, and number theory. *EMS Surv. Math. Sci.*, 1(1):1–46, 2014. doi:10.4171/EMSS/1.

30 Terence Tao. A symmetric formulation of the Croot-Lev-Pach-Ellenberg-Gijswijt capset bound, May 2016. URL: http://terrytao.wordpress.com/2016/05/18.

31 Floris van Doorn, Jakob von Raumer, and Ulrik Buchholtz. Homotopy Type Theory in Lean. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving*, pages 479–495. Springer International Publishing, 2017.

32 P. Wadler and S. Blott. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM. doi:10.1145/75277.75283.

33 Doron Zeilberger. A Motivated Rendition of the Ellenberg–Gijswijt Gorgeous proof that the Largest Subset of F_3^n with No Three-Term Arithmetic Progression is $O(c^n)$, with $c = \sqrt[3]{(5589+891\sqrt{33})/8} = 2.755104613023633000221 27\ldots$. *arXiv preprint*, 2016. arXiv:1607.01804.

A An Earlier Proof of Asymptotics

After submission of our paper, Dion Gijswijt suggested a further simplification to the approach we used for controlling the asymptotic behavior of the bound. The argument we present above in Sections 2 and 5 follows this suggestion. For the sake of completeness, we present here our original approach, which may be of interest in its own right.

A.1 Informal Description

We will bound the coefficients of the polynomials from (3):

$$m_d = \sum_{i=0}^{\lfloor d \rfloor} \left(\text{coefficient of } x^i \text{ in the polynomial } \left(1 + x + \ldots x^{q-1}\right)^n \right). \tag{5}$$

We can work in an algebraic manner as follows, thus avoiding Cauchy's residue theorem from complex analysis. Let k be any field, $f \in k[x]$, $i \in \mathbb{N}$, $\zeta \in k^*$ of finite order l, and $r \in k^*$. If $l > \max(\deg(f), i)$, then

$$l \cdot \big(\text{coefficient of } x^i \text{ in the polynomial } f\big) = \sum_{j=0}^{l-1} \frac{f(r\zeta^j)}{r^i \zeta^{ij}}. \tag{6}$$

The key ingredient for proving this statement is the following special case of the geometric sum, where ζ and l are as above and $h \in \mathbb{Z}$.

$$\sum_{j=0}^{l-1} \zeta^{hj} = \begin{cases} 0 & \text{if } l \nmid h \\ l & \text{if } l \mid h \end{cases}$$

Repeatedly applying (6) to (5) with $k = \mathbb{C}$, $\zeta = \exp(2\pi\sqrt{-1}/l)$ for any $l > n(q-1)$, and $r \in \mathbb{R}$ satisfying $0 < r < 1$, as well as calculating and estimating quite a bit, we obtain that

$$m_{(q-1)n/3} \leq B_{r,q} C_{r,q}^n$$

for some constants $B_{r,q}, C_{r,q} \in \mathbb{R}_{>0}$ depending only on r and q. Specifically, we can take $C_{r,q}$ as in (4).

A.2 Formalization

We pick up at the beginning of Section 5.2, where we have not yet established an algebraic representation for `cf`. It is necessary to get a better handle on the coefficients of `one_coeff_poly ^ n`. A brief detour into estimates with complex numbers will result in the following bound:

```
theorem lemma_13_10 (n : ℕ) {r : ℝ} (hr : r > 0) :
  cf n j ≤ (((one_coeff_poly q)^n).eval₂ coe r) / r^j
```

Note that for `p : polynomial ℕ` and `r : ℝ`, `p.eval₂ coe r` embeds the coefficients of `p` into the real numbers and evaluates the resulting polynomial at `r`. This operation is generic, and we will soon embed this same polynomial into $\mathbb{C}$.

To obtain the bound in `lemma_13_10`, we will use a general result about complex polynomials. We derive this directly, but we note that it also follows from general considerations about Laurent polynomials:

```
def ζk (k : ℤ) : ℂ := exp (2*π*I/k)

lemma pick_out_coef {f : polynomial ℂ} {i k : ℕ} (h1 : k > i)
  (h2 : k > nat_degree f) {r : ℝ} (h3 : r > 0) :
  (coeff f i) * k =
    (range k).sum (λ j, (eval (r*(ζk k)^j) f)/(r^i * (ζk k)^(i*j)))
```

When we instantiate `f` with the embedding of `one_coeff_poly ^ n` into $\mathbb{C}$, we see that this complex sum is in fact a nonnegative real number for each `i`, since it is equal to `cf i n`. We can thus approximate its absolute value using the triangle inequality to derive `lemma_13_10` above.

We can now write `m` in terms of the coefficients `cf`, and for each positive real `r`, we can bound `cf` from above in terms of `r`. It remains to establish a concrete upper bound on `m`.

We will do so using the same auxiliary value used in Section 5.2:

```
def crq (r : ℝ) (q : ℕ) :=
((one_coeff_poly q).eval₂ coe r) / r ^ ((q-1)/3)
```

It is convenient to first establish a bound in the case where n is divisible by 3. The proof of this bound combines lemma_13_8 and lemma_13_10 with some elementary results about geometric sums.

```
theorem lemma_13_11 (N : ℕ) {r : ℝ} (hr : 0 < r) (hr2 : r < 1) :
  m (3*N) ((q-1)*N) ≤ (1/(1-r)) * ((crq r q))^(3*N)
```

Recall that m n d is the number of monomials in n variables with total degree at most d. This number is clearly monotonic increasing in d; it is also easy to recognize that it is monotonic increasing in n, although formalizing this takes slightly more work. From these considerations and the previous lemma, we deduce:

```
theorem theorem_13_13 (n : ℕ) {r : ℝ} (hr : 0 < r) (hr2 : r < 1) :
  (m n ((q - 1)*n / 3)) ≤ ((crq r q)^2 / (1 - r)) * (crq r q)^n
```

As earlier, we can now derive from elementary calculus:

```
theorem lemma_13_15 : ∃ r : ℝ, 0 < r ∧ r < 1 ∧ crq r q < q
```

Instantiating theorem_13_13 with this r, invoking theorem_12_1, and abstracting the type parameter α leads us to the theorem general_cap_set.

We finally return to the original cap set problem with q = 3. Since we have used the same function crq as in Section 5.2, we can optimize it in r in the same way to find the value r := (real.sqrt 33 - 1) / 8. Aided by the numeral and ring normalization tactics in mathlib, we establish that 0 < r < 1 and that crq r 3 = ((3 / 8)^3 * (207 + 33*real.sqrt 33))^(1/3). We compute the rough approximation (crq r q)^2 / (1 - r) ≤ 198 to conclude:

```
theorem cap_set {n : ℕ} {A : finset (fin n → ℤ/3ℤ)} :
  (∀ x y z ∈ A, x + y + z = 0 → x = y ∧ x = z) →
    A.card ≤ 198 * (((3/8) ^ 3 * (207 + 33 * sqrt 33)) ^ (1/3)) ^ n
```

Nine Chapters of Analytic Number Theory in Isabelle/HOL

Manuel Eberl
Technical University of Munich, Boltzmannstraße 3, Garching bei München, Germany
https://www21.in.tum.de/~eberlm
manuel.eberl@tum.de

Abstract

In this paper, I present a formalisation of a large portion of Apostol's *Introduction to Analytic Number Theory* in Isabelle/HOL. Of the 14 chapters in the book, the content of 9 has been mostly formalised, while the content of 3 others was already mostly available in Isabelle before.

The most interesting results that were formalised are:

- The Riemann and Hurwitz ζ functions and the Dirichlet L functions
- Dirichlet's theorem on primes in arithmetic progressions
- An analytic proof of the Prime Number Theorem
- The asymptotics of arithmetical functions such as the prime ω function, the divisor count $\sigma_0(n)$, and Euler's totient function $\varphi(n)$

2012 ACM Subject Classification Mathematics of computing → Mathematical analysis

Keywords and phrases Isabelle, theorem proving, analytic number theory, number theory, arithmetical function, Dirichlet series, prime number theorem, Dirichlet's theorem, zeta function, L functions

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.16

Supplement Material The proof developments in the *Archive of Formal Proofs* (AFP) that this work refers to are listed in the bibliography. Additionally, a precise overview of what material from the book has been formalised and which theorems in the book correspond to which theorems in the formalisation can be found at 10.5281/zenodo.3262266.

Funding This work was supported by DFG grant NI 491/16-1. Part of it was conducted during a research visit in collaboration with the ALEXANDRIA project (ERC grant 742178).

Acknowledgements I would like to thank John Harrison for doing all the incredibly hard work of creating an extensive library of complex analysis in HOL Light – the first of its kind – and Larry Paulson and Wenda Li for porting it to Isabelle/HOL and extending it even further. Without these efforts, my work would not have been possible. Larry Paulson also started off the analytic proof of the Prime Number Theorem in Isabelle and allowed me to take over and replace it with a more high-level approach. I also thank Jeremy Avigad and Johannes Hölzl, who commented on a draft of this document, and the anonymous reviewers, who gave a number of helpful suggestions.

1 Introduction

The formalisation of Apostol's book in Isabelle/HOL started from the simple desire to have more properties about Euler's φ function available in the system. However, Apostol's style turned out to be very amenable to formalisation, and the subject matter was both of great interest as a basis for further development of number theory in Isabelle and as a case study for Isabelle's libraries on asymptotics and complex analysis. After 1.5 years of a highly part-time one-person effort, most of the book (and quite a bit of material that goes beyond the book) has been formalised:

- Chapters 1, 5, and 9 consist of fairly basic material (e. g. GCDs, congruences, Quadratic reciprocity), most of which was already available in the Isabelle/HOL library.
- The results from Chapters 2, 3, 4, and 6 have been formalised in their entirety.
- Chapters 10, 11, and 12 have been formalised with some omissions.
- For Chapters 7 and 13 (Dirichlet's Theorem and PNT), equivalent results have been formalised using a different approach.
- Chapters 8 and 14 have been skipped. The former is being actively worked on; the latter concerns number partitions and has little connection to the other material in the book.
- Various interesting results from other sources (e. g. Hildebrand's lecture notes [21]) that are not proven in Apostol's book have also been formalised.

For more precise information on this, see the supplementary material listed before.

I put particular focus on developing a usable library of Dirichlet series on one hand and concrete results about the distribution of primes on the other. As the development is much too large to be presented here in full, I will go through a high-level description of some of the most interesting material. Special attention will be given to parts where I encountered difficulties or chose a different route than Apostol did in his book. Proofs will only be given in the form of very brief sketches, e. g. when it is necessary in order to understand difficulties in formalising them. I would like to refer readers who are interested in the actual *proofs* either to my commented formalisation in the *Archive of Formal Proofs* (AFP) [13, 12, 15, 18, 16] or to the numerous excellent textbooks and lecture notes on the subject [2, 21, 7]. I chose not to show Isabelle code in this presentation since the main results are very close to mathematical notation (e. g. Re $s \geq 1 \Longrightarrow s \neq 1 \Longrightarrow$ zeta $s \neq 0$) and showing the Isabelle code would therefore not provide much additional insight.

Let me now give an outline of the sections to follow: Section 2 lists related work. Section 3 defines formal Dirichlet series and their connection to complex-analytic functions. Section 4 introduces multiplicative characters and Dirichlet characters. Section 5 builds on the Dirichlet series library to treat various L functions, such as the famous Riemann ζ function. Section 6 describes my formalisation of the Prime Number Theorem (PNT). Section 7 gives some more examples of interesting results that were formalised. Section 8 gives an overview of the size of various parts of my formalisation and the effort involved in creating it. Lastly, in Section 9, some conclusions are drawn from this project.

▶ Remark 1. Any sum $\sum_p$ or product $\prod_p$ is to be understood to run over prime p only.

2 Related Work

The first formalisation of a result related to this work was that of the PNT in Isabelle/HOL by Avigad *et al.* [5] in 2007. They formalised the elementary Selberg–Erdős proof.[1] Carneiro formalised the same proof in Metamath [11].

Harrison developed the first (and until now only) formalisation of an analytic proof of the PNT in 3,600 lines [20] of HOL Light. He followed Newman's presentation, which I also did.[2]

[1] Unfortunately, this work was never submitted to the AFP and has not been maintained since then. At the time of writing this paper, the proofs are 12 years old; the formalisation comprises almost 27,000 lines, and many of them are unstructured proof scripts. Bringing them up to date to work with a modern version of Isabelle would be a massive undertaking. However, much of the more general material developed by Avigad *et al.* was moved to Isabelle's library, and for a considerable part of the remaining material, equivalent results are now already a part of the Isabelle library or my work anyway.

[2] Paulson later ported Harrison's development to Isabelle/HOL, but the ported proofs were lengthy and not very readable, so he and I decided that it would be better to redo them in a more high-level style, which I did. Only a few small lemmas were kept.

Harrison also proved Dirichlet's Theorem [19] and I used some of the high-level structure of his development as an inspiration for mine. Moreover, formalisations of Bertrand's postulate exist by Harrison in HOL Light, by Théry in Coq [27], by Riccardi in Mizar [26], by Carneiro in Metamath [10], by Asperti and Ricciotti in Matita [4], and by Biendarra and Eberl in Isabelle/HOL [9].

The big difference between these formalisations and the present one is that this one contains not just *one* result and the material required for it, but the majority of a textbook on the subject. Many proofs are much simpler and more "high-level" through the use of *Dirichlet series* and Isabelle's advanced machinery for asymptotic reasoning.

3 Dirichlet Series

The central objects in analytic number theory are *Dirichlet series*. These are the main tools that set apart my approach to formalised number theory from that of previous formalisation work in multiplicative number theory like that by Avigad *et al.*, Harrison, and Carneiro.

▶ **Definition 2** (Formal Dirichlet series). *A formal series of the form* $f(s) = \sum_{n=1}^{\infty} a_n n^{-s}$ *is called a Dirichlet series. Typically, the* a_n *are real or complex (we will mostly look at the complex case). The Dirichlet series over* $\mathbb{R}$ *or* $\mathbb{C}$ *form a commutative ring with the obvious choices for 0, 1, and addition. Multiplication is defined as* $(\sum_{n=1}^{\infty} a_n n^{-s}) \cdot (\sum_{n=1}^{\infty} b_n n^{-s}) = \sum_{n=1}^{\infty} (\sum_{k \cdot l = n} a_k b_l) n^{-s}$. *Also,* $\sum_{n=1}^{\infty} a_n n^{-s}$ *has a multiplicative inverse iff* $a_1 \neq 0$.

▶ **Theorem 3** (Convergence of Dirichlet series). *Each Dirichlet series has abscissæ of convergence* σ_c *and absolute convergence* σ_a *such that the infinite sum corresponding to it is absolutely summable for* $\operatorname{Re}(s) > \sigma_a$*, conditionally summable for* $\operatorname{Re}(s) \in (\sigma_c, \sigma_a)$*, and divergent for* $\operatorname{Re}(s) < \sigma_c$ *(where* $\operatorname{Re}(s)$ *denotes the real part of* s*). The* σ_c *and* σ_a *satisfy* $\sigma_c \leq \sigma_a \leq \sigma_c + 1$ *and may be* $\pm\infty$.

Much like formal power series (i. e. ordinary generating functions) for combinatorics, Dirichlet series are closely associated with number theory. Like generating functions, they are of great interest as mere formal objects, but when they converge, their interpretation as a complex-valued function is also enormously useful, as we will see.

Various formal analogues of analytic operations can be defined for Dirichlet series e. g. reciprocal, derivative, integral, $\exp(f(s))$, $\ln f(s)$, $f(s + s_0)$, $f(m \cdot s)$, and subseries. These have similar properties to their analytic counterparts (e. g. $\exp(f(s))' = f'(s)\exp(f(s))$) even when they do not converge. When they do converge, they typically agree with their analytic counterparts. This allows one to prove properties of the analytic functions by reasoning on the formal level and vice versa.

There are 4,800 lines of material on formal Dirichlet series in my formalisation. This is far too much to show here, so I simply say that it contains all of Chapter 11 in Apostol's book and more, except for Sections 11.10 and 11.11. I will only show a few small examples that illustrate the aforementioned interplay of the formal and the analytical level:

One example of using formal Dirichlet series to derive an analytic result is this:

▶ **Theorem 4.** *Let* $\omega(n)$ *be the number of distinct prime factors of* n *and* $\mu(n)$ *the Möbius* μ *function, i. e.* $(-1)^{\omega(n)}$ *if* n *is square-free and 0 otherwise. Then* $\sum_{n=1}^{\infty} \mu(n)/n^2 = 6/\pi^2$.

Proof. Consider the formal series $\zeta(s) := \sum_{n=1}^{\infty} n^{-s}$ and $M(s) := \sum_{n=1}^{\infty} \mu(n) n^{-s}$. It is clear that they both converge absolutely for $\operatorname{Re}(s) > 1$ by the comparison test. It is easy to show $\sum_{d|n} \mu(d) = [n = 1]$, i. e. $\zeta(s)M(s) = 1$ holds formally [2]. Thus, it also holds analytically for $\operatorname{Re}(s) > 1$ so that we have $\zeta(2)M(2) = 1$ and therefore $M(2) = 1/\zeta(2)$, where $\zeta(2)$ – the famous *Basel problem* – has the well-known value $\pi^2/6$ [6]. ◀

The following theorem allows us to transfer an analytic equality to the formal level:

▶ **Theorem 5** (Uniqueness of Dirichlet series)**.** *Let $f(s)$, $g(s)$ be two formal Dirichlet series whose abscissa of convergence is $< \infty$. If there exists a sequence s_k with $\mathrm{Re}(s_k) \longrightarrow \infty$ and $\forall k.\ f(s_k) = g(s_k)$, then $f(s)$ and $g(s)$ are equal as formal Dirichlet series.*

▶ Remark 6. In Isabelle, the condition on the existence of the sequence s_k is replaced by the following equivalent and more concise formulation using *filters* [22]:

$$\exists_F\ s \textbf{ in } \mathrm{Re} \textbf{ going-to } \text{at-top}.\ f(s) = g(s)$$

The filter "f **going-to** F" is the contravariant image of F under f, i. e. "Re **going-to** at-top" describes the neighbourhood of complex numbers with "sufficiently large" real part.

The "$\exists_F\ x$ **in** F. $P(x)$" notation stands for "$P(x)$ holds frequently in F", i. e. the complement of P is not in the filter F. Less formally, one could say that $P(x)$ holds "again and again". In the case of "Re **going-to** at-top", this means that for any $C \in \mathbb{R}$, there exists an s with $\mathrm{Re}(s) \geq C$ for which the property is fulfilled.

▶ **Definition 7** (Truncation operator)**.** *For a Dirichlet series $f(s) = \sum_{n=1}^{\infty} a_n n^{-s}$, let $T_m(f(s)) = \sum_{n=1}^{m} a_n n^{-s}$ denote the m-th order truncation of $f(s)$. The result is a Dirichlet polynomial, i. e. a Dirichlet series with only finitely many non-zero coefficients.*

▶ **Theorem 8.** $f(s) = g(s) \longleftrightarrow \forall m.\ T_m(f(s)) = T_m(g(s))$.

The following is an instance where these theorems are used to avoid a lot of complicated reasoning on the formal level by leveraging a result on the analytic level:

▶ **Theorem 9.** *For any (not necessarily convergent) Dirichlet series $f(s)$ and $g(s)$, we have* $\exp(f(s) + g(s)) = \exp(f(s)) \exp(g(s))$.

Proof. It is clear that the result holds analytically whenever the series converge, so if the series have a non-empty half-plane of convergence, they must be equal by Theorem 5. The question is how to show this if we do *not* know whether the series converge anywhere.

The key is to use Theorem 8 together with $T_m(\exp(h(s))) = T_m(\exp(T_m(h(s))))$. This allows us to assume w. l. o. g. that the series in question are Dirichlet polynomials and therefore converge everywhere. ◀

▶ Remark 10. This technique of showing an equality on Dirichlet series by showing that it holds for all Dirichlet polynomials works if the two sides of the equation in question are continuous functions w. r. t. the topology on formal Dirichlet series, i. e. each coefficient of the result only depends on finitely many coefficients of the input. The topological structure of Dirichlet series was not formalised yet, but this is a useful fact to keep in mind.

The following is another important theorem connecting a series with the function it defines that we will use later:

▶ **Theorem 11** (Pringsheim–Landau)**.** *Let $f(s)$ be a Dirichlet series with non-negative real coefficients and $\sigma_a \neq \pm\infty$. Then $f(s)$ has a singularity at σ_a.*

Conversely, if $f(s)$ has an an analytic continuation to some half-plane $\{s \mid \mathrm{Re}(s) > c\}$, then $\sigma_a \leq c$. In particular, if $f(s)$ is entire, the series must converge everywhere.

4 Characters of a Finite Abelian Group

The next concept we shall explore is that of a *multiplicative character*, which will be needed to prove Dirichlet's theorem. For this section, let $G = (G, \cdot, 1)$ be a finite abelian group.

▶ **Definition 12** (Multiplicative character). *A character is a group homomorphism $\chi : G \to \mathbb{C}^\times$, i. e. $\chi(1) = 1$ and $\chi(a \cdot b) = \chi(a)\chi(b)$ for any $a, b \in G$. The character χ_0 that maps every element to 1 is called the principal character.*

For the necessary group theory, I use the `HOL-Algebra` library by Ballarin, which models a group as a record containing entries for the operation $\cdot$, the neutral element 1, and an explicit carrier set (which does not have to be the full type universe). The latter is necessary in HOL because notions such as subgroups cannot easily be expressed without explicit carrier sets. The fact that such a record indeed describes a group is then formalised as a *locale* [8] called *group*, which fixes such a record and assumes that all the usual group axioms hold.

A character can then be defined as a locale that extends the *group* locale by fixing a function $\chi :: \alpha \to \mathbb{C}$ (where α is the type of the group elements) and assuming that the two homomorphism properties mentioned above hold for χ. For convenience, I only assume $\chi(1) \neq 0$ (from which $\chi(1) = 1$ easily follows) and I additionally require $\chi(x) = 0$ for any x not in the carrier of the group. The latter is to ensure *extensionality*, i. e. two characters are equal as HOL values iff they return the same result on every group element.

▶ **Definition 13** (Pontryagin dual group). *Denote the set of characters of G by $\widehat{G}$. Then $\widehat{G}$ forms a group $\widehat{G} := (\widehat{G}, \cdot, \chi_0)$ with point-wise multiplication and χ_0 as the identity. This group is called the Pontryagin dual group of G.*

▶ **Theorem 14** (Number of characters). $|\widehat{G}| = |G|$

In Isabelle, the proof is by induction on the subgroups of G, using a custom induction rule inspired by Apostol's proof. The idea here is to successively "adjoin" elements, i. e. for a subgroup H and some $x \in G \setminus H$, we form $\langle H; x\rangle$, the subgroup generated by $H \cup \{x\}$:

▶ **Lemma 15** (Induction on a group). *Let $G = (G, \cdot, 1)$ be a group and H some subgroup of G. Let P be some property on groups. If $P(H)$ holds and $P(H')$ implies $P(\langle H'; x\rangle)$ for all subgroups $H' \supseteq H$ and all $x \in G \setminus H'$, then $P(G)$ holds.*

I use this to show a stronger version of Theorem 14 that is just as easy to show:

▶ **Theorem 16** (Number of character extensions). *Let H be a subgroup of G and $\chi \in \widehat{H}$. Let*

$$C(G) := \{\chi' \in \widehat{G} \mid \forall x \in H.\ \chi'(x) = \chi(x)\}$$

denote the set of characters on G that agree with χ on H. Then $|C(G)| \cdot |H| = |G|$, i. e. there are precisely $|G|/|H|$ ways to extend a character on H to a character on G.

Proof. By straightforward induction according to Lemma 15, using the bijection

$$f : C(\langle H'; x\rangle) \to C(H') \times \{z \in \mathbb{C} \mid z^n = 1\}, \quad f(\chi) = (y \mapsto \chi(y), \chi(x))$$

in the induction step. ◀

Theorem 14 follows directly by taking $H = (\{1\}, \cdot, 1)$. Another useful corollary is this:

▶ **Corollary 17.** *For any $x \neq 1$, there exists a $\chi \in \widehat{G}$ such that $\chi(x) \neq 1$.*

With this, we can prove a nice property that Apostol does not cover at all:

▶ **Theorem 18** (Isomorphism to the double dual). *G is isomorphic to its double dual via the natural isomorphism $\nu : G \to \widehat{\widehat{G}}$, $\nu(x) = (\chi \mapsto \chi(x))$.*

This isomorphism is useful for the next properties:

▶ **Theorem 19** (Orthogonality relations). *For any $\chi \in \widehat{G}$ resp. $x \in G$, we have:*

$$\sum_{x \in G} \chi(x) = \begin{cases} |G| & \text{if } \chi = \chi_0 \\ 0 & \text{otherwise} \end{cases} \quad (1) \qquad \sum_{\chi \in \widehat{G}} \chi(x) = \begin{cases} |G| & \text{if } x = 1 \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

Apostol's proof for (1) is very simple and straightforward to formalise. In order to show (2) from (1), Apostol represents the set of characters of G as a *character table* of G, i. e. a $|G| \times |G|$ complex matrix. If we denote this matrix by A, (1) shows that $AA^* = nI$ (where A^* is the conjugate transpose of A). By simple linear algebra, $A^*A = nI$ and thus (2) follows.

Formalising this argument would have required importing Isabelle's linear algebra library and doing some tedious work to relate the matrix to the characters, so I chose another route: One *could* prove (2) relatively easily using the same induction principle as before in about 70 lines, but the easiest way is to simply use Pontryagin duality: (2) is, in fact, nothing but the dual of (1), with $\widehat{G}$ for G and $\nu(x)$ for χ. This requires only 6 lines of Isabelle code.

▶ **Definition 20** (Dirichlet character). *A Dirichlet character χ for the modulus $m \in \mathbb{N}_{>1}$ is a character of the multiplicative group of the residue ring $\mathbb{Z}/m\mathbb{Z}$. For convenience, χ is represented as a periodic function of type $\mathbb{N} \to \mathbb{C}$ with period m, i. e. $\chi(k) = \chi(k \bmod m)$.*

▶ Remark 21. Apostol's and my treatment of characters are quite elementary. There is an alternative, more group-theoretic view on this: It is straightforward to show that $\widehat{G_1 \times G_2} \simeq \widehat{G}_1 \times \widehat{G}_2$ and that $\widehat{C}_n \simeq C_n$ for cyclic groups C_n. Together with the *Fundamental Theorem of Finite Abelian Groups*, this implies a stronger variant of Theorem 14, namely $\widehat{G} \simeq G$. However, unlike with $\widehat{\widehat{G}} \simeq G$, the isomorphism is not natural and establishing it indeed requires the Fundamental Theorem, which is currently not available in `HOL-Algebra`. Since the formal proofs that were presented in this section are still reasonably short, I do not think this is a big problem.

5 The L Functions

In this section, we will look at four functions from the class of L functions: Riemann's ζ function, Dirichlet's L function, Hurwitz's ζ function, and the periodic ζ function. These are all complex-valued functions that are defined by an infinite sum for $\mathrm{Re}(s) > 1$ and can be analytically or meromorphically continued to the entire complex plane.

▶ **Definition 22** (Riemann's ζ function). *For $\mathrm{Re}(s) > 1$, the Riemann ζ function is given by the Dirichlet series $\zeta(s) = \sum_{n=1}^{\infty} n^{-s}$.*

▶ **Definition 23** (Dirichlet L functions). *Let χ be a Dirichlet character for the modulus $m > 0$. Then $L(s, \chi)$ is given by the Dirichlet series $L(s, \chi) = \sum_{n=1}^{\infty} \chi(n) n^{-s}$ for $\mathrm{Re}(s) > 1$ if $\chi = \chi_0$ and for $\mathrm{Re}(s) > 0$ if $\chi \neq \chi_0$.*

We immediately get the following properties for free from the Dirichlet series library:

▶ **Theorem 24.** *Let $\Lambda(n)$ denote the von Mangoldt function. Then, if $\mathrm{Re}(s) > 1$:*

$$\zeta(s) = \prod_p \frac{1}{1-p^{-s}} \qquad \zeta'(s) = -\sum_{n=1}^{\infty} \frac{\ln n}{n^s} \qquad \ln \zeta(s) = \sum_{n=2}^{\infty} \frac{\Lambda(n)}{\ln n \cdot n^s}$$

$$L(s,\chi) = \prod_p \frac{1}{1-\chi(p)p^{-s}} \qquad L'(s,\chi) = -\sum_{n=1}^{\infty} \frac{\chi(n)\ln n}{n^s} \qquad \ln L(s,\chi) = \sum_{n=2}^{\infty} \frac{\chi(n)\Lambda(n)}{\ln n \cdot n^s}$$

However, $\zeta(s)$ and $L(s,\chi)$ can be defined on a larger domain:

▶ **Theorem 25** (Analytic continuation of $\zeta(s)$ and $L(s,\chi)$)**.**
1. *$\zeta(s)$ can be continued to an analytic function on $\mathbb{C} \setminus \{1\}$ with a simple pole at $s = 1$.*
2. *For non-principal χ, $L(s,\chi)$ can be continued to an entire function.*
3. *For $\chi = \chi_0$, we have $L(s,\chi_0) = \zeta(s) \cdot \prod_{p|m}(1-p^{-s})$, i. e. $L(s,\chi_0)$ is equal to $\zeta(s)$ up to an entire factor and is therefore also analytic on $\mathbb{C} \setminus \{1\}$ with a simple pole at $s = 1$.*

The difficult part here is to actually construct the analytic continuations. To do this uniformly and without duplication of work, Newman uses a generalisation of $\zeta(s)$:

▶ **Definition 26** (Hurwitz's ζ function)**.** *Let $a \in \mathbb{R}_{>0}$ and $\mathrm{Re}(s) > 1$. Then $\zeta(s,a)$ is given by the (non-Dirichlet) series $\zeta(s,a) = \sum_{n=0}^{\infty} (n+a)^{-s}$.*

Apostol only considers $\zeta(s,a)$ for $a \in (0,1]$ since some results only hold for $a \leq 1$ and the case of $a > 1$ can be reduced to $a \in (0,1]$. It is, however, useful to allow also $a \geq 1$ – e. g. in Newman's proof of the PNT, as noted already by Harrison [20].

▷ Claim 27. $\zeta(s,a)$ can be continued analytically to $\mathbb{C} \setminus \{1\}$ with a simple pole at $s = 1$. Both Riemann's ζ function and the Dirichlet L functions can easily be expressed in terms of $\zeta(s,a)$, so that a continuation for Hurwitz's ζ also yields continuations for the other two [2].

The main question now is therefore how to construct the continuation of $\zeta(s,a)$.

5.1 Analytic Continuation of Hurwitz's ζ Function

Apostol constructs the continuation using an integral along an infinite contour. I did formalise this eventually (see Theorem 36), but when I first defined $\zeta(s,a)$ in Isabelle, this approach seemed quite daunting to me, so I chose another route that seems to be folklore [2, 24] and that I discovered independently: Since $\zeta(s,a)$ is defined for $\mathrm{Re}(s) > 1$ by an infinite sum $\sum_{n=0}^{\infty}(n+a)^{-s}$ and the corresponding improper integral $\int_0^{\infty}(x+a)^{-s}\,\mathrm{d}x$ is easy to compute, the Euler–MacLaurin summation formula [14] suggests itself. Applying it, we obtain

$$\sum_{n=0}^{\infty}(s+a)^{-n} - \frac{a^{1-s}}{s-1} = \frac{a^{-s}}{2} + \sum_{i=1}^{N} \frac{B_{2i}}{(2i)!}\, a^{-s-2i+1}\, s^{\overline{2i-1}} + \frac{(-1)^{2N} s^{\overline{2N+1}}}{(2N+1)!} \int_0^{\infty} P_{2N+1}(t) \cdot (t+a)^{-s-2N-1}\,\mathrm{d}t \tag{3}$$

where $s^{\overline{k}}$ denotes the rising factorial, B_k is the k-th Bernoulli number, and $P_k(t)$ is the periodic version of the Bernoulli polynomial $B_k(t)$, i. e. $P_k(t) = B_k(t - \lfloor t \rfloor)$.

The right-hand side is now actually analytic on a larger domain: all terms except the last one are clearly entire functions in s; the only non-obvious term is the integral in the last summand. Leibniz's rule shows that the definite integral $\int_0^b$ is analytic in s, and an integral version of the Weierstraß M-test then shows that the improper integral $\int_0^{\infty}$ is uniformly convergent and therefore analytic in s for $\mathrm{Re}(s) > -2N$.

Let us write $\mathit{prezeta}_N(s,a)$ for the right-hand side. This is then a function in s that is analytic for $\mathrm{Re}(s) > -2N$ and that also fulfils

$$\mathrm{prezeta}_N(s,a) = \sum_{n=0}^{\infty} (s+a)^{-n} - \frac{a^{1-s}}{s-1} \qquad \text{for } \mathrm{Re}(s) > 1\ .$$

This means that two functions $\mathit{prezeta}_M$ and $\mathit{prezeta}_N$ will always agree on $\mathrm{Re}(s) > 1$, and by analytic continuation they will then also agree on their entire domain, i. e. for all s with $\mathrm{Re}(s) > -2\max(M,N)$. We can therefore define a full analytic continuation to all of $\mathbb{C}$ by choosing N "big enough" for each input, i. e. we define:

$$\mathrm{prezeta}(s,a) := \mathrm{prezeta}_{\max(0,1-\lceil \mathrm{Re}(s)/2 \rceil)}(s,a)$$

This function is entire and agrees with any of the $\mathit{prezeta}_N(s,a)$ for all s with $\mathrm{Re}(s) > -2N$. Thus, it is an analytic continuation of the left-hand side of (3) so that we can simply define

$$\zeta(s,a) := \mathrm{prezeta}(s,a) + \frac{a^{1-s}}{s-1}$$

to obtain the Hurwitz ζ function on all of $\mathbb{C} \setminus \{1\}$. For convenience, I choose $\zeta(1,a) = 0$ as is often done in HOL-based systems (cf. $\Gamma(-n)$ for $n \in \mathbb{N}$ in Isabelle/HOL and HOL Light). The advantage of the Euler–MacLaurin approach is that it is simple to implement because all of the "heavy lifting" has already been done in the AFP entry on the Euler–MacLaurin formula.

Various basic properties of the Hurwitz and Riemann ζ functions then follow in a straightforward way, of which I show some notable ones here:

▶ **Theorem 28** (Special values of ζ). *For any $n \in \mathbb{N}_{\geq 0}$, we have:*

$$\zeta(a,-n) = -\frac{B_{n+1}(a)}{n+1} \qquad \zeta(-n) = -\frac{B_{n+1}}{n+1} \qquad \zeta(2n) = \frac{(-1)^{n+1} \cdot B_{2n} \cdot (2\pi)^{2n}}{2(2n)!}$$

where $B_n = B_n(1)$ are the Bernoulli numbers with $B_1 = \frac{1}{2}$. In particular, this implies the famous $\zeta(-1) = -\frac{1}{12}$ and the Basel problem $\zeta(2) = \pi^2/6$.

▶ **Theorem 29** (Integral representation for $\zeta(s,a)$). *For any s with $\mathrm{Re}(s) > 1$, we have:*

$$\Gamma(s)\zeta(s,a) = \int_0^\infty \frac{t^{s-1}e^{-at}}{1-e^{-t}}\, dt$$

5.2 The Non-Vanishing of $\zeta(s)$ and $L(s,\chi)$ for $\mathrm{Re}(s) = 1$

The following is a core ingredient in the Prime Number Theorem and Dirichlet's Theorem:

▶ **Theorem 30.** *For any s with $\mathrm{Re}(s) \geq 1$, we have $\zeta(s) \neq 0$ and $L(s,\chi) \neq 0$.*

The case of $\mathrm{Re}(s) > 1$ is a simple consequence of the Euler product formula for $\zeta(s)$ and $L(s,\chi)$ (cf. Theorem 24); the difficult part is the case $\mathrm{Re}(s) = 1$. For this, I formalised the very simple proof presented by Newman [25], whose key ingredient is the aforementioned Pringsheim–Landau theorem (see Theorem 11). This proof is stunningly short and its high-level reasoning translates well to Isabelle/HOL now that a library of Dirichlet series is available. The gain is most striking for the Dirichlet L function, where Apostol's proof only treats the case of $s = 1$, and even that proof is still more complicated than Newman's and

involves lengthy complicated "Big-O" reasoning. Indeed, in a first version of the formalisation, I formalised Apostol's proof, but it was considerably longer and messier than the new version – with the added bonus that the new one is also more general.

Harrison also only proves $L(1,\chi) \neq 0$ – indeed, he does not define $L(s,\chi)$ at all; he defines only $L(1,\chi)$ since that is all that is required for Dirichlet's theorem. Despite this and the much higher verbosity of structured Isabelle proofs compared to HOL Light,his proof is longer than mine. The reason for this is that his proof is very elementary and uses very little library material while mine builds on a large library of Dirichlet series. However, I think that the comparison is still not entirely unjustified since all of this material is sufficiently general to be called "library material" (as opposed to technical lemmas specifically designed for this one proof), and building sufficiently large and general libraries to make proofs like this cleaner and easier is, after all, one of our goals in formalisation.

5.3 Hurwitz's Formula

More as a challenge to myself and the Isabelle libraries, I chose to formalise another non-trivial property of the ζ functions:

▶ **Theorem 31** (Reflection formula for $\zeta(s)$)**.** *For $s \notin \{0,1\}$, we have:*

$$\frac{1}{\Gamma(s)} \cdot \zeta(1-s) = 2(2\pi)^{-s} \cos(\pi s/2)\zeta(s)$$

Note that while $\Gamma(s)$ has poles at $s \in \mathbb{Z}_{\leq 0}$, its reciprocal $1/\Gamma(s)$ is entire, so the formula holds even for $s \in \mathbb{Z}_{<0}$.

This formula is a corollary of a more general one for $\zeta(s,a)$ known as *Hurwitz's formula*:

▶ **Theorem 32** (Hurwitz's formula)**.** *Let $a \in (0,1)$ and $s \in \mathbb{C} \setminus \{0\}$ with $a \neq 1 \vee s \neq 1$. Then:*

$$\frac{1}{\Gamma(s)} \cdot \zeta(1-s,a) = (2\pi)^{-s}\big(i^{-s}F(s,a) + i^{s}F(s,-a)\big)$$

Here, $F(s,a)$ is the *periodic ζ function*, which we still have to define:

▶ **Definition 33** (Periodic ζ function)**.** *For $\mathrm{Re}(s) > 1$, the periodic ζ function $F(s,a)$ is given by the Dirichlet series $F(s,a) = \sum_{n=1}^{\infty} e^{2i\pi na} n^{-s}$.*

▷ Claim 34. $F(s,a)$ is called *periodic* because $F(s,a+n) = F(s,a)$ for any integer n. For non-integer a, the above series converges for $\mathrm{Re}(s) > 0$ and can be continued to an entire function. For integer a, it is simply the Riemann ζ function.

Apostol does not discuss the analytic continuation of $F(s,a)$ at all, but it seemed useful to me to do this nonetheless. The strategy I used to construct the continuation of $F(s,a)$ for non-integer a is somewhat interesting: Theorem 32 can be rearranged to give a formula that expresses $F(s,a)$ in terms of $\zeta(1-s,a)$ and $\zeta(1-s,1-a)$:

▶ **Theorem 35.** *Let $a \in (0,1)$ and $s \in \mathbb{C} \setminus \mathbb{N}$. Then:*

$$F(s,a) = i(2\pi)^{s-1}\Gamma(1-s)\big(i^{-s}\zeta(1-s,a) - i^{s}\zeta(1-s,1-a)\big)$$

We therefore proceed like this (assuming w. l. o. g. $a \in (0,1)$):

1. Show Theorem 32 for $\mathrm{Re}(s) > 1$ (where F is simply given by its Dirichlet series).
2. Use this to show Theorem 35 for $\mathrm{Re}(s) > 1$.

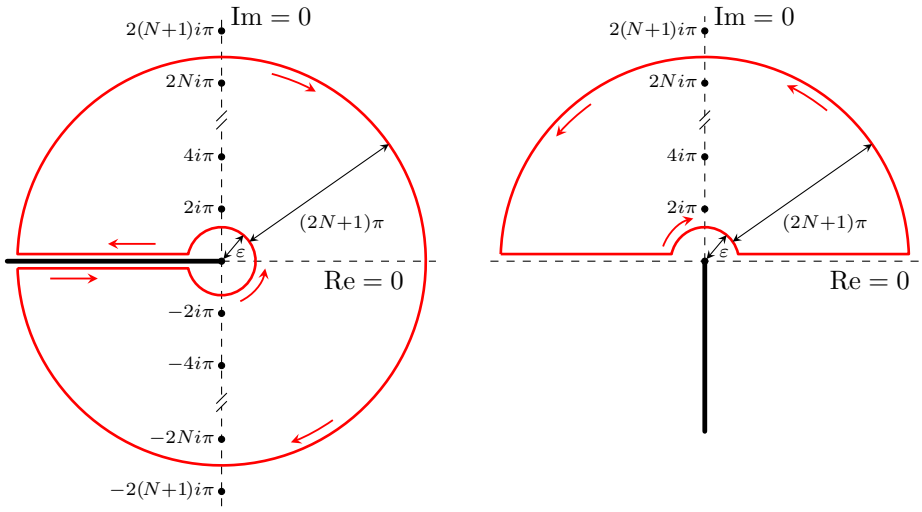


Figure 1 Apostol's integration contour and my modified version for proving Hurwitz's formula ($\varepsilon < 2\pi$). The black dots are the poles of the integrand; the thick black line is its branch cut. Note that in both cases, the line segments of the contour lie *on* the real axis despite the small gap in the illustration.

3. Use the right-hand side of Theorem 35 as the definition of $F(s,a)$ for $s \notin \mathbb{N}$. Compatibility with the Dirichlet series definition follows by analytic continuation.
4. Since the Dirichlet series definition covers $\mathrm{Re}(s) > 0$ and the new definition covers $\mathbb{C} \setminus \mathbb{N}$, the only point left is $s = 0$, which is a removable singularity that can be eliminated via

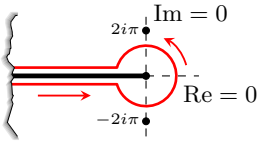
$$F(0,a) := \lim_{s\to 0} F(s,a) = \frac{i}{2\pi}\big(\mathrm{prezeta}(1,a) - \mathrm{prezeta}(1,1-a) + \ln(1-q) - \ln q\big) - \frac{1}{2}\,.$$

5. Extend the validity of Theorems 32 and 35 to their full domains by analytic continuation.

The only difficult part here is the first step, which we shall look at now. First of all, we require the contour integral representation for $\zeta(s,a)$ mentioned in Section 5.1:

▶ **Theorem 36.** *For any $s \in \mathbb{C} \setminus \{1\}$, we have*

$$\zeta(s,a) = \frac{\Gamma(1-s)}{2i\pi} \int_{\circlearrowleft} \frac{z^{s-1}e^{az}}{1-e^z}\,dz \qquad \textit{where } \circlearrowleft =$$



if the inner circle has radius $\varepsilon < 2\pi$. This continues $\zeta(s,a)$ analytically to $\mathbb{C} \setminus \{1\}$.

Proof. Due to analytic continuation, we can assume $\mathrm{Re}(s) > 1$ w. l. o. g. By homotopy, all contours with a radius $\varepsilon < 2\pi$ yield the same result. Letting $\varepsilon \to 0$, the contribution of the circle vanishes and the $\int_{\circlearrowleft}$ becomes the $\int_0^\infty$ from Theorem 29. ◀

▶ Remark 37. Note that in order to even state this theorem formally, one needs to make the limit inherent in this "improper contour integral" explicit. I chose to decompose the integral as $\int_{\circlearrowleft} = \int_{\curvearrowleft} - \int_{\curvearrowright}$. The two line segments can then be written as Lebesgue integrals $\int_0^{-\infty}$, leaving only two finite circular arcs as the remainder. There is yet another subtlety hidden in this integral that will be discussed in Section 5.3.1.

The proof of Hurwitz's formula for $\mathrm{Re}(s) > 1$ then proceeds by computing this contour integral in a different way using the Residue Theorem. To do this, we first need to approximate it by an integral over a *finite* contour $C_{N,\varepsilon}$ such that $\int_{C_{N,\varepsilon}} \longrightarrow \int_{\text{⊨○}}$ as $N \to \infty$. Figure 1 shows Apostol's choice for $C_{N,\varepsilon}$. Applying the Residue Theorem to this, we get

$$\frac{1}{2i\pi} \int_{C_{N,\varepsilon}} \frac{z^{-s} e^{az}}{1 - e^z} \,\mathrm{d}z = \sum_{z_0} \mathrm{ind}_{C_{N,\varepsilon}}(z_0) \operatorname*{Res}_{z=z_0} \frac{z^{-s} e^{az}}{1 - e^z} \tag{4}$$

where the sum on the right-hand side extends over all the singularities of the integrand (represented by black dots in Figure 1). We can now let $N \to \infty$ so that the contribution of the outer circle vanishes. The integral on the left-hand side is then simply a $\int_{\text{⊨○}}$, which is equal to $\zeta(s, a)/\Gamma(s)$ by Theorem 36, and winding number on the right-hand side -1 for every non-zero pole z_0. Evaluating this sum, we find that it indeed equals $\frac{i^{-s}}{(2\pi)^s} F(s, a) + \frac{i^s}{(2\pi)^s} F(s, -a)$, which concludes the proof of Hurwitz's formula for $\mathrm{Re}(s) > 1$. ⌟

The formalisation of the proof was fairly routine. It is, however, quite large and tedious, containing almost 1,000 lines of proof code compared to 6.5 pages in Newman's book (both including the proof of Theorem 36). This seems to be a common pattern in Isabelle proofs using the Residue Theorem and it is likely due to the many side conditions that need to be shown, many of which are of geometric nature and thus much easier to explain to a human than to a theorem prover. Side conditions like the analyticity of the integrand, on the other hand, can be solved mostly automatically using Isabelle's general-purpose automation together with specialised theorem collections like `analytic_intros`.

Some aspects of the formal proofs of these statements deserve more attention, and we will discuss them now.

5.3.1 Branch Cuts

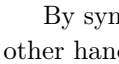
In both theorems, the term z^{-s} is a multi-valued function. It is defined in Isabelle as $e^{-s \ln z}$ where ln is the standard branch of the logarithm, which has a branch cut on the negative real axis. The two lines of Apostol's contour lie directly on this cut, taking different branches of the logarithm (indeed, if they did not, they would simply cancel each other). This makes sense formally when considering the integrand as a multi-valued function in the sense of a Riemann surface, but we do not have any of this analytic machinery in Isabelle.

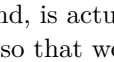
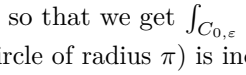
My first idea to circumvent this problem was to resort to some kind of limiting argument by placing the two horizontal lines not directly on the real axis, but some ε above (resp. below) it. However, this would likely have been a very tedious argument to do in Isabelle. I therefore decided to again cut the contour into two halves, similarly to Remark 37. When their integrals are added together, we recover Apostol's contour integral. Due to symmetry, it is actually again enough to look at the upper half (cf. the right part of Figure 1), as the lower one follows by conjugation.

For this upper contour ⌒, we can now integrate over the same branch of the logarithm everywhere. In order to avoid the branch cut of the standard logarithm, I use a different branch $\widetilde{\ln}\, z := \ln(-iz) + \frac{1}{2} i\pi$, whose branch cut lies on the negative imaginary axis, safely away from our contour. I also reversed the contour so that the winding numbers are all 1.

5.3.2 Homotopy

The proof of Theorem 36 uses the fact that the integral along ⊨○ is invariant for all radii $\varepsilon < \pi$. This is because all of these contours are homotopic, i. e. they can be continuously deformed into one another without crossing any of the singularities of the integrand. However, proving that this is the case turned out to be very tedious in Isabelle because there are almost no library theorems that help showing that two composite paths are homotopic.

I circumvented this problem in the following way: First of all, I restricted myself to $\varepsilon < \pi$.[3] Next, since the line segments extending from $-\pi$ to $-\infty$ are the same for all ε, we can ignore them and focus on the finite subcontour . It can be seen that  .

By symmetry, it is enough to show that  is invariant under changes of ε. This, on the other hand, is actually a corollary of (4): If we let $N := 0$, the sum on the right-hand side vanishes so that we get  for all ε. Since  and  (a half circle of radius π) is independent of ε, it follows that  is indeed the same for all ε.

Effectively, this replaces the homotopy argument (which is intuitively obvious for humans and not mentioned at all by Apostol) with a much "heavier" invocation of the Residue Theorem – but since we already applied the Residue Theorem anyway, all that work is already done.

5.3.3 Winding Numbers

The evaluation of the winding numbers $\text{ind}_{C_{N,\varepsilon}}(z_0)$ is also easy for a human: the contour $C_{N,\varepsilon} =$  *clearly* winds counter-clockwise once around each pole $2ni\pi$ with $0 < n \leq N$, and all the other poles are *clearly* completely outside the contour. Proving these things in a theorem prover, on the other hand, is notoriously difficult [20], especially for a more complicated contour like this.

To show that the poles outside the contour really do lie outside (i. e. have winding number 0), I use simple geometric arguments: for the branch cut on the negative imaginary axis, one can draw a vertical line from each point to $-i\infty$ without crossing $C_{N,\varepsilon}$, so the winding number for these points must be 0. Moreover, $C_{N,\varepsilon}$ is contained in a ball of radius $(2N+1)\pi$, which is a convex set that does not contain any of the poles with $n > N$. Thus, these poles must also have winding number 0.

The more difficult part is to show that the winding number for the points inside the contour is 1. Geometric arguments for this are difficult. One approach would be to show that the contour is a closed simple curve (which implies that the winding number must be either -1, 0, or 1) and then weigh the contributions of the four different parts of the curve to show that the overall value must be positive, thus 1. However, to avoid having to do this work, I instead use Li's framework for computing winding numbers in Isabelle [23]. It is based on computing Cauchy indices and comes with some setup to handle combinations of line segments and circular arcs almost automatically, allowing me to prove that the winding numbers are 1 with a mere 18 lines of proof code.

6 The Prime Number Theorem

The formal statement of the PNT is simply the asymptotic estimate $\pi(x) \sim x \ln x$, where $\pi(x)$ is the number of prime numbers $\leq x$. I will now explain, in a high-level way, how the formalised proof works. First of all, let us define the following functions related to primes:

[3] This restriction could easily be lifted by allowing arbitrary radii in (4) instead of just $(2N+1)\pi$.

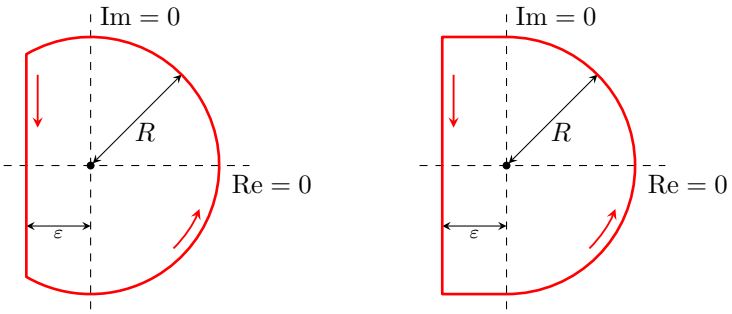


Figure 2 Newman's integration contour in his proof of Ingham's Tauberian theorem and Harrison's modified version. The dot in the middle is the pole of the integrand at the origin.

▶ **Definition 38.**

$$\pi(x) = \sum_{p \le x} 1 = |\{p \mid p \text{ prime} \wedge p \le x\}| \qquad p_n = \text{the } n\text{-th prime number } (p_0 = 2)$$
$$\vartheta(x) = \sum_{p \le x} \ln p \qquad \mathfrak{M}(x) = \sum_{p \le x} \ln p / p$$
$$\psi(x) = \sum_{n \le x} \Lambda(n) = \sum_{p^k \le x} \ln p \qquad M(x) = \sum_{n \le x} \mu(n)$$

$\pi(x)$ is usually called the "prime-counting function". $\vartheta(x)$ and $\psi(x)$ are the first and the second Chebyshev function. $\mu(n)$ is the Möbius μ function. $\mathfrak{M}(x)$ is a non-standard notation I adopted; the function that it denotes is related to Mertens' first theorem and a key part in Newman's proof of the PNT.

▶ **Theorem 39.** *The following are all equivalent formulations of the PNT, i. e. given one of them, it is fairly easy to show the other ones by elementary means:*

$$\pi(x) \sim x / \ln x \qquad \pi(x) \ln \pi(x) \sim x \qquad p_n \sim n \ln n \qquad \vartheta(x) \sim x \qquad \psi(x) \sim x \qquad M(x) \in o(x)$$

Most of these equivalence proofs are quite short, both on paper and in Isabelle.

Newman's approach to prove the PNT is then to prove $\mathfrak{M}(x) = \ln x + c + o(1)$, which implies $\vartheta(x) \sim x$ fairly directly as we shall see. The key ingredient is a Tauberian theorem first proven by Ingham, which we will discuss now.

6.1 Ingham's Tauberian Theorem

A Tauberian theorem is a theorem that allows one to show – under certain conditions – that a series converges in some region if the function that it defines exists there. In our case, Ingham's theorem allows us to show that certain Dirichlet series converge not just to the right of the abscissa of convergence, but *on it* as well. The precise statement is as follows:

▶ **Theorem 40** (Ingham's Tauberian theorem)**.** *Let $F(s) = \sum a_n n^{-s}$ be a Dirichlet series with $a_n \in O(n^{\sigma-1})$ for some $\sigma \in \mathbb{R}$. Then F converges to an analytic function $f(s)$ for $\operatorname{Re}(s) > \sigma$. If $f(s)$ is analytic on the larger set $\operatorname{Re}(s) \ge \sigma$, then F also converges to $f(s)$ for all $\operatorname{Re}(s) \ge \sigma$.*

One can w. l. o. g. assume $\sigma = 1$. Newman then proves the theorem by applying the Residue Theorem twice, once to a circle around 0 with a vertical cut-off line to the left of the origin, close to the abscissa of convergence (see Figure 2) and once to a full circle around the origin.

My formal proof follows Newman's argument very closely, but like Harrison, I use a modified version of Newman's contour: a semicircle plus a rectangle (see Figure 2). The value of the integral is the same in both cases since the two contours are homotopic, but the bounding of the contributions of the various parts of the contour is different.

The reason why I picked Harrison's contour over Newman's is that I could not understand how Newman's bounding of the different contributions fits to his contour, and it seems likely that this is also the reason why Harrison altered the contour in the first place. Additionally, the shape of the inside of Harrison's contour is somewhat easier to describe.

The formal proof is quite short (roughly 500 lines) and was – apart from the issue I just mentioned – very straightforward to write. However, it again suffers from the aforementioned typical problems of complex analysis in Isabelle, namely having to prove many side conditions such as the geometry of the integration contours. The winding numbers, on the other hand, are unproblematic this time since the contours are very simple.

6.2 An Overview of the Remainder of Newman's Proof

Recall that our main objective was to prove

$$\mathfrak{M}(x) \sim \ln x + c + o(1)\ . \tag{5}$$

The starting point is Mertens' First Theorem, which I prove following e. g. Hildebrand [21]:

▶ **Theorem 41** (Mertens' First Theorem)**.** $\mathfrak{M}(x) = \ln x + O(1)$

To then show (5) from this, Newman defines the Dirichlet series $f(s) := \sum_{n=1}^{\infty} \mathfrak{M}(n) n^{-s}$. Since $\mathfrak{M}(n) - \ln n$ is bounded, $f(s)$ converges absolutely for $\mathrm{Re}(s) > 1$. Rearrangement yields

$$f(s) = \sum_p \frac{\ln p}{p} \zeta(s, p) \qquad \text{for } \mathrm{Re}(s) > 1$$

and further rearrangements show

$$f(s) = \frac{A(s) - \zeta'(s)/\zeta(s)}{s-1} \qquad \text{for } \mathrm{Re}(s) > 1$$

for some function $A(s)$ that is analytic for $\mathrm{Re}(s) > \frac{1}{2}$. Moreover, $\zeta'(s)/\zeta(s)$ is analytic for $\mathrm{Re}(s) \geq 1$, $s \neq 1$ due to the non-vanishing of $\zeta(s)$ in that domain (cf. Theorem 30).

Putting everything together, we obtain that $f(s)$ can be continued analytically to $\mathrm{Re}(s) \geq 1$ except for a double pole at $s = 1$. As Newman states, this double pole can be turned into a simple pole by adding $\zeta'(s)$, and that simple pole can then be eliminated by subtracting a suitable multiple of $\zeta(s)$, yielding a function $g(s) := f(s) + \zeta'(s) - c\,\zeta(s)$ that is analytic for $\mathrm{Re}(s) \geq 1$ and has the Dirichlet series

$$g(s) = \sum_{n=1}^{\infty} \underbrace{(\mathfrak{M}(n) - \ln n - c)}_{=:\, a_n} n^{-s}\ .$$

Applying Theorem 40, we deduce that this series converges for $\mathrm{Re}(s) \geq 1$. For $s = 1$, this means that $\sum_{n=1}^{\infty} \frac{a_n}{n}$ is summable. Next, Newman proves the following lemma:

▶ **Lemma 42.** *Let $a_n : \mathbb{N} \to \mathbb{R}$ be non-decreasing and $\sum_{n=0}^{\infty} \frac{a_n}{n}$ be summable. Then $a_n \longrightarrow 0$.*

Applied to our a_n from before, we get $\mathfrak{M}(n) - \ln n \longrightarrow c$. From this, the slightly stronger version on real numbers (5) follows easily by noting that $\ln x - \ln \lfloor x \rfloor \longrightarrow 0$.

There were no major difficulties in formalising any of this. However, some parts deserve a few comments:

- The rearrangements leading to the analytic continuation of $f(s)$ involve changing the order of summation in nested infinite sums. To do this, I used Isabelle's library for absolutely summable families. This makes the arguments nice to formalise, but the library has the problem of having a function for the *value* of an infinite sum and for its *existence*. Any rearrangement of sums therefore has to be done twice, once for the value of the sum and once for its summability. Similar problems occur in Isabelle with nested integrals and it is not clear if and how this can be avoided in a HOL-based theorem prover.
- Showing that $A(s)$ is indeed analytic for $\mathrm{Re}(s) > \frac{1}{2}$ was a surprisingly easy application of the *Weierstraß M test* with the bounding series $M_n := \ln n(Cn^{-x-1} + n^{-x}(n^x - 1)^{-1})$. The proof obligation that M_n be summable can be solved by showing $M_n \in O(n^{-1-\varepsilon})$ with a suitable $\varepsilon > 0$, and this can be shown by Isabelle's automation for real limits [17].
- The pole cancellation argument showing that $g(s)$ is analytic is about 86 lines long, which is not too long, but still longer than one might expect given that it is obvious considering the Laurent series expansions of the functions involved. This is due to the fact that there is currently no theory of Laurent series expansions in Isabelle yet. In the future, this entire argument could potentially be automated by computing Laurent series expansions for meromorphic functions similarly to how Isabelle's automation already computes Multiseries expansions [17] for real-valued functions.
- The proof of Lemma 42 is very technical and tedious, but it seems to me that this is the case in Newman's paper presentation as well.

The last remaining step, showing that $\mathfrak{M}(x) - \ln x \longrightarrow c$ implies $\vartheta(x) \sim x$, is left as an exercise to the reader by Newman. Harrison was not quite sure what Newman meant [20] and proceeded to prove a number of very technical and ad-hoc lemmas that I find very difficult to follow. Therefore, instead of attempting to port Harrison's proof, I followed Newman's hint in the book and used Abel's summation formula to write $\vartheta(x)$ in terms of $\mathfrak{M}(x)$:

$$\vartheta(x) = x\mathfrak{M}(x) - \int_2^x \mathfrak{M}(t)\,\mathrm{d}t \tag{6}$$

Substituting (5) into (6) yields, in a straightforward way,

$$\begin{aligned}\vartheta(x) &= x\ln x + cx + o(x) - \int_2^x \ln t + c + o(1)\,\mathrm{d}t\\ &= x\ln x + cx + o(x) - (x\ln x - x + cx + o(x)) = x + o(x)\end{aligned}$$

and thus the desired $\vartheta(x) \sim x$. I find it likely that this is what Newman had in mind. ◀

7 Various Other Interesting Results

In this last section, I will give a few examples of other interesting number-theoretic results that I have formalised. The proofs were all fairly straightforward and there is not much to be said about them, but they are worth mentioning nonetheless.

▶ **Theorem 43** (Dirichlet's Theorem). *Let $m > 0$ and $gcd(k, m) = 1$. Then there are infinitely many primes congruent k modulo m.*

▶ **Theorem 44** (Elementary bounds for $\pi(x)$ and p_n)**.** *For any $x \geq 2$ and $n > 0$, we have:*

$$\frac{1}{6}\frac{x}{\ln x} < \pi(x) < 3(e^{-1} + \ln 2)\frac{x}{\ln x} \qquad \text{and} \qquad \frac{139}{443} n \ln n \leq p_{n-1} < 12(n \ln n + n \ln(12/e))$$

In particular, this implies $\pi(x) \in \Theta(x/\ln x)$ and $p_n \in \Theta(n \ln n)$. All of this can be derived without the PNT (hence "elementary" results).

▶ **Theorem 45** (Mertens' three theorems)**.**

- $-1 - 9\pi^{-2} < \mathfrak{M}(n) - \ln n \leq \ln 4$ *for all $n > 0$ and thus* $|\mathfrak{M}(n) - \ln n| < 2$.
- $|(\sum_{p \leq x} 1/p) - \ln\ln x - M| \leq 4/\ln x$ *for all $x \geq 2$ and thus* $\sum_{p \leq x} 1/p = \ln\ln x + M + O(1/\ln x)$ *where M is the Meissel–Mertens constant.*
- $\prod_{p \leq x}(1 - 1/p) = C/\ln x + O(\ln^{-2} x)$ *for some constant $C > 0$.*

Typically, number-theoretic functions that talk about a *single* integer such as $\varphi(n)$ and $\sigma_0(n)$ oscillate heavily and therefore have no nice asymptotics like $\pi(x) \sim x \ln x$. However, their *averages* (i. e. $\sum_{n \leq x} \varphi(n)$) are often more well-behaved:

▶ **Theorem 46** (Averages of arithmetical functions)**.**

- *Let $S(x)$ denote the number of square-free integers $\leq x$. Then $S(x) = \frac{6}{\pi^2}x + O(\sqrt{x})$, i. e. $6/\pi^2 \approx 60.8\,\%$ of integers are square-free.*
- *Euler's totient function φ fulfils $\sum_{n \leq x} \varphi(n) = \frac{3}{\pi^2}x^2 + O(x \ln x)$, i. e. on average, an integer n has $\frac{3}{\pi^2}n$ numbers $\leq n$ that are coprime to it ($\approx 30.4\,\%$).*
- *The divisor function σ_0 fulfils $\sum_{n \leq x} \sigma_0(n) = x \ln x + (2\gamma - 1)x + O(\sqrt{x})$ where $\gamma \approx 0.5772$ is the Euler–Mascheroni constant, i. e. on average, an integer n has $\ln n + 2\gamma - 1$ divisors.*
- $\sum_{n \leq x} \sigma_\alpha(n) = \frac{\zeta(\alpha+1)}{\alpha+1} x^{\alpha+1} + O(R(x))$ *for $\alpha > 0$ where $R(x) = x \ln x$ if $\alpha = 1$ and $R(x) = x^{\max(1,\alpha)}$ otherwise.*
- $\sum_{n \leq x} \sigma_{-\alpha}(n) = \zeta(\alpha + 1)\, x + O(R(x))$ *for $\alpha > 0$ where $R(x) = \ln x$ if $\alpha = 1$ and $R(x) = x^{\max(0,1-\alpha)}$ otherwise.*

Lastly, the following are interesting consequences that follow relatively easily from the PNT:

▶ **Corollary 47.**

- *For each $c > 1$, there exists an x_0 s. t. all intervals $(x, cx]$ with $x \geq x_0$ contain a prime.*
- *The fractions of the form p/q for prime p, q are dense in $\mathbb{R}_{>0}$.*
- $\operatorname{lcm}(1, \ldots, n) = \exp(x + o(x))$
- $\limsup_{n\to\infty} \omega(n) \ln\ln n / \ln n = 1$
- $\limsup_{n\to\infty} \ln \sigma_0(n) \ln\ln n / \ln n = \ln 2$
- $\liminf_{n\to\infty} \varphi(n) \ln\ln n / n = C$ *for some $C \in \mathbb{R}_{>0}$*

The last three statements perhaps deserve some more explanation: They give asymptotic bounds for $\omega(n)$, $\sigma_0(n)$, and $\varphi(n)$. For instance, $\omega(n) < c \ln n / \ln\ln n$ for all sufficiently large n if $c > 1$, but $\omega(n) > c \ln n / \ln\ln n$ for infinitely many n if $c < 1$. Thus, $\ln n / \ln\ln n$ is the best possible upper bound of that shape for $\omega(n)$ (and analogously for the other two).

As for the other direction, recall that $\omega(p) = 1$, $\sigma_0(p) = 2$, and $\varphi(p) = p - 1$. Therefore, the above results show that $\omega(n)$ oscillates between 1 and $\ln n / \ln\ln n$, $\sigma_0(n)$ oscillates between 2 and $2^{\ln n / \ln\ln n}$, and $\varphi(n)$ oscillates between $Cn/\ln\ln n$ and $n - 1$.

8 Size of the Formalisation

The formalised material is spread over five AFP entries [13, 12, 15, 18, 16]. They have a combined size of roughly 25,000 lines of Isabelle code, with the two largest single files by far being those on the analytic properties of Dirichlet series and the properties of the ζ functions.

With the exception of a few minor results, the work presented here was done in 1.5 years by one person – however, the work was not done continuously, but sporadically whenever I found time for it. The total amount of time that went into it is therefore difficult to measure. As a point of reference, the formalisation of Newman's proof of the Prime Number Theorem (with all the components such as Dirichlet series and the ζ function already in place) comprises 3300 lines and took 6 days of full-time work. However, I used two small lemmas that had previously been ported from Harrison's HOL Light formalisation by Paulson. Considering this, a time frame of 7 days for proving the Prime Number Theorem seems reasonable. Based on this, a total effort of 4–6 person-months for the entire work seems realistic.

The formalisation proceeded smoothly and without major difficulties, although some aspects of it stand out as considerably more painful than one might hope:

1. applying the Residue Theorem
2. geometric properties of integration contours
3. manipulating nested infinite sums
4. establishing homotopy of concrete composite paths
5. reasoning about cancellation of poles

For the first three items, it is not clear to me if and how this can be improved – or if, perhaps, there is simply an inherent difficulty in doing such things formally.

Item 4 could probably be addressed by providing more library results about homotopy.

Item 5 could be easily managed by building a tactic to automatically compute Laurent series expansions for meromorphic functions, similar to the existing one for Multiseries expansions of real functions [17]. This would be an interesting project for the future. Extending the limit automation to use not just full asympotic expansions but also partial asymptotic information (such as $\vartheta(x) \sim x$) would also occasionally eliminate some tedious manual work.

A related issue is that reasoning with asymptotic expansions like $f(x) = x^2 + \ln x + O(1/x)$ can be tedious in Isabelle/HOL. They *can* be written as $f =o\ (\lambda x.\ x\hat{}2 + \ln x)\ +o\ O(\lambda x.\ 1/x)$, but there is currently little support for working with them. Affeldt *et al.* [1] demonstrated an approach for this in Coq that could possibly be adapted to Isabelle/HOL.

9 Conclusion

I formalised a large portion of a mathematical textbook on an advanced topic, namely Analytic Number Theory. While some results from this field have been formalised before (such as Dirichlet's Theorem and the Prime Number Theorem), they typically tried to obtain a short route to the result without building an actual library of Analytic Number Theory.

In my opinion, this work demonstrates the following:

- Formalising an entire mathematical textbook in a modern theorem prover *can* be feasible with a moderate amount of effort.
- Good and extensive libraries (e. g. on complex analysis and Dirichlet series) can yield short, clear, and high-level proofs of "high-profile" results like the Prime Number Theorem.
- Specialised tools (e. g. for proving limits or computing winding numbers) are invaluable, as they can take care of tedious and uninteresting parts of the proofs and "close the gap" between what is obvious to a human mathematician and what is easy to do in the system.

There is already work in progress on formalising the remaining parts of Apostol's book. After that, a natural continuation would be to focus on the second volume of Apostol's book, which is called *Modular Functions and Dirichlet Series in Number Theory* [3]. This would be another big step in formalising the essential tools of modern number theory in a theorem prover.

References

1 Reynald Affeldt, Cyril Cohen, and Damien Rouhling. Formalization Techniques for Asymptotic Reasoning in Classical Analysis. *J. Formalized Reasoning*, 11(1):43–76, 2018. doi:10.6092/issn.1972-5787/8124.

2 Tom M. Apostol. *Introduction to analytic number theory*. Undergraduate Texts in Mathematics. Springer-Verlag, 1976. doi:10.1007/978-1-4757-5579-4.

3 Tom M. Apostol. *Modular Functions and Dirichlet Series in Number Theory*, volume 41 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1990. doi:10.1007/978-1-4612-0999-7.

4 Andrea Asperti and Wilmer Ricciotti. A proof of Bertrand's postulate. *Journal of Formalized Reasoning*, 5(1):37–57, 2012. doi:10.6092/issn.1972-5787/3406.

5 Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff. A Formally Verified Proof of the Prime Number Theorem. *ACM Trans. Comput. Logic*, 9(1), December 2007. doi:10.1145/1297658.1297660.

6 Raymond Ayoub. Euler and the zeta function. *The American Mathematical Monthly*, 81(10):1067–1086, 1974. doi:10.2307/2319041.

7 Joseph Bak and Donald J. Newman. *Complex Analysis*. Undergraduate Texts in Mathematics. Springer New York, 1999.

8 Clemens Ballarin. Locales: A Module System for Mathematical Theories. *Journal of Automated Reasoning*, 52(2):123–153, 2014. doi:10.1007/s10817-013-9284-7.

9 Julian Biendarra and Manuel Eberl. Bertrand's postulate. *Archive of Formal Proofs*, January 2017. , Formal proof development. URL: http://isa-afp.org/entries/Bertrands_Postulate.html.

10 Mario Carneiro. Arithmetic in Metamath, Case Study: Bertrand's Postulate. *CoRR*, abs/1503.02349, 2015. arXiv:1503.02349.

11 Mario Carneiro. Formalization of the prime number theorem and Dirichlet's theorem. In *Proceedings of the 9th Conference on Intelligent Computer Mathematics (CICM 2016)*, pages 10–13, 2016. URL: http://ceur-ws.org/Vol-1785/F3.pdf.

12 Manuel Eberl. Dirichlet L-functions and Dirichlet's theorem. *Archive of Formal Proofs*, December 2017. , Formal proof development. URL: http://isa-afp.org/entries/Dirichlet_L.html.

13 Manuel Eberl. Dirichlet series. *Archive of Formal Proofs*, October 2017. , Formal proof development. URL: http://isa-afp.org/entries/Dirichlet_Series.html.

14 Manuel Eberl. The Euler–MacLaurin Formula. *Archive of Formal Proofs*, March 2017. , Formal proof development. URL: http://isa-afp.org/entries/Euler_MacLaurin.html.

15 Manuel Eberl. The Hurwitz and Riemann ζ Functions. *Archive of Formal Proofs*, October 2017. , Formal proof development. URL: http://isa-afp.org/entries/Zeta_Function.html.

16 Manuel Eberl. Elementary Facts About the Distribution of Primes. *Archive of Formal Proofs*, February 2019. , Formal proof development. URL: http://isa-afp.org/entries/Prime_Distribution_Elementary.html.

17 Manuel Eberl. Verified Real Asymptotics in Isabelle/HOL. Draft available at https://www21.in.tum.de/~eberlm/real_asymp.pdf, 2019.

18 Manuel Eberl and Lawrence C. Paulson. The Prime Number Theorem. *Archive of Formal Proofs*, September 2018. , Formal proof development. URL: http://isa-afp.org/entries/Prime_Number_Theorem.html.

19 John Harrison. A formalized proof of Dirichlet's theorem on primes in arithmetic progression. *Journal of Formalized Reasoning*, 2(1):63–83, 2009. doi:10.6092/issn.1972-5787/1558.

20 John Harrison. Formalizing an analytic proof of the Prime Number Theorem (Dedicated to Mike Gordon on the occasion of his 60th birthday). *Journal of Automated Reasoning*, 43(3):243–261, October 2009. doi:10.1007/s10817-009-9145-6.

21 A. J. Hildebrand. Introduction to analytic number theory (lecture notes). https://faculty.math.illinois.edu/~hildebr/ant/.

22 Johannes Hölzl, Fabian Immler, and Brian Huffman. Type Classes and Filters for Mathematical Analysis in Isabelle/HOL. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, volume 7998 of *Lecture Notes in Computer Science*, pages 279–294. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-39634-2_21.

23 Wenda Li and Lawrence C. Paulson. Evaluating Winding Numbers and Counting Complex Roots through Cauchy Indices in Isabelle/HOL. *CoRR*, abs/1804.03922, 2018. arXiv:1804.03922.

24 M. Ram Murty and Marilyn Reece. A simple derivation of $\zeta(1-K) = -B_K/K$. *Funct. Approx. Comment. Math.*, 28:141–154, 2000. doi:10.7169/facm/1538186691.

25 Donald J. Newman. *Analytic number theory*. Number 177 in Graduate Texts in Mathematics. Springer, 1998. doi:10.1007/b98872.

26 Marco Riccardi. Pocklington's theorem and Bertrand's postulate. *Formalized Mathematics*, 14:47–52, January 2006. doi:10.2478/v10037-006-0007-y.

27 Laurent Théry. Proving Pearl: Knuth's Algorithm for Prime Numbers. In David Basin and Burkhart Wolff, editors, *Theorem Proving in Higher Order Logics*, pages 304–318, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. doi:10.1007/10930755_20.

A Certifying Extraction with Time Bounds from Coq to Call-By-Value λ-Calculus

Yannick Forster
Saarland University, Saarland Informatics Campus (SIC), Saarbrücken, Germany
forster@ps.uni-saarland.de

Fabian Kunze
Saarland University, Saarland Informatics Campus (SIC), Saarbrücken, Germany
kunze@ps.uni-saarland.de

Abstract

We provide a plugin extracting Coq functions of simple polymorphic types to the (untyped) call-by-value λ-calculus L. The plugin is implemented in the MetaCoq framework and entirely written in Coq. We provide Ltac tactics to automatically verify the extracted terms w.r.t a logical relation connecting Coq functions with correct extractions and time bounds, essentially performing a certifying translation and running time validation. We provide three case studies: A universal L-term obtained as extraction from the Coq definition of a step-indexed self-interpreter for L, a many-reduction from solvability of Diophantine equations to the halting problem of L, and a polynomial-time simulation of Turing machines in L.

2012 ACM Subject Classification Theory of computation → Type theory; Mathematics of computing → Lambda calculus

Keywords and phrases call-by-value, λ-calculus, Coq, constructive type theory, extraction, computability

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.17

Supplement Material The Coq development is accessible at
`https://github.com/uds-psl/certifying-extraction-with-time-bounds`.

1 Introduction

Every function definable in constructive type theory is computable in a model of computation. This also enables many proof assistants based on constructive type theory to implement extraction into a "real" programming language. On the more foundational side, various realisability models for fragments of constructive type theory increase the trust in this meta-theorem, because realisers for types are the codes of computable functions.

The computability of all definable functions also enables the study of synthetic computability theory in constructive type theory [7, 4]. For instance, one can define decidability by `dec P := ∃ f, ∀ x, P x ↔ f x = true` and no reference to a concrete model of computation is needed. The undecidability of a predicate p can be shown by defining a many-one reduction from the halting problem of Turing machines to p in Coq, again without referring to a concrete model. The computability of all definable functions can, however, not be proved inside the type theory itself, similar to other true statements like parametricity. At the same time, for every concrete defined function of the type theory, one can always prove computability as theorem in the type theory. Given for instance any concrete function $f : \mathbb{N} \to \mathbb{N}$ definable in constructive type theory, one can construct a term of the λ-calculus t_f s.t. for all $n : \mathbb{N}$, there is a proof in the type theory that $t_f\ \overline{n}$ reduces to $\overline{fn}$ (where $\overline{\cdot}$ is a suitable encoding of natural numbers). The construction of t_f from f is relatively simple, since it is syntax-directed and the terms of type theory are just (possibly type-decorated) terms of an expressive untyped λ-calculus. Another way to see this construction is as extraction from the type theory into the λ-calculus.

We implement one such construction of λ-terms t_f for a certain subset of type theory: We use the MetaCoq framework [2] to extend the proof assistant Coq with a command to extract Coq functions of simple polymorphic types into the weak call-by-value λ-calculus L and provide tactics to automatically prove the correctness of the term. In addition to the correctness, our extraction command can generate recurrence equations that, if instantiated with a function by the user, describe the time complexity as number of β-steps of the extracted λ-term on its arguments. Our target calculus L has been used before to formalise computability theory in Coq [12]. Since it is (syntactically) the pure λ-calculus, recursive functions have to be encoded using a fixed-point combinator and inductive types using Scott's encoding.

Our extraction has several use cases:

First, while parts of computability theory can be formalised in Coq without referring to a model of computation [7], one needs a deep embedding of computable functions to e.g. construct universal machines. Our framework then allows the user to write all functions in Coq and automatically get λ-terms computing them, similar to practice on paper where function in the model are never spelled out. For instance, the automated construction of a universal λ-term takes about 30 lines and no manual proofs, whereas by hand construction and verification take about 500 lines [12].

Second, to the best of our knowledge, there are no formalisations of computational complexity theory in any proof assistant. We hope that our framework can be used to enable formalisations of basic complexity theory. One tedium – even on paper – when doing complexity theory in a way such that all details are spelled out is that constructing and verifying functions in the chosen model of computation is hard. With our framework, this burden is significantly lowered: Implementations can be given in Coq and only a suitable running-time function has to be given by hand. We extract a definition of Turing machines to show that L can simulate k steps of a Turing machine in a number of β-steps linear in k.

Third, synthetic undecidability and the notion of synthetic decidability and enumerability have been analysed in Coq [6, 7, 11, 20]. This resulted in a library of undecidable problems in Coq [10]. All problems of the library are shown undecidable by reduction from the halting problem of Turing machines. To show that all contained problems are actually interreducible with the halting problem, one has to give many-one reductions from the problems to the halting problem. Using extraction, a reduction to the halting problem for L is straightforward: It suffices to prove enumerability in Coq, which follows a clear scheme, and then extract the Coq enumerator automatically to L. We demonstrate the power of this method by reducing solvability of Diophantine equations to the halting problem of L.

Lastly, it might be beneficial to use classical axioms like choice when verifying reductions. Since the computability of all definable functions does not necessarily hold given classical assumptions, one can extract the used reductions to L to ensure their computability.

Related Work. Myreen and Owens [25] implement a proof-producing translation from the higher-order logic implemented in the HOL4 system with a state-and-exception monad into CakeML [17]. The translation also produces proofs for the translated terms, similar to our approach. Hupel and Nipkow [14] give a verified compiler from a deep embedding of Isabelle/HOL to CakeML. Similar to our work, they use a logical relation to connect Isabelle definitions to an intermediate representation.

Mullen et al. [24] provide a verified compiler from a subset of Coq to assembly. Anand et al. [1] report on ongoing work on verifying the full extraction process of Coq, also based on the MetaCoq framework. They extract Coq functions into Clight, an intermediate language of

the CompCert compiler, and are thus able to obtain verified assembly code for Coq functions. Letouzey [21] describes the theoretical foundations of extraction in Coq. Our logical relation can be seen as a light-weight version of his simulation predicate for simple polymorphic types.

Köpp [16] verifies program extraction for functions in the Minlog proof assistant into a λ-calculus-like system.

Guéneau et al. [13] verify the asymptotic complexity of functional programs in Coq, based on separation logic with time credits.

We have reported on a preliminary version of our extraction plugin in [8].

2 The call-by-value λ-calculus L

We use the weak call-by-value λ-calculus L defined in [12] and based on [26, 19] as target language. It comes with an inductive type of terms

$$s, t, u, v \ : \ \mathbf{T} \ ::= \ n \mid st \mid \lambda s \qquad (n : \mathbb{N})$$

and a recursive function s_u^k providing a simple, capturing *substitution* operation:

$$\begin{aligned} k_u^k &:= u & n_u^k &:= n \quad (\text{if } n \neq k) \\ (st)_u^k &:= (s_u^k)(t_u^k) & (\lambda s)_u^k &:= \lambda(s_u^{1+k}) \end{aligned}$$

We will freely switch between a named representation for examples and the representation using de Bruijn indices for definitions, i.e. we write $\lambda xy.x$ for $\lambda\lambda 1$.

We define an inductive weak call-by-value *reduction relation* $s \succ t$:

$$\frac{}{(\lambda s)(\lambda t) \succ s_{\lambda t}^0} \qquad \frac{s \succ s'}{st \succ s't} \qquad \frac{t \succ t'}{st \succ st'}$$

We write $\succ^*$ for the reflexive transitive closure of $\succ$, $\succ^k$ for exactly and $\succ^{\leq k}$ for at most k steps.

Note that – contrary to Coq reduction – L-reduction does not apply below binders. Due to the capturing substitution relation, reduction is only well-behaved on closed terms. We call a term *closed* if it has no free variables. Closed abstractions are called *procedures* and are the (only) normal forms of normalising, closed terms.

L provides for recursion using a fixed-point operator:

Lemma 1 (Fact 6 [12]). *There is a function $\rho : \mathbf{T} \to \mathbf{T}$ s.t. $(\rho u)v$ reduces to $u(\rho u)v$ for procedures u, v.*

Inductive datatypes can be encoded using Scott encodings [23, 15], which we explain in Section 4.3.

One crucial property of L reduction is that it is uniformly confluent, making every reduction to a normal form have the same length:

▶ **Theorem 2** (Corollary 8 [12]). *If $s \succ^{k_1} v_1$, $s \succ^{k_2} v_2$ for procedures v_i, then $v_1 = v_2 \wedge k_1 = k_2$.*

For the remainder of this paper, we will write $\mathbb{T}$ for the type of types in Coq, $\mathbb{P}$ for the type of propositions, $\mathbb{L}\,X$ and $\mathbb{O}\,X$ for lists and options over X, and $\mathbb{1}$ (with $\star : \mathbb{1}$) for the unit type.

3 Correctness and time bounds

We define when a term computes a Coq function using two logical relations, one considering just correctness, and one correctness with time bounds. Crucial for both definitions is the notion of an encoding function:

Definition 3. *A function $\varepsilon_A : A \to \mathbf{T}$ is an encoding function for a type A if ε_A is injective and only returns procedures.*

Notice that the only types where such a function can be defined are computationally relevant (i.e. non-propositional), countable types like $\mathbb{B}$, $\mathbb{N}$, $\mathbb{O}\,X$, or $\mathbb{L}\,X$ over countable X.

3.1 Correctness

We define a logical relation $t_a \sim a$, meaning the L-term t_a correctly computes a. We will only define this predicate for elements $a : A$ where A is a simple type of the form $A_1 \to \cdots \to A_n$.

We define the predicate $t_a \sim a$ as follows:

$$\frac{}{\varepsilon_A a \sim a}\ (\text{for } a : A) \qquad \frac{\begin{array}{c} t_f \text{ is a procedure} \wedge \\ \forall a t_a.\ t_a \sim a \to \Sigma v : \mathbf{T}.\ t_f t_a \succ^* v \wedge v \sim fa \end{array}}{t_f \sim f}\ (\text{for } f : A \to B)$$

For elements $a : A$ for encodable types A the only term computing them is their encoding. Functions $f : A \to B$ are computed by a procedure t_f, if for every $a : A$ computed by t_a the term $t_f\ t_a$ computes fa. Note that we could alternatively define the first rule s.t. every term t convertible to the encoding $\varepsilon_A a$ computes a, and then simplify the second rule to read $t_f t_a \sim fa$. While technically correct, this simplification does not work for the extension of the relation with time complexity. We thus stick with the more complicated second rule where we require a term v (using the type theoretical sum Σ[1]) s.t. $t_f t_a$ reduces to v and v computes fa.

Defining this predicate in Coq is not entirely straightforward. As common when defining logical relations, the definition is not strictly positive and thus not accepted by Coq as inductive predicate. The standard approach for non strictly positive predicates is to translate them into a recursive function. However, here we would need recursion over types, which is not supported in Coq's type theory. We circumvent this restrictions by defining a type former $\mathfrak{T} : \mathbb{T} \to \mathbb{T}$ capturing exactly the types we want to recurse on and define the predicate by recursion on $\texttt{ty} : \mathfrak{T} A$:

```
Inductive 𝔗 : Type → Type :=
  𝔗_base A `{registered A} : 𝔗 A (* base types *)
| 𝔗_arr A B (ty₁ : 𝔗 A) (ty₂ : 𝔗 B) : 𝔗 (A → B). (* functions types *)

Fixpoint computes {A} (ty : 𝔗 A) {struct ty}: A → T→ Type :=
  match ty with
    𝔗_base ⇒ fun x ext ⇒ (ext = enc x)
  | 𝔗_arr A B ty₁ ty₂ ⇒ fun f t_f ⇒ proc t_f * (* t_f is closed and normal *)
                                  ∀ (a : A) t_a, computes ty₁ a t_a →
                                  {v : term & (* there exist\[s a term v*)
                                  (t_f t_a >* v)* computes ty₂ (f a) v} end.
```

The first constructor of $\mathfrak{T}$ takes every encodable type as argument, denoted in Coq by the `registered` type class, which we explain in Section 4.4. The second constructor captures exactly non-dependent functions. The definition of `computes` then exactly captures the inductive rules given above[2]. By making $\mathfrak{T}$ a type class, instances `ty` can always be obtained automatically.

[1] For non type-theoriest, Σ can be read as a computable existential quantifier.
[2] Note that `{v : term & P v}` is Coq-notation for a dependent pair.

As a running example, we will use the function $\texttt{map}\ X\ Y : (X \to Y) \to \mathbb{L}\,X \to \mathbb{L}\,Y$ on lists for fixed types X and Y. We assume that X, Y, $\mathbb{L}\,X$ and $\mathbb{L}\,Y$ are all encodable. Then $t \sim \texttt{map}\ X\ Y$ is equivalent to t being a procedure and the proposition $\forall(f : X \to Y)(t_f : \mathbf{T})(L : \mathbb{L}\,X).\ t_f \sim f \to t\ t_f\ (\varepsilon L) \succ^* \varepsilon(\texttt{map}\ X\ Y f\ L)$.

Note that $\sim$ is defined similarly to $[\![\cdot]\!]_2$ on inductives and functions in [21].

3.2 Time bounds

We extend the computability predicate to include time bounds. As time measure for a term we use its number of β-steps to a normal form, which is shown reasonable in [9]. The time bound is expressed depending on the input itself, not its size: e.g. for $f : \mathbb{L}\,\mathbb{N} \to \mathbb{B}$ with $t_f \sim f$, we want to have a time complexity function $\tau_f : \mathbb{L}\,\mathbb{N} \to \mathbb{N}$ such that $\forall L : \mathbb{L}\,\mathbb{N}.\ t_f(\varepsilon L) \succ^{\leq(\tau_f\,L)} \varepsilon(fL)$.

We generalise this idea to also account for higher-order functions and define the type $\mathcal{C}$ of complexity measures τ_a for $a : A$ as follows:

$$\mathcal{C}A := \mathbb{1} \qquad \mathcal{C}(A \to B) := A \to \mathcal{C}A \to \mathbb{N} \times \mathcal{C}B$$

Given the term $\texttt{map}\ X\ Y$ of type $(X \to Y) \to \mathbb{L}\,X \to \mathbb{L}\,Y$ as above, its complexity measure $\tau_{\texttt{map}\ X\ Y}$ will be $(X \to Y) \to (X \to \mathbb{1} \to \mathbb{N} \times \mathbb{1}) \to \mathbb{N} \times (\mathbb{L}\,X \to \mathbb{1} \to \mathbb{N} \times \mathbb{1})$, which is equivalent to $(X \to Y) \to (X \to \mathbb{N}) \to \mathbb{N} \times (\mathbb{L}\,X \to \mathbb{N})$, i.e. it is a function that, given an argument $f : X \to Y$ and a complexity measure $\tau_f : \mathcal{C}(X \to Y)$ (being equivalent to $X \to \mathbb{N}$), returns a pair of the number of steps $\texttt{map}\,f$ needs to (partially) evaluate, and a function that for $L : \mathbb{L}\,X$ computes the remaining number of steps $\texttt{map}\,f\,L$ needs to evaluate.

We can extend the computability predicate with time bounds into a predicate $t_a \sim^{\tau_a} a$:

$$\frac{}{\varepsilon_A a \sim^{\tau} a}\ (\text{for } a : A) \qquad \frac{\begin{array}{c} t_f \text{ is a procedure} \wedge \\ \forall a t_a \tau_a.\ t_a \sim^{\tau_a} a \to \Sigma v : \mathbf{T}. \\ t_f t_a \succ^{\leq n} v \wedge v \sim^{\tau} fa \text{ where } \tau_f a \tau_a = (n, \tau) \end{array}}{t_f \sim^{\tau_a} f}\ (\text{for } f : A \to B)$$

The first rule is essentially unchanged: Since encoded terms $\varepsilon_A a$ are always normal, $\varepsilon_A a \sim^{\tau} a$ holds for every complexity measure τ. For the second rule, we decompose $\tau_f a \tau_a$ into n and τ. The complexity measure $\tau : \mathcal{C}B$ is the complexity measure for $v \sim^{\tau} fa$ and n is the number of steps $t_f\ t_a$ needs to reach v.

Similar to before, we implement the predicate by recursion on an element of $\mathfrak{T}A$:

```
Fixpoint computesTime {A} (ty : 𝔗 A) {struct ty}: A → T→ C A → Type := (* ... *).
```

4 Extraction

We describe the different tools needed to extract functions, constructors and to generate encoding functions.

4.1 Template-Coq

Template-Coq is a quoting library for Coq, now part of the MetaCoq project and originally developed by Malecha [22]. The current state of the project is explained by Anand et al. [2] and Boulier [5].

Template-Coq provides an inductive type `term` implementing the abstract syntax of Coq as an inductive type (Figure 1a). It comes with a monad `TemplateMonad : Type → Prop` (Figure 1b) which allows operations like quoting (i.e. converting Coq terms into their abstract

syntax tree), unquoting (i.e. converting abstract syntax trees into Coq terms), evaluating terms, and making definitions. An operation `m : TemplateMonad A` can be executed using the `Run TemplateProgram m` vernacular command.

As an example, the following function obtains the type of its input by unquoting it into a pair of a type and an element, projecting out the type and returning its quotation:

```
Definition tmTypeOf (s : term) :=
  u ← tmUnquote s ;;
  u'←  tmEval hnf (my_projT1 u) ;;
  t ← tmQuote u';;
  ret t
```

```
Inductive term : Set :=
| tRel      : nat → term
| tLambda   : name → term (* the type *) → term → term
| tLetIn    : name → term (* the term *) → term (* the type *) → term → term
| tApp      : term → list term → term
| tConst    : kername → universe_instance → term
| tConstruct : inductive → nat → universe_instance → term
| tCase     : (inductive * nat) (* num of parameters *) →
              term (* type info *) → term (* discriminee *) →
              list (nat * term) (* branches *) → term
| tFix      : term → nat → term
(* ... *).
```

(a) Term representation.

```
Inductive TemplateMonad : Type → Prop :=
(* Monadic operations *)
| tmReturn : ∀ {A:Type}, A → TemplateMonad A
| tmBind : ∀ {A B : Type}, TemplateMonad A →
             (A → TemplateMonad B) → TemplateMonad B
(* General commands *)
| tmPrint : ∀ {A:Type}, A → TemplateMonad unit
| tmFail : ∀ {A:Type}, string → TemplateMonad A
| tmEval : reductionStrategy → ∀ {A:Type}, A → TemplateMonad A
(* Return the defined constant *)
| tmDefinitionRed : ident → option reductionStrategy → ∀ {A:Type}, A → TemplateMonad A
| tmLemmaRed : ident → option reductionStrategy → ∀ A, TemplateMonad A
(* Quoting and unquoting commands *)
| tmQuote : ∀ {A:Type}, A → TemplateMonad term
| tmUnquote : term → TemplateMonad {T : Type & T}
| tmUnquoteTyped : ∀ A, term → TemplateMonad A
```

(b) Monad operations.

Figure 1 Template-Coq's definitions.

4.2 Extracting Terms

We define a monadic function `extract` which can extract admissible Coq terms into L. In order to extract a Coq term, all the constants appearing in it have to be extracted. To save work, we remember previously generated extracts, similar to Anand et al. [2], who use explicit dictionaries for this task. We employ Coq's type class mechanism instead of dictionaries:

```
Class extracted {A : Type} (a : A) := int_ext : T.
```

This also defines a function `int_ext` which allows referring to the extracted term corresponding to `a` as `int_ext a`, if it exists, and otherwise get an error.

```
Definition map (A B : Type) : (A → B) → list A → list B := fun f ⇒
  fix map := match l with | [] ⇒ @nil B | a :: t ⇒ @cons B (f a) (map l) end
```

Figure 2 Definition of `map : ∀ A B : Type, (A → B)→ list A → list B`.

We restrict the terms we can extract to admissible terms:

▶ **Definition 4.** *A type A is admissible if A is of the form $\forall X_1 \dots X_n : \mathbb{T}.\ B_1 \to \dots \to B_m$ with $B_m \neq \mathbb{T}$. Terms $a : A$ are admissible if A is admissible and if all constants $c : C$ that are proper subterms of a are either*

1. *admissable and occur syntactically on the left hand side of an application fully instantiating the type-parameters of c with constants or*
2. *of type $\mathbb{T}$ and occur syntactically on the right hand side of an application instantiating type parameters.*

This means a type A is admissible if it has no quantification over terms, quantification over types in A is in prenex normal form and the return type of A is not $\mathbb{T}$. The function `map` (Figure 2) for instance is admissible. The only constants appearing in its body are `nil` and `cons`, which are both admissible and occur fully instantiated.

We define an extraction function which correctly extracts admissible terms of a type without type-parameters. If we want to extract polymorphic functions like `map` we use Coq's section mechanism and fix the types `A` and `B` as section variables and extract `map A B`.

The type of the extraction function is

```
extract : (nat → nat)→ term → nat → TemplateMonad T
```

The first argument is an environment argument which tracks lifting information for de Bruijn indices for the treatment of fixed points. The last argument is a fuel argument, needed because recursion on the right-hand constituents of an application is not structurally recursive.

Dealing with variables and binders is relatively straightforward, since Template-Coq already uses a de Bruijn representation of terms. Variables translate directly to variables, functions to λ and fixed points can be translated using ρ from 1. We have to lift variables when entering an abstraction using the standard de Bruijn lifiting operation ($\uparrow$):

```
Notation "↑ E" := (fun n ⇒ match n with 0 ⇒ 0 | S n ⇒ S (E n) end).

Fixpoint extract env s fuel :=
  match fuel with 0 ⇒ tmFail "out of fuel" | S fuel ⇒
  match s with
    Ast.tRel n ⇒ t ← tmEval cbv (var (env n));; ret t
  | Ast.tLambda _ _ s ⇒ t ← extract (↑ env) s fuel ;; ret (lam t)
  | Ast.tFix [BasicAst.mkdef _ nm ty s _] _ ⇒
      t ← extract (fun n ⇒ S (env n)) (Ast.tLambda nm ty s) fuel ;; ret (rho t)
```

In order to extract applications `s R` (where `R` is a list of all arguments), we count the number of type parameters of `s`. If it has none, extraction is straightforward recursion. We extract `s R` by folding over the list `R` as the application of the extraction of all subterms:

```
  | Ast.tApp s R ⇒
    p ← tmDependentArgs s;;
    if p =? 0 then
      t ← extract env s fuel;;
      monad_fold_left (fun t1 s2 ⇒ t2 ← extract env s2 fuel ;; ret (app t1 t2)) R t
```

If `s` has `p > 0` type parameters, we assume that it is the syntax of a previously extracted constant. We split `R` into type parameters `P` and the list of computational arguments `L` and unquote `tApp s P` as `a`. We then obtain an extraction `t` for the constant `a` using the `tmTryInfer` operation invoking type class search. Finally, we again recursively extract by folding over the list of arguments `L`:

```
    else
      let (P, L) := (firstn p R, skipn p R) in
      s' ← tmEval cbv (Ast.tApp s P);;
      (if closedn 0 s'
         then ret tt
         else tmFail "The term contains variables as type parameters.");;
      a ← tmUnquote s' ;;
      a' ← tmEval cbn (my_projT2 a);;
      n ← (tmEval cbv (String.append (name_of s) "_term") ≫=tmFreshName) ;;
      i ← tmTryInfer n (Some cbn) (extracted a') ;;
      let t := (@int_ext _ _ i) in
      monad_fold_left (fun t1 s2 ⇒ t2 ← extract env s2 fuel ;; ret (app t1 t2)) L t
```

For all other syntactic constructs we refer to the Coq code.

We wrap the extraction function into an operation which adds definitions:

```
Definition tmExtract (nm : option string) {A} (a : A) : TemplateMonad T:=
  q ← tmUnfoldTerm a ;;
  t ← extract (fun x ⇒ x) q FUEL ;;
  match nm with
    Some nm ⇒ nm ← tmFreshName nm ;;
                @tmDefinitionRed nm None (extracted a) t ;;
                tmExistingInstance nm;;ret t
  | None ⇒ ret t
  end.
```

4.3 Generation of Scott encodings

We use Scott encodings [23, 15] to encode inductive types and its constructors. Scott encodings represent the matches on the inductive type. For instance, the Scott encoding of the booleans are $\varepsilon_{\mathbb{B}}\texttt{true} = \lambda xy.x$ and $\varepsilon_{\mathbb{B}}\texttt{false} = \lambda xy.y$. For natural numbers, the encodings are $\varepsilon_{\mathbb{N}}0 = \lambda zs.z$ and $\varepsilon_{\mathbb{N}}(Sn) = \lambda zs.s(\varepsilon_{\mathbb{N}}n)$.

As before, we use type classes to remember previously generated encodings:

```
Class encodable (A : Type) := enc_f : A → T.
Class registered (A : Type) := mk_registered
  { enc :> encodable A ;          (* the encoding function for A *)
    proc_enc : ∀ a, proc (enc a); (* encodings are procedures *)
    inj_enc : injective enc       (* encoding is injective *) }.
```

For an inductive type with n constructors, the constructor of index i which takes a arguments has Scott encoding `gen_constructor` $a\ n\ i := \lambda x_1 \dots x_a.\lambda y_1 \dots y_n.y_i x_1 \dots x_a$.

For natural numbers (a type with two constructors, i.e. $n = 2$), the constructor S (which has index $i = 1$ and takes one argument, i.e. $a = 1$) has encoding $\lambda x.\lambda y_1 y_2.y_2 x$ (or $\lambda\lambda\lambda(02)$).

We use `gen_constructor` to define a monadic operation `tmExtractConstr`. If we want to extract `map`, we first extract the two constants occurring in its definition (i.e. `nil` and `cons`) and then the actual function, always fully applied to their type parameters:

```
Section Fix_X_Y.
  Context { X Y : Set }. Context { encY : encodable Y }.

  Run TemplateProgram (tmExtractConstr "nil_term" (@nil X)).
  Run TemplateProgram (tmExtractConstr "cons_term" (@cons X)).
  Run TemplateProgram (tmExtract "map_term" (@map X Y)).
End Fix_X_Y.
```

4.4 Generation of Encoding Functions

We restrict our generation of encoding functions to simple inductive types of the form

```
Inductive T (X1 ... Xp : Type) : Type :=
(* ... *) | constr_i_T : A1 → ... → An → T X1 ... Xp | (* ... *).
```

where `Aj` for $1 \leq \mathtt{j} \leq \mathtt{n}$ is either encodable or exactly `T X1 ... Xn`.

For a fully instantiated inductive type `B = T X1 ... Xp` with `n` constructors we define the encoding function $\varepsilon_{\mathtt{B}}$ as follows:

```
fix f (b : B) := match b with
 | ... | constr_i_T (x1 : A1) ... (xn : An) ⇒ λy1 ... yp.yi (f1 x1 ) ... (fn xn) | ... end
```

where `fj` for $1 \leq \mathtt{j} \leq \mathtt{n}$ is a recursive call `f` if `Aj = B`, or $\varepsilon_{\mathtt{Aj}}$ otherwise. We implement a monadic function `tmEncode` which can be used like this:

```
Section Fix_X.
  Variable (X:Type). Context {intX : registered X}.
  Run TemplateProgram (tmEncode "list_enc" (list X)).
End Fix_X.
```

Note that in principle, more types are Scott-encodable, but we leave the automatic generation for those types to future work.

4.5 Extraction in Coq

To be able to connect extracts t_a to terms a using the predicates $t_a \sim a$ and $t_a \sim^{\tau_a} a$ we define two type classes: The class `computable` is parameterised over a and contains an extracted term $t_a : \mathbf{T}$ and a proof of $t_a \sim a$. The class `computableTime` is in addition parametrised over a time complexity function τ_a:

```
Class computable {A : Type} {ty : 𝔗 A} (a : A) : Type :=
  { ext :> extracted a;
    extCorrect : computes ty a ext }.

Class computableTime {A : Type} (ty : 𝔗 A) (a : A): Type :=
  { extT : extracted a; evalTime : 𝒞 A ;
    extTCorrect : computesTime ty a extT evalTime }.
```

This way, we can write `ext a` or `extT a` for previously extracted terms t_a. Note that since all relevant information can be obtained through the paramaters and the types of the fields, we can leave all instances of this classes opaque in Coq.

5 Automated Verification

We now give an overview over the set of tactics we provide in our framework. All tactics are written in Ltac only, but some of them use the monadic operations explained in the last section. We first explain the tactics to simplify L-terms. We then show how to register inductive datatypes to be used with the framework. Lastly, we explain how to prove the computability relation $t_a \sim a$ and infer recurrence equations for a time bound τ_a.

5.1 Symbolic Simplification for L

All tactics in this section are concerned with proving goals of the form "s is a procedure" or "s reduces to t", or transforming a goal like "s reduces to t" to "s' reduces to t" by simplifying s to s'. While all terms s we simplify will be closed, they might not be concrete terms, e.g. contain the encoding of an arbitrary natural number. The tactics will not unfold definitions.

Lproc: The tactic `Lproc` can prove that a term is closed, an abstraction or a procedure. It syntactically decomposes the term and uses a hint database for easier extensibility.

Lbeta: The tactic `Lbeta` simplifies L-terms by reducing all β-redices of the form $(\lambda s)t$ which are visible without unfolding definitions. It uses `Lproc` to show that t is a procedure and that folded definitions used in s are closed, thus left unchanged by the substitution. `Lbeta` is implemented by reflection, treating names as opaque and using closures to evaluate big terms more efficiently. It can keep track of the number of beta-reductions performed. For example, it simplifies the L-term $(\lambda xy.\, xyy)\, u\, v$ in 2 steps to $u\, v\, v$.

Lrewrite: The tactic `Lrewrite` simplifies terms by the use of a hint database with the same name, containing the correctness statements for previously extracted terms, and by the use of local assumptions, which are important for recursion. For efficiency reasons, it does not use Coq's built-in rewriting and instead traverses terms to find subterms where a hint from the database is applicable. For example, it simplifies the L-term $t_+(t_+(\varepsilon_{\mathbb{N}}\ x)(\varepsilon_{\mathbb{N}}\ 5))(\varepsilon_{\mathbb{N}}\ y)$ to $\varepsilon_{\mathbb{N}}(x+5+y)$. While traversing, `Lrewrite` replaces occurences of t_y with $y : Y$ of registered type by the trivial instance with extraction $\varepsilon_Y y$. This guarantees canonicity of instances of `computable` for registered types.

Additionally, `Lrewrite` simplifies $t_f\, t_x$ to t_{fx} for $x : X$ and $f : X \to Y$. The concrete instance of `computable`(fx) is constructed by combining the instances for f and x.

Lsimpl: The tactic `Lsimpl` repeatedly applies `Lbeta` and `Lrewrite` in alternation and can solve trivial goals by reflexivity.

Time bounds: All tactics can be used to analyse time bounds as well: `Lbeta`, `Lrewrite`, and `Lsimpl` transform goals of the form $s \succ^{?k} t$ to goals of the form $s' \succ^{?k} t$ for an s' with $s \succ^{k1} s'$, instantiating the existential variable `?k` with `k1 + ?k'`.

5.2 Registering Inductive Datatypes

To register an inductive datatype we provide the monadic operation `tmGenEncode : ident → Type → TemplateMonad unit`:

```
Run TemplateProgram (tmGenEncode "nat_enc" nat).
Hint Resolve nat_enc_correct : Lrewrite.
```

The operation generates the encoding function and three obligations, which are discharged automatically[3]. The first and second obligation regard procedureness and injectivity of the generated encoding function by tactics `register_proc` and `register_inj`.

The third obligation is saved as `nat_enc_correct` and is generated similarly to the encoding function. It states that the encoding behaves like Scott encoding and is also proven automatically, using the tactic `extract match`. In the case of natural numbers, it has the following type: `nat_enc_correct : ∀ (n:nat)(s t:term), proc s → proc t → enc n s t` $\succ^{\leq 2}$ `match n with 0 ⇒ s | S n' ⇒ t (enc n') end`. The lemma has to be registered in the hint database `Lrewrite` manually in order to be used by our tactics.

To work with an inductive type, a user also has to extract its constructors. The constant constructors (e.g. 0 for natural numbers) are trivially computable by their encoding:

```
Instance reg_is_ext ty (R : registered ty) (x : ty) : computable x.
Proof. ∃ (enc x). reflexivity. Defined.
```

[3] Using `Global Obligation Tactic` of the `Program` mode shipped with Coq.

A specific instance is only needed for the functional constructors of inductive data types:

```
Instance term_S : computable S. Proof. extract constructor. Qed.
```

The `extract constructor` tactic extracts constructors as described in Section 4.3 and show their correctness fully-automatically as described in the next section.

5.3 Automatically Proving Correctness

As an example[4], we take the boolean disjunction `orb x y := if x then true else y`. For the user, the extraction is fully automatic:

```
Instance term_orb : computable orb. Proof. extract. Qed.
```

The tactic `extract` first extracts the Coq term as described in Section 4.2. In this case, the result is $\lambda xy.x(\texttt{ext true})y$. The verification is then performed by iterating the tactic `cstep`, where in each step a goal is of the form $s \sim \texttt{f}$. The tactic `cstep` performs simplifications depending on the Coq term `f`.

Here, the initial proof goal reads as follows:

$(\lambda xy.x(\texttt{ext true})y) \sim (\texttt{fun x y} \Rightarrow \texttt{if x then true else y})$

In case the Coq term is of function type and not syntactically a `fix`, `cstep` uses the definition of $\sim$ on function types and assumes a boolean `x` computed by a term `ext x`. This yields as intermediate goal the existence of a procedure v with

$(\lambda xy.x(\texttt{ext true})y)(\texttt{ext x}) \succ^* v$ and $v \sim (\texttt{fun y} \Rightarrow \texttt{if x then true else y})$

Now `cstep` uses `Lsimpl` to derive v by simplifying the term $(\lambda xy.x(\texttt{ext true})y)(\texttt{ext x})$ to $\lambda y.(\texttt{ext x})(\texttt{ext true})y$, yielding the proof goal

$\lambda y.(\texttt{ext x})(\texttt{ext true})y \sim (\texttt{fun y} \Rightarrow \texttt{if x then true else y})$

The next call of `cstep` assumes a fixed boolean `y` and simplifies by `Lrewrite`:

$\texttt{if x then ext true else ext y} \sim \texttt{if x then true else y}$

In case the Coq term syntactically has a case distinction on top, `cstep` performs the same case distinction for the proof, here leaving the two goals $\texttt{ext true} \sim \texttt{true}$ and $\texttt{ext y} \sim \texttt{y}$. In both cases the Coq term is of registered type and the next call of `cstep` proves these goals using the definition of $\sim$.

5.3.1 Recursive Functions

Recall that recursive functions in Coq are defined via the `fix` (or `Fixpoint`) construct, which allows the application of recursive calls to "smaller" arguments, where the notion "smaller" is due to the guardedness checker of Coq. The tactic `cstep` proves the correctness using `fix` as well, with the same recursive calls as the extracted function. Therefore, the guardedness checker will accept the proof for exactly the same reasons it accepted the function definition[5].

As an example[6], the extraction of `map A B` (see Figure 2) for registered types `A` and `B` is of shape $\lambda f.\rho\ v_1$ for a procedure v_1, where ρ is the fixed-point combinator from Lemma 1.

[4] Available as an interactive example in `Tactics/ComputableDemo.v` as `Example correctness_example`

[5] The guardedness checker rejects some of our produced proofs when extracting functions not directly structurally recursive: This is due to the additional heuristics in the guardedness checker.

[6] Available as an interactive example in `Tactics/ComputableDemo.v` as `Example correct_recursive`

To verify this term, the proof goal is

$\lambda f.\rho\ v_1 \sim$ `fun f ⇒ fix map l := (...)`

The first call of `cstep` is as in Section 5.3 and yields the following goal, where v_2 is obtained by replacing the `L`-variable f with `ext f` for a fixed computable `f : A → B` in v_1:

$(\rho\ v_2) \sim$ `fix map l := (...)`

In case the Coq term is syntactically a `fix`, `cstep` uses the definition of $\sim$ on function types, but generalises the goal over all arguments of `fix` (in this case only `l`):

$\forall$`l`.$\Sigma v : \mathbf{T}.(\rho\ v_2)$`(ext l)` $\succ^* v \wedge v \sim$ `(fix map l := (...))(ext l)`

`cstep` now inserts a a `fix` into the proof term, obtaining an inductive hypothesis `IH` of the same type as the goal. For the proof term to type-check in the end, `IH` can only be used on arguments structurally smaller than `l`. To guarantee this, `cstep` always performs a case analysis on the recursive argument first, i.e. in this case on `l`, yielding two goals.

In both resulting cases, `cstep` calls `Lrewrite` which uses the inductive hypothesis `IH` to simplify all occurrences of $(\rho\ v_2)$`(ext l')` to `ext ((fix map l := (...))l')`. In both goals, `cstep` needs to obtain a procedure v with $(\rho\ v_2)$`(ext l)`, which is done using `Lsimpl`. For `l` $= []$, the goal is trivial because `ext []` $\sim []$. In the recursive case `l = x :: l'`, `Lsimpl` yields the trivial goal

`ext (f x :: ((fix map l := (...))l')` $\sim$ `f x :: ((fix map l := (...))l')`

5.3.2 Higher-Order Functions

Terms containing higher-order functions applied to arguments need a syntactic transformation to be supported by our framework. To verify the correctness of e.g. `map (fun x ⇒ x + y)l` as part of a bigger program, we essentially need to show

$t_{\texttt{map}}\,(\lambda x.t_+\ x\ y)\,(\varepsilon$`l`$) \sim$ `map (fun x ⇒ x + y)l`

To use the definition of $\sim$ for $t_{\texttt{map}}$, we would have to show $(\lambda x.t_+\ x\ y) \sim$ `(fun x ⇒ x + y)`. This introduces several difficulties, one is that the term might contain free variables that need to be beta abstracted, and another one occurs when time bound are of interest: Since our verification of time bounds is only semi-automatic and requires the user to instantiate the recurrences by hand, we would need to interrupt the proof here for a user to fill in the concrete time bounds for $(\lambda x.t_+\ x\ y)$.

We thus restrict the scope of the framework and only cover applications of higher order functions to arguments which syntactically are composed from previously extracted term by application (without the use of abstractions). In this case this would mean that one has to define a Coq term `f y := fun x ⇒ x + y`, which has to be extracted before `map (f y)l`.

5.4 Proving Time Bounds

All simplification tactics also keep track of the number of β-steps in reductions and can thus be used to infer recurrence equations a correct time complexity function has to satisfy. The only obligation left to the user when proving instances of `computableTime` is to provide a solution to this recurrence equations. As an example, we consider boolean disjunction again and want to find a time complexity function $\tau : \mathbb{B} \to \mathbb{1} \to \mathbb{N} \times (\mathbb{B} \to \mathbb{1} \to \mathbb{N} \times \mathbb{1})$:

```
Instance term_orb : computableTime orb τ.
Proof. extract.
```

This leaves the user with the recurrence equations $\pi_1(\tau x\star) \geq 1$ and $\pi_1(\pi_2(\tau x\star)y\star) \geq 3$, indicating that $t_{\texttt{orb}}$ needs one step to reduce to an abstraction if applied to an encoded boolean x and this abstraction needs 3 further steps to a value if applied to a boolean y. Thus,

choosing τ as `fun _ _ ⇒ (1,fun _ _ ⇒ (3,tt))` works. We provide the tactic `solverec` which simplifies goals containing inequations and tries to show them using the `lia` tactic shipped with Coq. If proving the inequality needs further reasoning, the tactic presents the user with simplified goals.

The recurrence equations for the time bound are inferred incrementally by `cstep` using an existential variable. To prove `computableTime orb` τ, `cstep` first introduces an assumption `H : ?P` τ and opens a new goal `?P` τ. In each step, `cstep` performs the transformations described in Section 5.3 while keeping track of the number of steps, asserting that τ needs to be larger than the number of β-steps performed by instantiating `?P` further. For non-recursive functions, this will only produce lower bounds for components of τ, while for recursive correctness proof it produces inequalities that contain τ on both sides.

To find time bound functions interactively, we define the opaque polymorphic constant `cnst {X:Type} (x:X): nat := 0` which can be used as a place-holder for unknown constants. To find the time complexity for `map`[7] one would start with the following:

```
Lemma termT_map A B (Rx : registered A) (Ry: registered B):
    computableTime (@map A B) (fun f τ_f ⇒ (cnst "c",fun l _ ⇒ (cnst ("g",l),tt))).
Proof. extract. solverec.
```

This yields three conditions: `1 <= cnst "c"`, `7 <= cnst ("g", [])`, and `fst (τ_f a tt) + cnst ("g", l) + 11 <= cnst ("g", a :: l)`. Note that `cnst` allows us to keep track of the different arguments that the time bound is instantiated with later. As expected, the time bound of map must also sum up all the time bounds for calling `f` on all elements of the list, and indeed, `solverec` can show the lemma using this time bound:

```
fun f τ_f ⇒ (1, fun l _ ⇒ (fold_right (fun x res ⇒ π1 (τ_f x tt) + res + 11) 7 l, tt))
```

6 Case studies

We provide three case studies: A universal L-term obtained as extraction from the Coq definition of a step-indexed self-interpreter for L (in `Functions/Universal.v`), a many-one reduction from solvability of Diophantine equations to the halting problem of L (in `Reductions/H10.v`), and a linear simulation of Turing machines in L (in `TM/TMEncoding.v`).

6.1 Step-indexed L-interpreter

A step-indexed interpreter for L is a function `eva` : $\mathbb{N} \to \mathbf{T} \to \mathbb{O}\,\mathbf{T}$ s.t. for closed s we have $(\exists n.\ \texttt{eva}\ n\ s = \lfloor t \rfloor) \leftrightarrow (s \succ^* t \wedge t\ \textit{is a procedure})$. The function can be defined as follows [12]:

```
Fixpoint eva (n : nat) (u : term) :=
  match u with
  | var n ⇒ None | lam s ⇒ Some (lam s)
  | app s t ⇒ match n with
              | 0   ⇒ None
              | S n ⇒ match eva n s, eva n t with
                      | Some (lam s), Some t ⇒ eva n (subst s 0 t)
                      |   _ , _ ⇒ None
  end          end    end.
```

Here `subst s 0 t` denotes substitution, which uses `Nat.eqb` as boolean equality test on natural numbers. We extract all three functions in reverse order. To do so, we first need encodings for natural numbers and term constructors as shown in Section 5.2 and encodings for terms. We first generate the encoding function and register it:

[7] Available as an interactive example in `Tactics/ComputableDemo.v` as `comeUp_timebound`

```
Run TemplateProgram (tmGenEncode "term_enc" term).
Hint Resolve term_enc_correct : Lrewrite.
```

We can then extract the non-constant constructors, `Nat.eqb`, `subst`, and `eva`:

```
Instance term_var : computableTime var (fun n _ ⇒ (1, tt)).
Proof. extract constructor. solverec. Qed.
Instance term_app : computableTime app (fun s1 _ ⇒ (1, (fun s2 _ ⇒ (1, tt)))).
Proof. extract constructor. solverec. Qed.
Instance term_lam : computableTime lam (fun s _ ⇒ (1, tt)).
Proof. extract constructor. solverec. Qed.

Instance termT_nat_eqb :
  computableTime Nat.eqb (fun x _ ⇒ (5, (fun y _ ⇒ ((min x y) * 15 + 8, tt)))).
Proof. extract. solverec. Qed.

Instance term_substT :
  computableTime subst (fun s _ ⇒ (5, (fun n _ ⇒ (1, (fun t _ ⇒
                        (15 * n * size s + 43 * (size s) ^ 2 + 13, tt)))))).
Proof. extract. solverec. Qed.

Instance term_eva : computable eva.
Proof. extract. Qed.
```

Note that the implementation of `eva` is very naive and needs steps exponential in n, we thus omit its time complexity.[8] A more reasonable implementation could be obtained by extracting the heap-based abstract machine from [18] to L.

6.2 Diophantine equations

The problems contained in the library of undecidable problems in Coq [10] are proven undecidable by a chain of many-one reductions starting at the halting problem for Turing machines. As a matter of fact, all problems contained in the library so far are actually interreducible. An easy way to prove this is to reduce leafs in the reduction graph to the halting problem for L defined as $\mathcal{E}s := \exists v.(s \succ^* v \wedge v \textit{ is an abstraction})$ and then implement one general reduction from $\mathcal{E}$ to the halting problem of Turing machines.

As an example how to reduce problems to $\mathcal{E}$ we use our framework to reduce solvable Diophantine equations [20], i.e. Hilbert's tenth problem H10, to $\mathcal{E}$.

We first explain the general structure using mathematical notation. In [7], the authors define synthetic notions of decidability and enumerability. If this definitions are enriched with explicit computability assumptions, one obtains:

▶ **Definition 5.** *A predicate* $p : X \to \mathbb{P}$ *is* L*-decidable if there exists* a computable $f : X \to \mathbb{B}$ *s.t.* $\forall x.\ px \leftrightarrow fx = \mathsf{tt}$.

▶ **Definition 6.** *A predicate* $p : X \to \mathbb{P}$ *is* L*-enumerable if there exists* a computable $f : \mathbb{N} \to \mathbb{O}\,X$ *s.t.* $\forall x.\ px \leftrightarrow \exists n. fn = \lfloor x \rfloor$.

Theorem 7. *If* $p : X \to \mathbb{P}$ *is* L*-enumerable and equality on* X *is* L*-decidable, then* $p \preceq \mathcal{E}$.

Proof. Let f be the (computable) enumerator $\mathbb{N} \to \mathbb{O}\,X$ and $d : X \times X \to \mathbb{B}$ the (computable) equality decider. We define $s := \lambda x.\ \mu(\lambda n.\ t_f n\ (\lambda y. t_d\ x\ y)\ t_{\mathsf{ff}})$. Here, μ is an unbounded search operator, i.e. s performs unbouded search for x in the range of f. Then $px \leftrightarrow \mathcal{E}(s\ \overline{x})$. ◀

[8] The recurrence equation generated for eva one would have to solve reads $f(1+n)(s_1 s_2) \geq f\ n\ s_1 + f\ n\ s_2 + 43 \cdot (\mathtt{size}\ t_1)^2 + f\ n\ (t_1{}^0_{t_2}) + 53$, with `eva` $n\ s_1 = \lambda t_1$ and `eva` $n\ s_2 = t_2$.

Moreover, it is easier to implement concrete enumerators based on lists, i.e. computable enumerators $f : \mathbb{N} \to \mathbb{L}\, X$ s.t. $px \leftrightarrow \exists n.\ x \in fn$. The equivalence proof of both notions can be found in [7]. Extending the proof with explicit computability assumptions as needed here is straightforward and we refer to the Coq code.

We now switch to a more technical notation and show how to construct such a list enumerator for H10 in Coq. We first define the type of polynomials, generate its encoding and extract its constructors:

```
Inductive poly : Set :=
    poly_cst : nat → poly          | poly_var : nat → poly
  | poly_add : poly → poly → poly | poly_mul : poly → poly → poly.

Run TemplateProgram (tmGenEncode "enc_poly" poly).
Hint Resolve enc_poly_correct : Lrewrite.

Instance term_poly_cst : computable poly_cst. extract constructor. Qed.
Instance term_poly_var : computable poly_var. extract constructor. Qed.
Instance term_poly_add : computable poly_add. extract constructor. Qed.
Instance term_poly_mul : computable poly_mul. extract constructor. Qed.
```

We define evaluation of polynomials under assignments `S : list nat` as and the decision problem H10 as follows:

```
Fixpoint eval (p : poly) (S : list nat) :=
  match p with
  | poly_cst n ⇒ n
  | poly_var n ⇒ nth n S 0
  | poly_add p1 p2 ⇒ eval p1 S + eval p2 S
  | poly_mul p1 p2 ⇒ eval p1 S * eval p2 S
  end.
Definition H10 '(p1, p2) := ∃ S, eval p1 S = eval p2 S.
Instance term_eval : computable eval. extract. Qed.
```

where `nth n S d` returns the `n`-th element in `S`, or `d` if `S` is not long enough. We also define a computable function `poly_eqb : poly → poly → bool` deciding syntactic equality.

To show that H10 is L-enumerable, we enumerate all polynomials using `L_poly : nat → list poly`. Due to the restriction that higher-order arguments can not syntactically contain abstractions, we first extract uncurried versions of the constructors:

```
Definition poly_add' '(x,y) : poly := poly_add x y.
Instance term_poly_add' : computable poly_add'. extract. Qed.

Definition poly_mul' '(x,y) : poly := poly_mul x y.
Instance term_poly_mul' : computable poly_mul'. extract. Qed.

Fixpoint L_poly n : list (poly) :=
  match n with
  | 0 ⇒ []
  | S n ⇒ L_poly n ++ map poly_cst (L_nat n) ++ map poly_var (L_nat n)
                   ++ map poly_add' (list_prod (L_poly n) (L_poly n))
                   ++ map poly_mul' (list_prod (L_poly n) (L_poly n))
  end.

Instance term_L_poly : computable L_poly. extract. Qed.
```

The last and crucial lemma is the adaption of Fact 2.9 from [7]:

Lemma 8. *If $p : X \times Y \to \mathbb{P}$ is* L*-enumerable, then $\lambda x.\exists y.\ p(x, y)$ is* L*-enumerable.*

Theorem 9. H10 *is* L*-enumerable.*

Proof. By Lemma 8 we have to give a list enumerator for two polynomials `p1` and `p2` together with solutions `S`:

```
fix f n := match n with 0 ⇒ []
  | S n ⇒ f n ++ filter (fun '(p1,p2,S) ⇒ Nat.eqb (eval p1 S) (eval p2 S))
                  (list_prod (list_prod (L_poly n) (L_poly n)) (L_list_nat n)) end.
```

where `list_prod` is the cartesian product on lists and `L_list_nat` is a list enumerator for `list nat`. ◀

Corollary 10. H10 $\preceq \mathcal{E}$

Proof. By Theorems 9 and 7. ◀

6.3 Turing Machines

We show how our framework can be used to reduce the halting problem of multi-tape Turing machines `Halt` to the halting problem of `L`. We employ a Coq implementation of the definition of Turing machines by Asperti and Ricciotti [3], who formalise Turing machines in Matita.

```
Definition loopM : ∀ (sig : finType) (n : nat) (M : mTM sig n),
   mconfig sig (states M) n → nat → option (mconfig sig (states M) n) := (* ... *)

Definition Halt :{ '(Sigma, n) : _ & mTM Sigma n & tapes Sigma n} → _ :=
  fun '(existT2 _ _ (Sigma, n) M tp) ⇒
    ∃ (f: mconfig _ (states M) _), halt (cstate f) = true
                                 ∧ ∃ k, loopM (mk_mconfig (start M) tp) k = Some f.
```

Their formalisation uses the (dependent) vector type to model multiple tapes and an explicit transition function. Both aspects do not fit in our framework directly. We thus showcase two techniques to extend our framework in certain cases.

First, to encode types not in the scope of the framework, we notice that an encoding for a type A can be obtained from an encoding function ε_B given an injective function $A \to B$. We pack this insight in the definition `registerAs`, which can be used as follows:

```
Instance register_vector X `{registered X} n : registered (Vector.t X n).
Proof. apply (registerAs VectorDef.to_list). (* injectivity proof *) Defined.
```

Second, we observe that computability is closed under extensional equality:

Definition 11. *We define extensional equality for a type A with $\mathtt{ty} : \mathfrak{T}A$ recursively on $\mathtt{ty}$. Elements x, y of an encodable type A are extensionally equal if they are equal. Functions $f, g : A \to B$ are extensionally equal if for all $a : A$, fa is extensionally equal to ga.*

Lemma 12. *If f and g are extensionally equal and $t \sim^\tau f$ then $t \sim^\tau g$.*

Combining those two insights allows us to extract any vector operation by extracting the corresponding list-operation.

Furthermore, we use the fact that functions with finite domain and co-domain can always be translated into a value table containing lists of pairs. We can thus show that every transition function is computable in time independent of the current configuration, and derive time bound for `loopM`, executing a machine for k steps:

```
Instance term_trans : computableTime (trans (m:=M)) (fun _ _ ⇒ (transTime,tt)).
Proof. (* ... *) Qed.

Instance term_loopM :
  let c1 := (haltTime + n*121 + transTime + 76) in let c2 := 13 + haltTime in
  computableTime (loopM (M:=M)) (fun _ _ ⇒ (5,fun k _ ⇒ (c1 * k + c2,tt))).
Proof. unfold loopM. extract. solverec. Qed.
```

Here `haltTime` and `transTime` are constants depending on the concrete machine, its number of tapes and its alphabet. By unbounded search over all number of steps k we obtain:

Theorem 13. `Halt` *reduces to* $\mathcal{E}$.

7 Conclusion

Formalisation. The tools in our framework heavily rely on Coq's tactic language Ltac to verify the correctness of extracted terms. During the verification, existential variables are crucial to generate the recurrence equations described in Section 5.4 while simultaneously simplifying the L-terms as described in Section 5.1. For this simplification, we implement a reflective simplification tactic for L-terms used in `Lbeta`. We tried to use setoid-rewriting for `Lrewrite`, but the need to track the number of reduction steps requires us to implement our own, domain-specific rewriting tactic in Ltac. This tactic implements bottom-up rewriting, resulting in smaller proof terms and faster rewriting, by performing many rewrite steps in one pass through the term: A tactic using congruence lemmas descends in the term and on the way out, rewriting steps are performed. We use the hint databases for the `auto`-tactic to add new lemmas for rewriting.

Typeclasses are employed as a kind of dictionary, e.g. to look up the extraction for a previously extracted function or its correctness lemma.

The framework consists of roughly 2100 lines of code, of which 370 are for the definitions described in Section 3.2 and their properties, 380 are for the extraction in Section 4, 950 are for the simplification presented in Section 5.1, and 420 are for the tactics proving those extracts correct in Section 5.3.

In total, the case studies consist of 340 lines of specification and 280 lines f code: 20 lines are for the universal machine, 200 for H10 and 400 for the Turing machine interpreter. All examples are built on a library of extracted functions concerning natural numbers, booleans and lists, which consists of 360 lines of code.

Future Work. There are several directions in which the framework can be extended. We would like to extend the framework to support space bounds in addition to time bounds, based on the space measure defined in [9]. Furthermore, our automation framework is sound by construction, because it produces proofs. We conjecture it to be complete for the described fragment of Coq's type theory we are considering, but reasoning about tactics programmed in Ltac is basically impossible. In the future, we would like to be able to support all of Coq's type theory (possibly leaving out co-inductive types). In order to do that, the extraction process would have to support proof and type erasure, which can be implemented using Template-Coq.

On the more conceptual side, our extraction basically returns realisers in a realisability model for the treated fragment of Coq's type theory. We would like to analyse and verify such realisability models using MetaCoq, possibly connecting the (weak call-by-value) evaluation relation defined in MetaCoq with reduction in the realisability model, yielding a proof that for a certain subset of Coq's type theory, all definable functions are indeed computable.

Lastly, we hope that our framework enables the formalisation of basic computational complexity theory in Coq. We would like to mechanise results like a time hierarchy theorem for the call-by-value λ-calculus. The commonly known proofs for Turing machines or similar models use self-interpreters. The tightness of the provable gap then depends on the time-efficiency of the interpreter in use. As mentioned, the self-interpreter given in Section 6.1 is too inefficient and we want to extract the interpreters described in [18] and [9] to L.

References

1 Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. CertiCoq: A verified compiler for Coq. In *The Third International Workshop on Coq for Programming Languages (CoqPL)*, 2017.

2 Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. Towards Certified Meta-Programming with Typed Template-Coq. In *International Conference on Interactive Theorem Proving*, pages 20–39. Springer, 2018.

3 Andrea Asperti and Wilmer Ricciotti. A formalization of multi-tape Turing machines. *Theoretical Computer Science*, 603:23–42, October 2015. doi:10.1016/j.tcs.2015.07.013.

4 Andrej Bauer. First steps in synthetic computability theory. *Electronic Notes in Theoretical Computer Science*, 155:5–31, 2006.

5 Simon Pierre Boulier. *Extending type theory with syntactic models*. PhD thesis, Ecole nationale supérieure Mines-Télécom Atlantique, 2018.

6 Yannick Forster, Edith Heiter, and Gert Smolka. Verification of PCP-Related Computational Reductions in Coq. *arXiv preprint arXiv:1711.07023*, 2017. Accepted at ITP 2018.

7 Yannick Forster, Dominik Kirst, and Gert Smolka. On synthetic undecidability in Coq, with an application to the Entscheidungsproblem. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 38–51. ACM, 2019.

8 Yannick Forster and Fabian Kunze. Verified Extraction from Coq to a Lambda-Calculus. *Coq Workshop 2016*, 2016.

9 Yannick Forster, Fabian Kunze, and Marc Roth. The weak call-by-value λ-calculus is reasonable for both time and space. *CoRR*, abs/1902.07515, 2019. arXiv:1902.07515.

10 Yannick Forster and Dominique Larchey-Wendling. Towards a library of formalised undecidable problems in Coq: The undecidability of intuitionistic linear logic. In *Workshop on Syntax and Semantics of Low-level Languages, Oxford*, 2018.

11 Yannick Forster and Dominique Larchey-Wendling. Certified undecidability of intuitionistic linear logic via binary stack machines and Minsky machines. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 104–117. ACM, 2019.

12 Yannick Forster and Gert Smolka. Weak call-by-value lambda calculus as a model of computation in Coq. In *ITP 2017*, pages 189–206. Springer, 2017.

13 Armaël Guéneau, Arthur Charguéraud, and François Pottier. A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In *European Symposium on Programming*, pages 533–560. Springer, 2018.

14 Lars Hupel and Tobias Nipkow. A verified compiler from Isabelle/HOL to CakeML. In *European Symposium on Programming*, pages 999–1026. Springer, 2018.

15 Jan Martin Jansen. Programming in the λ-Calculus: From Church to Scott and Back. In *The Beauty of Functional Code*, volume 8106 of *LNCS*, pages 168–180. Springer, 2013.

16 Nils Köpp. Automatically verified program extraction from proofs with applications to constructive analysis. Master's thesis, LMU Munich, 2018. URL: http://www.mathematik.uni-muenchen.de/~schwicht/seminars/semws18/main.pdf.

17 Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In *ACM SIGPLAN Notices*, volume 49, pages 179–191. ACM, 2014.

18 Fabian Kunze, Gert Smolka, and Yannick Forster. Formal Small-step Verification of a Call-by-value Lambda Calculus Machine. In *Asian Symposium on Programming Languages and Systems*, pages 264–283. Springer, 2018.

19 Ugo Dal Lago and Simone Martini. The weak lambda calculus as a reasonable machine. *Theor. Comput. Sci.*, 398(1-3):32–50, 2008. doi:10.1016/j.tcs.2008.01.044.

20 Dominique Larchey-Wendling and Yannick Forster. Hilbert's Tenth Problem in Coq. In Herman Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*, volume 131 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:20, 2019.

21 Pierre Letouzey. *Certified functional programming: program extraction within Coq proof assistant*. PhD thesis, Université Paris-Sud, France, 2004.

22 Gregory Michael Malecha. *Extensible proof engineering in intensional type theory*. Harvard University, 2015.

23 Torben Æ. Mogensen. Efficient Self-Interpretations in lambda Calculus. *J. Funct. Program.*, 2(3):345–363, 1992. doi:10.1017/S0956796800000423.

24 Eric Mullen, Stuart Pernsteiner, James R Wilcox, Zachary Tatlock, and Dan Grossman. Œuf: minimizing the Coq extraction TCB. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 172–185. ACM, 2018.

25 Magnus O Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *Journal of Functional Programming*, 24(2-3):284–315, 2014.

26 Gordon D. Plotkin. Call-by-Name, Call-by-Value and the lambda-Calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975. doi:10.1016/0304-3975(75)90017-1.

Formal Proof and Analysis of an Incremental Cycle Detection Algorithm

Armaël Guéneau
Inria, Paris, France

Jacques-Henri Jourdan
CNRS, LRI, Univ. Paris Sud, Université Paris Saclay, France

Arthur Charguéraud
Inria & Université de Strasbourg, CNRS, ICube, Strasbourg, France

François Pottier
Inria, Paris, France

Abstract

We study a state-of-the-art incremental cycle detection algorithm due to Bender, Fineman, Gilbert, and Tarjan. We propose a simple change that allows the algorithm to be regarded as genuinely online. Then, we exploit Separation Logic with Time Credits to simultaneously verify the correctness and the worst-case amortized asymptotic complexity of the modified algorithm.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Theory of computation → Program verification; Software and its engineering → Correctness; Software and its engineering → Software performance

Keywords and phrases interactive deductive program verification, complexity analysis

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.18

Related Version An extended version of the paper is available at `https://hal.inria.fr/hal-02167236`.

Supplement Material The Coq development is accessible at `https://gitlab.inria.fr/agueneau/incremental-cycles`, and the mechanized metatheory of Separation Logic with Time Credits is available at `https://gitlab.inria.fr/charguer/cfml2`.

1 Introduction

A good algorithm must be correct. Yet, to err is human: algorithm designers and algorithm implementors sometimes make mistakes. Although testing can detect mistakes, it cannot in general prove their absence. Thus, when high reliability is desired, algorithms should ideally be verified. A "verified algorithm" traditionally means an algorithm whose correctness has been verified: it is a package of an implementation, a specification, and a machine-checked proof that the algorithm always produces a result that the specification permits.

A growing number of verified algorithms appear in the literature. To cite just a very few examples, in the area of graph algorithms, Lammich and Neumann [27, 26] verify a generic depth-first search algorithm which, among other applications, can be used to detect a cycle in a directed graph; Lammich [25], Pottier [36], and Chen et al. [10, 9] verify various algorithms for finding the strongly connected components of a directed graph. A verified algorithm can serve as a building block in the construction of larger verified software: for instance, Esparza et al. [12] use a cycle detection algorithm as a component in a verified LTL model-checker.

However, a good algorithm must not just be correct: it must also be fast, and reliably so. Many algorithmic problems admit a simple, inefficient solution. Therefore, the art and science of algorithm design is chiefly concerned with imagining more efficient algorithms, which often

are more involved as well. Due to their increased sophistication, these algorithms are natural candidates for verification. Furthermore, because the very reason for existence of these algorithms is their alleged efficiency, not only their correctness, but also their complexity, should arguably be verified.

Following traditional practice in the algorithms literature [22, 43], we study the complexity of an algorithm based on an abstract cost model, as opposed to physical worst-case execution time. Furthermore, we wish to establish asymptotic complexity bounds, such as $O(n)$, as opposed to concrete bounds, such as $3n + 5$. While bounds on physical execution time are of interest in real-time applications, they are difficult to establish and highly dependent on the compiler, the runtime system, and the hardware. In contrast, an abstract cost model allows reasoning at the level of source code. We fix a specific model in which every function call has unit cost and every other primitive operation has zero cost. Although one could assign a nonzero cost to each primitive operation, that would make no difference in the end: an asymptotic complexity bound is independent of the costs assigned to the primitive operations, and is robust in the face of minor changes in the implementation.

In prior work, Charguéraud and Pottier [8] verify the correctness and the worst-case amortized asymptotic complexity of an OCaml implementation of the Union-Find data structure. They establish concrete bounds, such as $4\alpha(n) + 12$, as opposed to asymptotic bounds, such as $O(\alpha(n))$. This case study demonstrates that it is feasible to mechanize such a challenging complexity analysis, and that this analysis can be carried out based on actual source code, as opposed to pseudo-code or an idealized mathematical model of the data structure. Charguéraud and Pottier use CFML [6, 7], an implementation inside Coq of Separation Logic [37] with Time Credits [3, 8, 17, 18, 32]. This program logic makes it possible to simultaneously verify the correctness and the complexity of an algorithm, and allows the complexity argument to depend on properties whose validity is established as part of the correctness argument. We provide additional background in Section 2.

In subsequent work, Guéneau, Charguéraud and Pottier [17] formalize the O notation, propose a way of advertising asymptotic complexity bounds as part of Separation Logic specifications, and implement support for this approach in CFML. They present a collection of small illustrative examples, but do not carry out a challenging case study.

One major contribution of this paper is to present such a case study. We verify the correctness and worst-case amortized asymptotic complexity of an incremental cycle detection algorithm (and data structure) due to Bender, Fineman, Gilbert, and Tarjan [4, §2]. With this data structure, the complexity of building a directed graph of n vertices and m edges, while incrementally ensuring that no edge insertion creates a cycle, is $O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$. Although its implementation is relatively straightforward, its design is subtle, and it is far from obvious, by inspection of the code, that the advertised complexity bound is respected.

As a second contribution, on the algorithmic side, we simplify and enhance Bender et al.'s algorithm. To handle the insertion of a new edge, the original algorithm depends on a runtime parameter, which limits the extent of a certain backward search. This parameter influences only the algorithm's complexity, not its correctness. Bender et al. show that setting it to $\min(m^{1/2}, n^{2/3})$ throughout the execution of the algorithm allows achieving the advertised complexity. This means that, in order to run the algorithm, one must anticipate the *final* values of m and n. This seems at least awkward, or even impossible, if one wishes to use the algorithm in an online setting, where the sequence of operations is not known in advance. Instead, we propose a modified algorithm, where the extent of the backward search is limited by a value that depends only on the *current* state. The pseudocode for both algorithms appears in Figure 2; it is explained later on (Section 5). The modified algorithm has the same complexity as the original algorithm and is a genuine online algorithm. It is the one that we verify.

As a third contribution, on the methodological side, we switch from $\mathbb{N}$ to $\mathbb{Z}$ in our accounting of execution costs, and explain why this leads to a significant decrease in the number of proof obligations. In our previous work [8, 17], costs are represented as elements of $\mathbb{N}$. In this approach, at each operation of (say) unit cost in the code, one must prove that the number of execution steps performed so far is less than the number of steps advertised in the specification. This proof obligation arises because, in $\mathbb{N}$, the equality $m + (n - m) = n$ holds if and only if $m \leq n$ holds. In contrast, in $\mathbb{Z}$, this equality holds unconditionally. For this reason, representing costs as elements of $\mathbb{Z}$ can dramatically decrease the number of proof obligations (Section 3). Indeed, one must then verify just once, at the end of a function body, that the actual cost is less than or equal to the advertised cost. The switch from $\mathbb{N}$ to $\mathbb{Z}$ requires a modification of the underlying Separation Logic, for which we provide a machine-checked soundness proof.

Our verification effort has had some practical impact already. For instance, the Dune build system [41] needs an incremental cycle detection algorithm in order to reject circular build dependencies as soon as possible. For this purpose, the authors of Dune developed an implementation of Bender et al.'s original algorithm, which we recently replaced with our improved and verified algorithm [16]. Our contribution increases the trustworthiness of Dune's code base, without sacrificing its efficiency: in fact, our measurements suggest that our code can be as much as 7 times faster than the original code in a real-world scenario. As another potential application area, it is worth mentioning that the second author (Jourdan) has deployed an as-yet-unverified incremental cycle detection algorithm in the kernel of the Coq proof assistant [44], where it is used to check the satisfiability of universe constraints [39, §2]. At the time, this yielded a dramatic improvement in the overall performance of Coq's proof checker: the total time required to check the Mathematical Components library dropped from 25 to 18 minutes [23]. The algorithm deployed inside Coq is more general than the verified algorithm considered in this paper, as it also maintains strong components, as in Section 4 of Bender et al.'s paper [4]. Nevertheless, we view the present work as one step towards verifying Coq's universe inference system.

In summary, the main contributions of this paper are:

- A simple yet crucial improvement to Bender et al.'s incremental cycle detection algorithm, making it a genuine online algorithm;
- An implementation of it in OCaml as a self-contained, reusable data structure;
- A machine-checked proof of the functional correctness and worst-case amortized asymptotic complexity of this implementation.
- The discovery of the nonobvious fact that counting time credits in $\mathbb{Z}$ leads to significantly fewer proof obligations, together with a study of the metatheory of Separation Logic with Time Credits in $\mathbb{Z}$ and support for it in CFML.

Our code and proofs are available online (Supplement Material). Our methodology is modular: at the end of the day, the verified data structure is equipped with a succinct specification (Figure 1) which is intended to serve as the sole reference when verifying a client of the algorithm. We believe that this case study illustrates the great power and versatility of our approach, and we claim that this approach is generally applicable to many other nontrivial data structures and algorithms.

2 Separation Logic with Time Credits

Hoare Logic [19] allows verifying the correctness of an imperative algorithm by using *assertions* to describe the state of the program. Separation Logic [37] improves modularity by employing assertions that describe only a fragment of the state and at the same time assert the unique

ownership of this fragment. In general, a Separation Logic assertion claims the ownership of certain *resources*, and (at the same time) describes the current state of these resources. A heap fragment is an example of a resource.

Separation Logic with Time Credits [3, 8, 32] is a simple extension of Separation Logic in which "a permission to perform one computation step" is also a resource, known as a *credit*. The assertion \$1 represents the unique ownership of one credit. The logic enforces the rule that every function call consumes one credit. Credits do not exist at runtime; they appear only in assertions, such as pre- and postconditions, loop invariants, and data structure invariants. For instance, the Separation Logic triple:

$$\forall\, g\, G.\ \{\, \mathrm{IsGraph}\, g\ G \ *\ \$(3\,|\mathrm{edges}\ G| + 5)\,\}\ \mathit{dfs}(g)\ \{\, \mathrm{IsGraph}\, g\ G\,\}$$

can be read as follows. If initially g is a runtime representation of the graph G and if $3m + 5$ credits are at hand, where m is the number of edges of G, then the function call $\mathit{dfs}(g)$ executes safely and terminates; after this call, g remains a valid representation of G, and no credits remain.

In the *dfs* example, assuming that the assertion $\mathrm{IsGraph}\, g\ G$ is credit-free (which means, roughly, that this assertion definitely does not own any credits), the precondition guarantees the availability of $3m + 5$ credits (and no more), and no credits remain in the postcondition. So, this triple guarantees that the execution of $\mathit{dfs}(g)$ involves at most $3m + 5$ computation steps. Later on in this paper (Section 8), we define $\mathrm{IsGraph}\, g\ G$ in such a way that it is *not* credit-free: its definition involves a nonnegative number of credits. If that were the case in the above example, then $3m + 5$ would have to be interpreted as an amortized bound. Amortization is discussed in greater depth in the next section (Section 4).

Admittedly, $3m + 5$ is too low-level a bound: it would be preferable to state that the cost of $\mathit{dfs}(g)$ is $O(m)$, a more abstract and more robust specification. Following Guéneau et al. [17], this can be expressed in the following style:

$$\begin{aligned}\exists(f : \mathbb{Z} \to \mathbb{Z}).\qquad & \text{nonnegative}\ f\ \wedge\ \text{monotonic}\ f\ \wedge\ f \preceq_{\mathbb{Z}} \lambda m.m \\ & \wedge\ \forall\, g\, G.\ \{\, \mathrm{IsGraph}\, g\ G\ *\ \$\, f(|\mathrm{edges}\ G|)\,\}\ \mathit{dfs}(g)\ \{\, \mathrm{IsGraph}\, g\ G\,\}\end{aligned}$$

The concrete function $\lambda m.(3m + 5)$ is no longer visible; it has been abstracted away under the name f. The specification states that f is nonnegative ($\forall m.\ f(m) \geq 0$), monotonic ($\forall m m'.\ m \leq m' \Rightarrow f(m) \leq f(m')$), and dominated by the function $\lambda m.m$, which means that f grows linearly.

The soundness of Separation Logic with Time Credits stems from the fact that a credit cannot be spent twice. Technically, the soundness metatheorem for Separation Logic with Time Credits guarantees that, for every valid Hoare triple, the following inequality holds:

$$\text{credits in precondition}\ \geq\ \text{steps taken}\ +\ \text{credits in postcondition}.$$

This type of metatheorem is proved by Charguéraud and Pottier [8, §3] and by Mével et al. [32] for Separation Logics with nonnegative credits.

The CFML tool can be viewed as an implementation of Separation Logic with Time Credits for OCaml inside Coq. CFML enables reasoning in forward style. The user inspects the source code, step by step. At each step, she is allowed to visualize and manipulate a description of the current program state in the form of a Separation Logic formula. This formula not only describes the current heap, but also indicates how many time credits are currently available. Guéneau et al. [17, §5, §6] describe the deduction rules of the logic and the manner in which they are applied.

3 Negative Time Credits

In the original presentations of Separation Logic with Time Credits [3, 8, 17, 18], credits are counted in $\mathbb{N}$. This seems natural because $\$n$ is interpreted as a permission to take n steps of computation, and a number of execution steps is never a negative value.

In this setting, credits are affine, that is, it is sound to discard them: the law $\$n \vdash \mathit{true}$ holds. The law $\$(m+n) \equiv \$m * \$n$ holds for every $m, n \in \mathbb{N}$. This splitting law is used when one wishes to spend a subset of the credits at hand. Yet, in practice, the law that is most often needed is a slightly different formulation. Indeed, if n credits are at hand and if one wishes to step over an operation whose cost is m, the appropriate law is $\$n \equiv \$(n-m) * \$m$, which holds only under the side condition $m \leq n$. (This is subtraction in $\mathbb{N}$, so $m > n$ implies $n - m = 0$.)

This side condition gives rise to a proof obligation, and these proof obligations tend to accumulate. If n credits are initially at hand and if one wishes to step over a sequence of k operations whose costs are $m_1, m_2, \ldots, m_k$, then k proof obligations arise: $n - m_1 \geq 0$, $n - m_1 - m_2 \geq 0$, and so on, until $n - m_1 - m_2 - \ldots - m_k \geq 0$. In fact, these proof obligations are redundant: the last one alone implies all of the previous ones. Unfortunately, in an interactive proof assistant such as Coq, it is not easy to take advantage of this fact and present only the last proof obligation to the user. Furthermore, in the proof of Bender et al.'s algorithm, we have encountered a more complex situation where, instead of looking at a straight-line sequence of k operations, one is looking at a loop, whose body is a sequence of operations, and which itself is followed with another sequence of operations. In this situation, proving that the very last proof obligation implies all previous obligations may be possible in principle, but requires a nontrivial strengthening of the loop invariant, which we would rather avoid, if at all possible!

To avoid this accumulation, in this paper, we work in a variant of Separation Logic where Time Credits are counted in $\mathbb{Z}$. Its basic laws are as follows:

$\$0 \equiv \mathit{true}$	zero credit is equivalent to nothing at all
$\$(m+n) \equiv \$m * \$n$	credits are additive
$\$n * [n \geq 0] \vdash \mathit{true}$	nonnegative credits are affine; negative credits are not

Quite remarkably, in the second law, there is no side condition. In particular, this law implies $\$0 \equiv \$n * \$(-n)$, which creates positive credit out of thin air, but creates negative credit at the same time. As put by Tarjan [42], "we can allow borrowing of credits, as long as any debt incurred is eventually paid off". In the third law, the side condition $n \geq 0$ guarantees that a debt cannot be forgotten. Without this requirement, the logic would be unsound, as the second and third laws together would imply $\$0 \vdash \$1$.

Because the second law has no side condition, stepping over a sequence of k operations whose costs are $m_1, m_2, \ldots, m_k$ gives rise to no proof obligation at all. At the end of the sequence, $n - m_1 - m_2 - \ldots - m_k$ credits remain, which the user typically wishes to discard. This is done by applying the third law, giving rise to just one proof obligation: $n - m_1 - m_2 - \ldots - m_k \geq 0$. In summary, switching from $\mathbb{N}$ to $\mathbb{Z}$ greatly reduces the number of proof obligations that appear about credits.

A secondary benefit of this switch is to reduce the number of conversions between $\mathbb{N}$ and $\mathbb{Z}$ that must be inserted in specifications and proofs. Indeed, we model OCaml's signed integers as mathematical integers in $\mathbb{Z}$. (We currently ignore the mismatch between OCaml's limited-precision integers and ideal integers. It should ideally be taken into account, but this is orthogonal to the topic of this paper.)

InitGraph
$\exists k.\ \{\$k\}\ \texttt{init\_graph()}\ \{\lambda g.\mathrm{IsGraph}\ g\ \varnothing\}$

DisposeGraph
$\forall g\, G.\ \mathrm{IsGraph}\ g\ G \Vdash \mathit{true}$

AddVertex
$\forall g\, G\, v.$
$\mathrm{let}\ m, n := |\mathrm{edges}\ G|, |\mathrm{vertices}\ G|\ \mathrm{in}$
$v \notin \mathrm{vertices}\ G \implies$
$\left\{ \begin{array}{l} \mathrm{IsGraph}\ g\ G\ * \\ \$(\psi\,(m, n+1) - \psi\,(m, n)) \end{array} \right\}$
$(\texttt{add\_vertex}\ g\ v)$
$\left\{ \lambda().\ \mathrm{IsGraph}\ g\ (G + v) \right\}$

AddEdge
$\forall g\, G\, v\, w.$
$\mathrm{let}\ m, n := |\mathrm{edges}\ G|, |\mathrm{vertices}\ G|\ \mathrm{in}$
$v, w \in \mathrm{vertices}\ G \wedge (v, w) \notin \mathrm{edges}\ G \implies$
$\left\{ \begin{array}{l} \mathrm{IsGraph}\ g\ G\ * \\ \$(\psi\,(m+1, n) - \psi\,(m, n)) \end{array} \right\}$
$(\texttt{add\_edge\_or\_detect\_cycle}\ g\ v\ w)$
$\left\{ \begin{array}{l} \lambda \mathit{res}.\ \mathrm{match}\ \mathit{res}\ \mathrm{with} \\ \quad \mid \mathrm{Ok} \Rightarrow \mathrm{IsGraph}\ g\ (G + (v, w)) \\ \quad \mid \mathrm{Cycle} \Rightarrow [w \longrightarrow_G^* v] \end{array} \right\}$

Acyclicity
$\forall g\, G.\ \mathrm{IsGraph}\ g\ G \Vdash$
$\quad \mathrm{IsGraph}\ g\ G\ *\ [\forall x.\ x \not\longrightarrow_G^+ x]$

Complexity
$\mathrm{nonnegative}\ \psi\ \wedge\ \mathrm{monotonic}\ \psi\ \wedge$
$\psi \preceq_{\mathbb{Z}\times\mathbb{Z}} \lambda(m, n).\ (m \cdot \min(m^{1/2}, n^{2/3}) + n)$

Figure 1 Specification of an incremental cycle detection algorithm.

Because negative time credits are not affine, it is not the case here that every assertion is affine, as in Iris [24] or in earlier versions of CFML. Affine and non-affine assertions must now be distinguished: a points-to assertion, which describes a heap-allocated object, remains affine; the assertion $\$n$ is affine if and only if n is nonnegative; an abstract assertion, such as $\mathrm{IsGraph}\ g\ G$, may or may not be affine, depending on the definition of IsGraph. (Here, it is in fact affine; see §4 and DisposeGraph in Figure 1.) We have adapted CFML so as to support this distinction.

From a metatheoretical perspective, the introduction of negative time credits requires adapting the proof of soundness of Separation Logic with Time Credits. We have successfully updated our pre-existing Coq proof of this result [8]; an updated proof is available online (Supplement Material).

4 Specification of the Algorithm

The interface for an incremental cycle detection algorithm consists of three public operations: `init_graph`, which creates a fresh empty graph, `add_vertex`, which adds a vertex, and `add_edge_or_detect_cycle`, which either adds an edge or report that this edge cannot be added because it would create a cycle.

Figure 1 shows a formal specification for an incremental cycle detection algorithm. It consists of six statements. InitGraph, AddVertex, and AddEdge are Separation Logic triples: they assign pre- and postconditions to the three public operations. DisposeGraph and Acyclicity are Separation Logic entailments. The last statement, Complexity, provides a complexity bound. It is the only statement that is specific to the algorithm discussed in this paper. Indeed, the first five statements form a generic specification, which any incremental cycle detection algorithm could satisfy.

The six statements in the specification share two variables, namely IsGraph and ψ. These variables are implicitly existentially quantified in front of the specification: a user of the algorithm must treat them as abstract.

The predicate IsGraph is an *abstract representation predicate*, a standard notion in Separation Logic [35]. It is parameterized with a memory location g and with a mathematical graph G. The assertion IsGraph g G means that a well-formed data structure, which represents the mathematical graph G, exists at address g in memory. At the same time, this assertion denotes the unique ownership of this data structure.

Because this is Separation Logic with Time Credits, the assertion IsGraph g G can also represent the ownership of a certain number of credits. For example, for the specific algorithm considered in this paper, we later define IsGraph g G as $\exists L.\ \$\phi(G, L) * \ldots$ (Section 8), where ϕ is a suitable potential function [42]. ϕ is parameterized by the graph G and by a map L of vertices to integer levels. Intuitively, this means that $\phi(G, L)$ credits are stored in the data structure. These details are hidden from the user: ϕ does not appear in Figure 1. Yet, the fact that IsGraph g G can involve credits means that the user must read AddVertex and AddEdge as amortized specifications [42]: the actual cost of a single `add_vertex` or `add_edge_or_detect_cycle` operation is not directly related to the number of credits that explicitly appear in the precondition of this operation.

The function ψ has type $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$. In short, $\psi(m, n)$ is meant to represent the advertised cost of a sequence of n vertex creation and m edge creation operations. In other words, it is the number of credits that one must pay in order to create n vertices and m edges. This informal claim is explained later on in this section.

InitGraph states that the function call `init_graph()` creates a valid data structure, which represents the empty graph $\varnothing$, and returns its address g. Its cost is k, where k is an unspecified constant; in other words, its complexity is $O(1)$.

DisposeGraph states that the assertion IsGraph g G is affine: that is, it is permitted to forget about the existence of a valid graph data structure. By publishing this statement, we guarantee that we are not hiding a debt inside the abstract predicate IsGraph. Indeed, to prove that DisposeGraph holds, we must verify that the potential $\phi(G, L)$ is nonnegative (Section 3).

AddVertex states that `add_vertex` requires a valid data structure, described by the assertion IsGraph g G, and returns a valid data structure, described by IsGraph g $(G + v)$. (We write $G + v$ for the result of extending the mathematical graph G with a new vertex v and $G + (v, w)$ for the result of extending G with a new edge from v to w.) In addition, `add_vertex` requires $\psi(m, n + 1) - \psi(m, n)$ credits. These credits are not returned: they do not appear in the postcondition. They either are actually consumed or become stored inside the data structure for later use. Thus, one can think of $\psi(m, n + 1) - \psi(m, n)$ as the amortized cost of `add_vertex`.

Similarly, AddEdge states that the cost of `add_edge_or_detect_cycle` is $\psi(m+1, n) - \psi(m, n)$. This operation returns either Ok, in which case the graph has been successfully extended with a new edge from v to w, or Cycle, in which case this new edge cannot be added, because there already is a path in G from w to v. (The proposition $w \longrightarrow_G^* v$ appears within square brackets, which convert an ordinary proposition to a Separation Logic assertion.) In the latter case, the data structure is invalidated: the assertion IsGraph g G is not returned. Thus, in that case, no further operations on the graph are allowed.

By combining the first four statements in Figure 1, a client can verify that a call to `init_graph`, followed with an arbitrary interleaving of n calls to `add_vertex` and m successful calls to `add_edge_or_detect_cycle`, satisfies the specification $\{\$(k + \psi(m, n))\}\ \ldots\ \{true\}$, where k is the cost of `init_graph`. Indeed, the cumulated cost of the calls to `add_vertex` and `add_edge_or_detect_cycle` forms a telescopic sum that adds up to $\psi(m, n) - \psi(0, 0)$, which itself is bounded by $\psi(m, n)$.

To insert a new edge from v to w and detect potential cycles:
- If $L(v) < L(w)$, insert the edge (v, w), declare success, and exit
- Perform a backward search:
  - start from v
  - follow an edge (backward) only if its source vertex x satisfies $L(x) = L(v)$
  - if w is reached, declare failure and exit
  - if F edges have been traversed, interrupt the backward search
    - → in Bender et al.'s algorithm, F is a constant Δ
    - → in our algorithm, F is $L(v)$
- If the backward search was not interrupted, then:
  - if $L(w) = L(v)$, insert the edge (v, w), declare success, and exit
  - otherwise set $L(w)$ to $L(v)$
- If the backward search was interrupted, then set $L(w)$ to $L(v) + 1$
- Perform a forward search:
  - start from w
  - upon reaching a vertex x:
    - if x was visited during the backward search, declare failure and exit
    - if $L(x) \geq L(w)$, do not traverse through x
    - if $L(x) < L(w)$, set $L(x)$ to $L(w)$ and traverse x
- Finally, insert the edge (v, w), declare success, and exit

Figure 2 Pseudocode for Bender et al.'s algorithm and for our improved algorithm.

Since Separation Logic with Time Credits is sound, the triple $\{\$(k+\psi(m,n))\}\ \ldots\ \{\mathit{true}\}$ implies that the actual worst-case cost of the sequence of operations is $k + \psi(m, n)$. This confirms our earlier informal claim that $\psi(m, n)$ represents the cost of creating n vertices and m edges.

Acyclicity states that, from the Separation Logic assertion IsGraph g G, the user can deduce that G is acyclic. In other words, as long as the data structure remains in a valid state, the graph G remains acyclic.

Although the exact definition of ψ is not exposed, Complexity provides an asymptotic bound: $\psi(m, n) \in O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$. Technically, the relation $\preceq_{\mathbb{Z}\times\mathbb{Z}}$ is a domination relation between functions of type $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ [17]. Our complexity bound thus matches the one published by Bender et al. [4].

5 Overview of the Algorithm

We provide pseudocode for Bender et al.'s algorithm [4, §2] and for our improved algorithm in Figure 2. The only difference between the two algorithms is the manner in which a certain internal parameter, named F, is set. The value of F influences the complexity of the algorithm, not its correctness.

When the user requests the creation of an edge from v to w, finding out whether this operation would create a cycle amounts to determining whether a path already exists from w to v. A naïve algorithm could search for such a path by performing a forward search, starting from w and attempting to reach v.

One key feature of Bender et al.'s algorithm is that a positive integer level $L(v)$ is associated with every vertex v, and the following invariant is maintained: L forms a pseudo-topological numbering. That is, "no edge goes down": if there is an edge from v to w, then $L(v) \leq L(w)$ holds. The presence of levels can be exploited to accelerate a search: for

instance, during a forward search whose purpose is to reach the vertex v, any vertex whose level is greater than that of v can be disregarded. The price to pay is that the invariant must be maintained: when a new edge is inserted, the levels of some vertices must be adjusted.

A second key feature of Bender et al.'s algorithm is that it not only performs a forward search, but begins with a backward search that is both *restricted* and *bounded*. It is restricted in the sense that it searches only one level of the graph: starting from v, it follows only *horizontal* edges, that is, edges whose endpoints are both at the same level. Therefore, all of the vertices that it discovers are at level $L(v)$. It is bounded in the sense that it is interrupted, even if incomplete, after it has processed a predetermined number of edges, denoted by the letter F in Figure 2.

A third key characteristic of Bender et al.'s algorithm is the manner in which levels are updated so as to maintain the invariant when a new edge is inserted. Bender et al. adopt the policy that the level of a vertex can never decrease. Thus, when an edge from v to w is inserted, all of the vertices that are accessible from w must be promoted to a level that is at least the level of v. In principle, there are many ways of doing so. Bender et al. proceed as follows: if the backward search was not interrupted, then w and its descendants are promoted to the level of v; otherwise, they are promoted to the next level, $L(v) + 1$. In the latter case, $L(v) + 1$ is possibly a new level. We see that such a new level can be created only if the backward search has not completed, that is, only if there exist at least F edges at level $L(v)$. In short, a new level may be created only if the previous level contains sufficiently many edges. This mechanism is used to control the number of levels.

The last key aspect of Bender et al.'s algorithm is the choice of F. On the one hand, as F increases, backward searches become more expensive, as each backward search processes up to F edges. On the other hand, as F decreases, forward searches become more expensive. Indeed, a smaller value of F leads to the creation of a larger number of levels, and (as explained later) the total cost of the forward searches is proportional to the number of levels.

Bender et al. set F to a constant Δ, defined as $\min(m^{1/2}, n^{2/3})$ throughout the execution of the algorithm, where m and n are upper bounds on the *final* numbers of edges and vertices in the graph. As explained earlier (Section 1), though, it seems preferable to set F to a value that does not depend on such upper bounds, as they may not be known ahead of time. In our modified algorithm, F stands for $L(v)$, where v is the source of the edge that is being inserted. This value depends only on the *current* state of the data structure, so our algorithm is truly online. We prove that it has the same complexity as Bender et al.'s original algorithm, namely $O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$.

6 Informal Complexity Analysis

We now present an informal complexity analysis of Bender et al.'s original algorithm. In this algorithm, the parameter F is fixed: it remains constant throughout the execution of the algorithm. Under this hypothesis, the following invariant holds: for every level k except the highest level, there exist at least F horizontal edges at level k (edges whose endpoints are both at level k). A proof is given in Appendix A of the extended version.

From this invariant, one can derive two upper bounds on the number of levels. Let K denote the number of nonterminal levels. First, the invariant implies $m \geq KF$, therefore $K \leq m/F$. Furthermore, for each nonterminal level k, the vertices at level k form a subgraph with at least F edges, which therefore must have at least $\sqrt{F}$ vertices. In other words, at every nonterminal level, there are at least $\sqrt{F}$ vertices. This implies $n \geq K\sqrt{F}$, therefore $K \leq n/\sqrt{F}$.

Let us estimate the algorithm's complexity. Consider a sequence of n vertex creation and m edge creation operations. The cost of one backward search is $O(F)$, as it traverses at most F edges. Because each edge insertion triggers one such search, the total cost of the backward searches is $O(mF)$. The forward search traverses an edge if and only if this edge's source vertex is promoted to a higher level. Therefore, the cost of a forward search is linear in the number of edges whose source vertex is thus promoted. Because there are m edges and because a vertex can be promoted at most K times, the total cost of the forward searches is $O(mK)$. In summary, the cost of this sequence of operations is $O(mF + mK)$.

By combining this result with the two bounds on K obtained above, one finds that the complexity of the algorithm is $O(m \cdot (F + \min(m/F, n/\sqrt{F}))$. A mathematical analysis (Appendix A of the extended version) shows that setting F to Δ, where Δ is defined as $\min(m^{1/2}, n^{2/3})$, leads to the asymptotic bound $O(m \cdot \min(m^{1/2}, n^{2/3}))$. This completes our informal analysis of Bender et al.'s original algorithm.

In our modified algorithm, in contrast, F is not a constant. Instead, each edge insertion operation has its own value of F: indeed, we let F stand for $L(v)$, where v is the source vertex of the edge that is being inserted. We are able to establish the following invariant: for every level k except the highest level, there exist at least k horizontal edges at level k. This subsequently allows us to establish a bound on the number of levels: we prove that $L(v)$ is bounded by a quantity that is asymptotically equivalent to Δ.

7 Implementation

Our OCaml code, shown in Figure 3, relies on auxiliary operations whose implementation belongs in a lower layer. We do not prescribe how they should be implemented and what data structures they should rely upon; instead, we provide a specification for each of them, and prove that our algorithm is correct, regardless of which implementation choices are made. We provide and verify one concrete implementation, so as to guarantee that our requirements can be met.

For brevity, we do not give the specifications of these auxiliary operations. Instead, we list them and briefly describe what they are supposed to do. Each of them is required to have constant time complexity.

To update the graph, the algorithm requires the ability to create new vertices and new edges (`create_vertex` and `add_edge`). To avoid creating duplicate edges, it must be able to test the equality of two vertices (`vertex_eq`).

The backward search requires the ability to efficiently enumerate the horizontal incoming edges of a vertex (`get_incoming`). The collection of horizontal incoming edges of a vertex y is updated during a forward search. It is reset when the level of y is increased (`clear_incoming`). An edge is added to it when a horizontal edge from x to y is traversed (`add_incoming`). The backward search also requires the ability to generate a fresh mark (`new_mark`), to mark a vertex (`set_mark`), and to test whether a vertex is marked (`is_marked`). These marks are consulted also during the forward search.

The forward search requires the ability to efficiently enumerate the outgoing edges of a vertex (`get_outgoing`). It also reads and updates the level of certain vertices (`get_level`, `set_level`).

Several choices arise in the implementation of graph search. First, the frontier can be either implicit, if the search is formulated as a recursive function, or represented as an explicit data structure. We choose the latter approach, as it lends itself better to the implementation of an interruptible search. Second, one must choose between an imperative style, where the

```
let rec visit_backward g target mark fuel stack =
  match stack with
  | [] -> VisitBackwardCompleted
  | x :: stack ->
    let (stack, fuel), interrupted = interruptible_fold (fun y (stack, fuel) ->
      if fuel = 0 then Break (stack, -1)
      else if vertex_eq y target then Break (stack, fuel)
      else if is_marked g y mark then Continue (stack, fuel - 1)
      else (set_mark g y mark; Continue (y :: stack, fuel - 1))
    ) (get_incoming g x) (stack, fuel) in
    if interrupted
    then if fuel = -1 then VisitBackwardInterrupted else VisitBackwardCyclic
    else visit_backward g target mark fuel stack

let backward_search g v w fuel =
  let mark = new_mark g in
  let v_level = get_level g v in
  set_mark g v mark;
  match visit_backward g w mark fuel [v] with
  | VisitBackwardCyclic -> BackwardCyclic
  | VisitBackwardInterrupted -> BackwardForward (v_level + 1, mark)
  | VisitBackwardCompleted -> if get_level g w = v_level
                              then BackwardAcyclic
                              else BackwardForward (v_level, mark)

let rec visit_forward g new_level mark stack =
  match stack with
  | [] -> ForwardCompleted
  | x :: stack ->
    let stack, interrupted = interruptible_fold (fun y stack ->
      if is_marked g y mark then Break stack
      else
        let y_level = get_level g y in
        if y_level < new_level then begin
          set_level g y new_level;
          clear_incoming g y;
          add_incoming g y x;
          Continue (y :: stack)
        end else if y_level = new_level then begin
          add_incoming g y x;
          Continue stack
        end else Continue stack
    ) (get_outgoing g x) stack in
    if interrupted then ForwardCyclic
    else visit_forward g new_level mark stack

let forward_search g w new_w_level mark =
  clear_incoming g w;
  set_level g w new_w_level;
  visit_forward g new_w_level mark [w]

let add_edge_or_detect_cycle (g : graph) (v : vertex) (w : vertex) =
  let succeed () = add_edge g v w; Ok in
  if vertex_eq v w then Cycle
  else if get_level g w > get_level g v then succeed ()
  else match backward_search g v w (get_level g v) with
    | BackwardCyclic -> Cycle
    | BackwardAcyclic -> succeed ()
    | BackwardForward (new_level, mark) ->
        match forward_search g w new_level mark with
        | ForwardCyclic -> Cycle
        | ForwardCompleted -> succeed ()
```

Figure 3 OCaml implementation of the verified incremental cycle detection algorithm.

frontier is represented as a mutable data structure and the code is structured in terms of "while" loops and "break" and "continue" instructions, and a functional style, where the frontier is an immutable data structure and the code is organized in terms of tail-recursive functions or higher-order loop combinators. Because OCaml does not have "break" and "continue", we choose the latter style.

The function `visit_backward`, for instance, can be thought of as two nested loops. The outer loop is encoded via a tail call to `visit_backward` itself. This loop runs until the stack is exhausted or the inner loop is interrupted. The inner loop is implemented via the loop combinator `interruptible_fold`, a functional-style encoding of a "for" loop whose body may choose between interrupting the loop (`Break`) and continuing (`Continue`). This inner loop iterates over the horizontal incoming edges of the vertex x. It is interrupted when a cycle is detected or when the variable `fuel`, whose initial value corresponds to F (Section 5), reaches zero.

The main public entry point of the algorithm is `add_edge_or_detect_cycle`, whose specification was presented in Figure 1. The other two public functions, `init_graph` and `add_vertex`, are trivial; they are not shown.

8 Data Structure Invariants

As explained earlier (Section 4), the specification of the algorithm refers to two variables, IsGraph and ψ, which must be regarded as abstract by a client. Figure 4 gives their formal definitions. The assertion IsGraph g G captures both the invariants required for functional correctness and those required for the complexity analysis. It is a conjunction of three conjuncts, which we describe in turn.

The conjunct IsRawGraph g G L M I asserts that there is a data structure at address g in memory, claims the unique ownership of this data structure, and summarizes the information that is recorded in this structure. The parameters G, L, M, I together form a logical model of this data structure: G is a mathematical graph; L is a map of vertices to integer levels; M is a map of vertices to integer marks; and I is a map of vertices to sets of vertices, describing horizontal incoming edges. The parameters L, M and I are existentially quantified in the definition of IsGraph, indicating that they are internal data whose existence is not exposed to the user.

The second conjunct, [Inv G L I], is a pure proposition that relates the graph G with the maps L and I. Its definition appears next in Figure 4. Anticipating on the fact that we sometimes need a relaxed invariant, we actually define a more general predicate InvExcept E G L I, where E is a set of "exceptions", that is, a set of vertices where certain properties are allowed *not* to hold. Instantiating E with the empty set $\emptyset$ yields Inv G L I.

The proposition InvExcept E G L I is a conjunction of five properties. The first four capture functional correctness invariants: the graph G is acyclic, every vertex has positive level, L forms a pseudo-topological numbering of G, and the sets of horizontal incoming edges represented by I are accurate with respect to G and L. The last property plays a crucial role in the complexity analysis (Section 6). It asserts that "every vertex has enough coaccessible edges at the previous level": for every vertex x at level $k+1$, there must be at least k horizontal edges at level k from which x is accessible. The vertices in the set E may disobey this property, which is temporarily broken during a forward search.

The last conjunct in the definition of IsGraph is $\$\phi(G, L)$. This is a *potential* [42], a number of credits that have been received from the user (through calls to `add_vertex` and `add_edge_or_detect_cycle`) and not yet spent. $\phi(G, L)$ is defined as $C \cdot (\text{net } G\ L)$. The

$$\text{IsGraph}\, g\, G := \exists L\, M\, I.\ \text{IsRawGraph}\, g\, G\, L\, M\, I * [\text{Inv}\, G\, L\, I] * \$\phi(G, L)$$

$$\text{Inv}\, G\, L\, I := \text{InvExcept}\, \emptyset\, G\, L\, I$$

$$\text{InvExcept}\, E\, G\, L\, I := \begin{cases} \textit{acyclicity:} & \forall x.\quad x \not\longrightarrow_G^+ x \\ \textit{positive levels:} & \forall x.\quad L(x) \geq 1 \\ \textit{pseudo-topological numbering:} & \forall x\, y.\ x \longrightarrow_G y \implies L(x) \leq L(y) \\ \textit{horizontal incoming edges:} & \forall x\, y.\ x \in I(y) \iff x \longrightarrow_G y \ \wedge\ L(x) = L(y) \\ \textit{replete levels:} & \forall x.\quad x \in E \vee \text{enoughEdgesBelow}\, G\, L\, x \end{cases}$$

$$\text{enoughEdgesBelow}\, G\, L\, x := |\text{coaccEdgesAtLevel}\, G\, L\, k\, x| \geq k \quad \text{where } k = L(x) - 1$$

$$\text{coaccEdgesAtLevel}\, G\, L\, k\, x := \{\, (y, z) \mid y \longrightarrow_G z \longrightarrow_G^* x \ \wedge\ L(y) = L(z) = k \,\}$$

$$\left.\begin{array}{lcl} \phi(G, L) & := & C \cdot (\text{net}\, G\, L) \\ \text{net}\, G\, L & := & \text{received}\, m\, n - \text{spent}\, G\, L \end{array}\right\} \text{where } m = |\text{edges}\, G| \text{ and } n = |\text{vertices}\, G|$$

$$\text{spent}\, G\, L := \sum_{(u,v)\, \in\, \text{edges}\, G} L(u)$$

$$\text{received}\, m\, n := m \cdot (\text{maxLevel}\, m\, n + 1)$$

$$\text{maxLevel}\, m\, n := \min(\lceil (2m)^{1/2} \rceil, \lfloor (\tfrac{3}{2} n)^{2/3} \rfloor) + 1$$

$$\psi(m, n) := C' \cdot (\text{received}\, m\, n + m + n)$$

Figure 4 Definitions of IsGraph and ψ, with auxiliary definitions.

constant C is derived from the code; its exact value is in principle known, but irrelevant, so we refer to it only by name. The quantity "net G L" is defined as the difference between "received m n", an amount that has been received, and "spent G L", an amount that has been spent. "net G L" can also be understood as a sum over all edges of a per-edge amount, which for each edge (u, v) is "maxLevel m $n - L(u)$". This is a difference between "maxLevel m n", which one can prove is an upper bound on the current level of every vertex, and $L(u)$, the current level of the vertex u. This difference can be intuitively understood as the number of times the edge (u, v) might be traversed in the future by a forward search, due to a promotion of its source vertex u.

We have reviewed the three conjuncts that form IsGraph g G. There remains to define ψ, which also appears in the public specification (Figure 1). Recall that $\psi(m, n)$ denotes the number of credits that we request from the user during a sequence of m edge additions and n vertex additions. Up to another known-but-irrelevant constant factor C', it is defined as "$m + n +$ received m n", that is, a constant amount per operation plus a sufficient amount to justify that $\phi(m, n)$ credits are at hand, as claimed by the invariant IsGraph g G. It is easy to check, by inspection of the last few definitions in Figure 4, that Complexity is satisfied, that is, $\psi(m, n)$ is $O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$.

The public function `add_edge_or_detect_cycle` expects a graph g and two vertices v and w. Its public specification has been presented earlier (Figure 1). The top part of Figure 5 shows the same specification, where IsGraph (01) and ψ (02) have been unfolded. This shows that we receive time credits from two different sources: the potential of the data structure, on the one hand, and the credits supplied by the user for this operation, on the other hand.

$$\forall g\ G\ L\ M\ I\ v\ w.\ \text{let } m := |\text{edges } G| \text{ and } n := |\text{vertices } G| \text{ in}$$
$$v, w \in \text{vertices } G \wedge (v, w) \notin \text{edges } G \implies$$

$$\left\{ \begin{array}{ll} \text{IsRawGraph } g\ G\ L\ M\ I\ *\ [\text{Inv } G\ L\ I]\ *\ \$\phi(G, L)\ * & (01) \\ \$(C' \cdot (\text{received } (m+1)\ n - \text{received } m\ n + 1)) & (02) \end{array} \right\}$$
$$(\texttt{add\_edge\_or\_detect\_cycle}\ g\ v\ w)$$

$$\left\{ \begin{array}{l} \lambda res.\ \text{match } res \text{ with} \\ \quad \mid \text{Ok} \Rightarrow\ \text{let } G' := G + (v, w) \text{ in } \exists L'\ M'\ I'. \\ \qquad\qquad \text{IsRawGraph } g\ G'\ L'\ M'\ I'\ *\ [\text{Inv } G'\ L'\ I']\ *\ \$\phi(G', L') \\ \quad \mid \text{Cycle} \Rightarrow [w \longrightarrow_G^* v] \end{array} \right\}$$

Figure 5 Specifications for edge creation, after unfolding of the representation predicate.

9 Specifications for the Algorithm's Main Functions

The specifications of the two search functions, `backward_search` and `forward_search`, appear in Figure 6. They capture the algorithm's key internal invariants and spell out exactly what each search achieves and how its cost is accounted for.

The function `backward_search` expects a nonnegative integer *fuel*, which represents the maximum number of edges that the backward search is allowed to process. In addition, it expects a graph g and two distinct vertices v and w which must satisfy $L(w) \leq L(v)$. (If that is not the case, an edge from v to w can be inserted immediately.) The graph must be in a valid state (03). The specification requires $A \cdot \mathit{fuel} + B$ credits to be provided (04), for some known-but-irrelevant constants A and B. Indeed, the cost of a backward search is linear in the number of edges that are processed, therefore linear in *fuel*.

This function returns either BackwardCyclic, BackwardAcyclic, or a value of the form BackwardForward(k, *mark*). The first line in the postcondition (05) asserts that the graph remains valid and changes only in that some marks are updated: M changes to M'.

The remainder of the postcondition depends on the function's return value, *res*. If it is BackwardCyclic, then there exists a path in G from w to v (06). If it is BackwardAcyclic, then v and w are at the same level and there is no path from w to v (07). In this case, no forward search is needed. If it is of the form BackwardForward(k, *mark*), then a forward search is required.

In the latter case, the integer k is the level to which the vertex w and its descendants should be promoted during the subsequent forward search. The value *mark* is the mark that was used by this backward search; the subsequent forward search uses this mark to recognize vertices reached by the backward search. The postcondition asserts that the vertex v is marked, whereas w is not (08), since it has not been reached. Moreover, every marked vertex lies at the same level as v and is an ancestor of v (09). Finally, one of the following two cases holds. In the first case, w must be promoted to the level of v and currently lies below the level of v (10) and the backward search is complete, that is, every ancestor of v that lies at the level of v is marked (11). In the second case, w must be promoted to level $L(v) + 1$ and there exist at least *fuel* horizontal edges at the level of v from which v can be reached (12).

The function `forward_search` expects the graph g, the target vertex w, the level k to which w and its descendants should be promoted, and the mark *mark* used by the backward search. The vertex w must be at a level less than k and must be unmarked. The graph must be in a valid state (13). The forward search requires a constant amount of credits B'. Furthermore, it requires access to the potential $\$\phi(G, L)$, which is used to pay for edge processing costs.

$$
\begin{array}{l}
\forall \mathit{fuel}\ g\ G\ L\ M\ I\ v\ w. \\
\mathit{fuel} \geq 0 \ \wedge\ v, w \in \text{vertices}\ G \ \wedge\ v \neq w \ \wedge\ L(w) \leq L(v) \implies \\
\left\{ \begin{array}{ll}
\text{IsRawGraph}\ g\ G\ L\ M\ I\ *\ [\text{Inv}\ G\ L\ I]\ * & (03) \\
\$(A \cdot \mathit{fuel} + B) & (04)
\end{array} \right\} \\
(\texttt{backward\_search}\ g\ v\ w\ \mathit{fuel}) \\
\left\{ \begin{array}{ll}
\lambda \mathit{res}.\ \exists M'. & \\
\quad \text{IsRawGraph}\ g\ G\ L\ M'\ I\ *\ [\text{Inv}\ G\ L\ I]\ * & (05) \\
\quad [\text{match}\ \mathit{res}\ \text{with} & \\
\quad \mid \text{BackwardCyclic} \Rightarrow w \longrightarrow_G^* v & (06) \\
\quad \mid \text{BackwardAcyclic} \Rightarrow L(v) = L(w) \wedge w \not\longrightarrow_G^* v & (07) \\
\quad \mid \text{BackwardForward}(k, \mathit{mark}) \Rightarrow & \\
\qquad M'\ v = \mathit{mark} \wedge M'\ w \neq \mathit{mark} \wedge & (08) \\
\qquad (\forall x.\ M'\ x = \mathit{mark} \implies L(x) = L(v) \wedge x \longrightarrow_G^* v) \wedge & (09) \\
\qquad (\quad (k = L(v) \wedge L(w) < L(v)\ \wedge & (10) \\
\qquad\quad \forall x.\ L(x) = L(v) \wedge x \longrightarrow_G^* v \implies M'\ x = \mathit{mark}) & (11) \\
\qquad \vee\ (k = L(v) + 1\ \wedge\ \mathit{fuel} \leq |\text{coaccEdgesAtLevel}\ G\ L\ (L(v))\ v|))] & (12)
\end{array} \right\}
\end{array}
$$

$$
\begin{array}{l}
\forall g\ G\ L\ M\ I\ w\ k\ \mathit{mark}. \\
w \in \text{vertices}\ G \ \wedge\ L(w) < k \ \wedge\ M\ w \neq \mathit{mark} \implies \\
\left\{ \begin{array}{ll}
\text{IsRawGraph}\ g\ G\ L\ M\ I\ *\ [\text{Inv}\ G\ L\ I]\ * & (13) \\
\$(B' + \phi(G, L)) & (14)
\end{array} \right\} \\
(\texttt{forward\_search}\ g\ w\ k\ \mathit{mark}) \\
\left\{ \begin{array}{ll}
\lambda \mathit{res}.\ \exists L'\ I'. & \\
\quad \text{IsRawGraph}\ g\ G\ L'\ M\ I'\ * & (15) \\
\quad [L'(w) = k \wedge (\forall x.\ L'(x) = L(x) \vee w \longrightarrow_G^* x)]\ * & (16) \\
\quad \text{match}\ \mathit{res}\ \text{with} & \\
\quad \mid \text{ForwardCyclic} \Rightarrow [\exists x.\ M\ x = \mathit{mark} \wedge w \longrightarrow_G^* x] & (17) \\
\quad \mid \text{ForwardCompleted} \Rightarrow & \\
\qquad \$\phi(G, L')\ * & (18) \\
\qquad [(\forall x\ y.\ L(x) < k \wedge w \longrightarrow_G^* x \longrightarrow_G y \implies M\ y \neq \mathit{mark}) \wedge & (19) \\
\qquad\ \text{InvExcept}\ \{x \mid w \longrightarrow_G^* x \wedge L'(x) = k\}\ G\ L'\ I'] & (20)
\end{array} \right\}
\end{array}
$$

Figure 6 Specifications for the main two auxiliary functions.

This function returns either ForwardCyclic or ForwardCompleted. It affects the low-level graph data structure by updating certain levels and certain sets of horizontal incoming edges: L and I are changed to L' and I' (15). The vertex w is promoted to level k, and a vertex x can be promoted only if it is a descendant of w (16).

If the return value is ForwardCyclic, then, according to the postcondition, there exists a vertex x that is accessible from w and that has been marked by the backward search (17). This implies that there is a path from w through x to v. Thus, adding an edge from v to w would create a cycle. In this case, the data structure invariant is lost.

If the return value is ForwardCompleted, then, according to the postcondition, $\phi(G, L')$ credits are returned (18). This is precisely the potential of the data structure in its new state. Furthermore, two logical propositions hold. First (19), the forward search has not

encountered a marked vertex: for every edge (x, y) that is accessible from w, where x is at level less than k, the vertex y is unmarked. (This implies that there is no path from w to v.) Second (20), the invariant Inv $G\ L'\ I'$ is satisfied *except* for the fact that the property of "replete levels" (Figure 4) may be violated at descendants of w whose level is now k. Fortunately, this proposition (20), combined with a few other facts that are known to hold at the end of the forward search, implies Inv $G'\ L'\ I'$, where G' stands for $G + (v, w)$. In other words, at the end of the forward search, all levels and all sets of horizontal incoming edges are consistent with the mathematical graph G', where the edge (v, w) exists. Thus, after this edge is effectively created in memory by the call `add_edge g v w`, all is well: we have both IsRawGraph $g\ G'\ L'\ M'\ I'$ and Inv $G'\ L'\ I'$, so `add_edge_or_detect_cycle` satisfies its postcondition, under the form shown in Figure 6.

10 Related Work

Neither interactive program verification nor Separation Logic with Time Credits are new (Section 1). Outside the realm of Separation Logic, several researchers present machine-checked complexity analyses, carried out in interactive proof assistants. Van der Weegen and McKinna [46] study the average-case complexity of Quicksort, represented in Coq as a monadic program. The monad is used to introduce both nondeterminism and comparison-counting. Danielsson [11] implements a *Thunk* monad in Agda and uses it to reason about the amortized complexity of data structures that involve delayed computation and memoization. McCarthy et al. [30] present a monad that allows the time complexity of a Coq computation to be expressed in its type. Nipkow [33] proposes machine-checked amortized complexity analyses of several data structures in Isabelle/HOL. The code is manually transformed into a cost function.

Several mostly-automated program verification systems can verify complexity bounds. Madhavan et al. [29] present such a system, which can deal with programs that involve memoization, and is able to infer some of the constants that appear in user-supplied complexity bounds. Srikanth et al. [40] propose an automated verifier for user-supplied complexity bounds that involve polynomials, exponentials, and logarithms. When a bound is not met, a counter-example can be produced. Such automated tools are inherently limited in the scope of programs that they can handle. For instance, the algorithm considered in the present paper appears to be far beyond reach of any of these fully automated tools.

There is also a vast body of work on fully-automated inference of complexity bounds, beginning with Wegbreit [47] and continuing with more recent papers and tools [15, 14, 2, 13, 20]. Carbonneaux et al.'s analysis produces certificates whose validity can be checked by Coq [5]. It is possible in principle to express these certificates as derivations in Separation Logic with Time Credits. This opens the door to provably-safe combinations of automated and interactive tools.

Finally, there is a rich literature on static and dynamic analyses that aim at detecting performance anomalies [34, 31, 21, 28, 45].

Early work on the verification of garbage collection algorithms includes, in some form, the verification of a graph traversal. For example, Russinoff [38] uses the Boyer-Moore theorem prover to verify Ben Ari's incremental garbage collector, which employs a two-color scheme. In more recent work, specifically focused on the verification of graph algorithms, Lammich [25], Pottier [36], and Chen et al. [10, 9] verify various algorithms for finding the strongly connected components of a directed graph. In particular, Chen et al. [9] repeat a single proof using Why3, Coq and Isabelle. None of these works include a verification of asymptotic complexity.

11 Conclusion

In this paper, we have used a powerful program logic to simultaneously verify the correctness and complexity of an actual implementation of a state-of-the-art incremental cycle detection algorithm. Although neither interactive program verification nor Separation Logic with Time Credits are new, there are still relatively few examples of applying this simultaneous-verification approach to nontrivial algorithms or data structures. We hope we have demonstrated that this approach is indeed viable, and can be applied to a wide range of algorithms, including ones that involve mutable state, dynamic memory allocation, higher-order functions, and amortization.

As a technical contribution, whereas all previous works use credits in $\mathbb{N}$, we use credits in $\mathbb{Z}$ and allow negative credits to exist temporarily. We explain in Section 3 why this is safe and convenient.

Following Guéneau et al. [17], our public specification exposes an asymptotic complexity bound: no literal constants appear in it. We remark, however, that it is often difficult to use something that resembles the O notation in specifications and proofs. Indeed, in its simplest form, a use of this notation in a mathematical statement $S[O(g)]$ can be understood as an occurrence of a variable f that is existentially quantified at the beginning of the statement: $\exists f.\ (f \preceq g) \wedge S[f]$. An example of such a statement was given earlier (Section 2). Here, f denotes an unknown function, which is dominated by the function g. The definition of the domination relation $\preceq$ involves further quantifiers [17]. In the analysis of a complex algorithm or data structure, however, it is often the case that an existential quantifier must be hoisted very high, so that its scope encompasses not just a single statement, but possibly a group of definitions, statements, and proofs. The present paper shows several instances of this phenomenon. In the public specification (Figure 1), the cost function ψ must be existentially quantified at the outermost level. In the definition of the data structure invariant (Figure 4) and in the proofs that involve this invariant, several constants appear, such as C and C', which must be defined beforehand. Thus, even if one could formally use $S[O(g)]$ as syntactic sugar for $\exists f.\ (f \preceq g) \wedge S[f]$, we fear that one might not be able to use this sugar very often, because a lot of mathematical work is carried out under the existential quantifier, in a context where f must be explicitly referred to by name. That said, novel ways of understanding the O notation may permit further progress; Affeldt et al. [1] make interesting steps in such a direction.

In future work, we would like to verify the algorithm that is used in the kernel of Coq to check the satisfiability of universe constraints. These are conjunctions of strict and non-strict ordering constraints, $x < y$ and $x \leq y$. This requires an incremental cycle detection algorithm that maintains strong components. Bender et al. [4, §5] present such an algorithm. It relies on a Union-Find data structure, whose correctness and complexity have been previously verified [8]. It is therefore tempting to re-use as much verified code as we can, without modification.

References

1 Reynald Affeldt, Cyril Cohen, and Damien Rouhling. Formalization Techniques for Asymptotic Reasoning in Classical Analysis. *Journal of Formalized Reasoning*, 11(1):43–76, 2018.

2 Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science*, 413(1):142–159, 2012.

3 Robert Atkey. Amortised Resource Analysis with Separation Logic. *Logical Methods in Computer Science*, 7(2:17), 2011.

4 Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan. A New Approach to Incremental Cycle Detection and Related Problems. *ACM Transactions on Algorithms*, 12(2):14:1–14:22, 2016.

5 Quentin Carbonneaux, Jan Hoffmann, Thomas Reps, and Zhong Shao. Automated Resource Analysis with Coq Proof Objects. In *Computer Aided Verification (CAV)*, volume 10427 of *Lecture Notes in Computer Science*, pages 64–85. Springer, 2017.

6 Arthur Charguéraud. Characteristic Formulae for the Verification of Imperative Programs, 2013. Unpublished. `http://www.charguer aud.org/research/2013/cf/cf.pdf`.

7 Arthur Charguéraud. The CFML tool and library. `http://www.charguer aud.org/softs/cfml/`, 2019.

8 Arthur Charguéraud and François Pottier. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *Journal of Automated Reasoning*, 2017.

9 Ran Chen, Cyril Cohen, Jean-Jacques Lévy, Stephan Merz, and Laurent Théry. Formal Proofs of Tarjan's Algorithm in Why3, Coq, and Isabelle. Manuscript, 2018.

10 Ran Chen and Jean-Jacques Lévy. A Semi-automatic Proof of Strong Connectivity. In *Verified Software: Theories, Tools and Experiments*, volume 10712 of *Lecture Notes in Computer Science*, pages 49–65. Springer, 2017.

11 Nils Anders Danielsson. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *Principles of Programming Languages (POPL)*, 2008.

12 Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A Fully Verified Executable LTL Model Checker. In *Computer Aided Verification (CAV)*, volume 8044 of *Lecture Notes in Computer Science*, pages 463–478. Springer, 2013.

13 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. *Journal of Automated Reasoning*, 58(1):3–31, 2017.

14 Sumit Gulwani. SPEED: symbolic complexity bound analysis. In *Computer Aided Verification (CAV)*, volume 5643 of *Lecture Notes in Computer Science*, pages 51–62. Springer, 2009.

15 Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *Principles of Programming Languages (POPL)*, pages 127–139, 2009.

16 Armaël Guéneau. Dune pull request #1955, March 2019.

17 Armaël Guéneau, Arthur Charguéraud, and François Pottier. A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification. In *European Symposium on Programming (ESOP)*, volume 10801 of *Lecture Notes in Computer Science*, pages 533–560. Springer, 2018.

18 Maximilian P. L. Haslbeck and Tobias Nipkow. Hoare Logics for Time Bounds: A Study in Meta Theory. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 10805 of *Lecture Notes in Computer Science*, pages 155–171. Springer, 2018.

19 C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

20 Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for OCaml. In *Principles of Programming Languages (POPL)*, pages 359–373, 2017.

21 Benjamin Holland, Ganesh Ram Santhanam, Payas Awadhutkar, and Suresh Kothari. Statically-Informed Dynamic Analysis Tools to Detect Algorithmic Complexity Vulnerabilities. In *Source Code Analysis and Manipulation (SCAM)*, pages 79–84, 2016.

22 John E. Hopcroft. Computer Science: The Emergence of a Discipline. *Communications of the ACM*, 30(3):198–202, 1987.

23 Jacques-Henri Jourdan. Coq pull request #89, July 2015.

24 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

25 Peter Lammich. Verified Efficient Implementation of Gabow's Strongly Connected Component Algorithm. In *Interactive Theorem Proving (ITP)*, volume 8558 of *Lecture Notes in Computer Science*, pages 325–340. Springer, 2014.

26 Peter Lammich. Refinement to Imperative/HOL. In *Interactive Theorem Proving (ITP)*, volume 9236 of *Lecture Notes in Computer Science*, pages 253–269. Springer, 2015.

27 Peter Lammich and René Neumann. A Framework for Verifying Depth-First Search Algorithms. In *Certified Programs and Proofs (CPP)*, pages 137–146, 2015.

28 Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. PerfFuzz: Automatically generating pathological inputs. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 254–265, 2018.

29 Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. Contract-based resource verification for higher-order functions with memoization. In *Principles of Programming Languages (POPL)*, pages 330–343, 2017.

30 Jay A. McCarthy, Burke Fetscher, Max S. New, Daniel Feltey, and Robert Bruce Findler. A Coq Library for Internal Verification of Running-Times. In *Functional and Logic Programming*, volume 9613 of *Lecture Notes in Computer Science*, pages 144–162. Springer, 2016.

31 Rashmi Mudduluru and Murali Krishna Ramanathan. Efficient Flow Profiling for Detecting Performance Bugs. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 413–424, 2016.

32 Glen Mével, Jacques-Henri Jourdan, and François Pottier. Time credits and time receipts in Iris. In Luis Caires, editor, *European Symposium on Programming (ESOP)*, volume 11423 of *Lecture Notes in Computer Science*, pages 1–27. Springer, 2019.

33 Tobias Nipkow. Amortized Complexity Verified. In *Interactive Theorem Proving (ITP)*, volume 9236 of *Lecture Notes in Computer Science*, pages 310–324. Springer, 2015.

34 Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *Programming Language Design and Implementation (PLDI)*, pages 369–378, 2015.

35 Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *Principles of Programming Languages (POPL)*, pages 247–258, 2005.

36 François Pottier. Depth-First Search and Strong Connectivity in Coq. In *Journées Françaises des Langages Applicatifs (JFLA)*, 2015.

37 John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science (LICS)*, pages 55–74, 2002.

38 David M. Russinoff. A Mechanically Verified Incremental Garbage Collector. *Formal Aspects of Computing*, 6(4):359–390, 1994. doi:10.1007/BF01211305.

39 Matthieu Sozeau and Nicolas Tabareau. Universe Polymorphism in Coq. In *Interactive Theorem Proving (ITP)*, volume 8558 of *Lecture Notes in Computer Science*, pages 499–514. Springer, 2014.

40 Akhilesh Srikanth, Burak Sahin, and William R. Harris. Complexity verification using guided theorem enumeration. In *Principles of Programming Languages (POPL)*, pages 639–652, 2017.

41 Jane Street. Dune: A composable build system, 2018.

42 Robert Endre Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

43 Robert Endre Tarjan. Algorithmic Design. *Communications of the ACM*, 30(3):204–212, 1987.

44 The Coq development team. *The Coq Proof Assistant*, 2019.

45 Luca Della Toffola, Michael Pradel, and Thomas R. Gross. Synthesizing programs that expose performance bottlenecks. In *Code Generation and Optimization (CGO)*, pages 314–326, 2018.

46 Eelis van der Weegen and James McKinna. A Machine-Checked Proof of the Average-Case Complexity of Quicksort in Coq. In *Types for Proofs and Programs*, volume 5497 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2008.

47 Ben Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9):528–539, 1975.

A Formalization of Forcing and the Unprovability of the Continuum Hypothesis

Jesse Michael Han[1]
Department of Mathematics, University of Pittsburgh, PA, USA
https://www.pitt.edu/~jmh288
jessemichaelhan@gmail.com

Floris van Doorn
Department of Mathematics, University of Pittsburgh, PA, USA
http://florisvandoorn.com/
fpvdoorn@gmail.com

Abstract

We describe a formalization of forcing using Boolean-valued models in the Lean 3 theorem prover, including the fundamental theorem of forcing and a deep embedding of first-order logic with a Boolean-valued soundness theorem. As an application of our framework, we specialize our construction to the Boolean algebra of regular opens of the Cantor space $2^{\omega_2 \times \omega}$ and formally verify the failure of the continuum hypothesis in the resulting model.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Type theory; Software and its engineering → Formal methods

Keywords and phrases Interactive theorem proving, formal verification, set theory, forcing, independence proofs, continuum hypothesis, Boolean-valued models, Lean

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.19

Supplement Material https://github.com/flypitch/flypitch

Funding Both authors were supported by the Sloan Foundation, grant G-2018-10067.

Acknowledgements We thank the members of the Pitt-CMU Lean group, particularly Simon Hudon, Jeremy Avigad, Mario Carneiro, and Tom Hales for their feedback and suggestions; we are also grateful to Dana Scott and John Bell for their advice and correspondence.

Introduction

The continuum hypothesis (CH) states that there are no sets strictly larger than the countable natural numbers and strictly smaller than the uncountable real numbers. It was introduced by Cantor [7] in 1878 and was the very first problem on Hilbert's list of twenty-three outstanding problems in mathematics. Gödel [14] proved in 1938 that CH was consistent with ZFC, and later conjectured that CH is independent of ZFC, i.e. neither provable nor disprovable from the ZFC axioms. In 1963, Paul Cohen developed *forcing* [10, 11], which allowed him to prove the consistency of ¬CH, and therefore complete the independence proof. For this work, which marked the beginning of modern set theory, he was awarded a Fields medal – the only one to ever be awarded for a work in mathematical logic.

In this paper we discuss the formalization of a Boolean-valued model of set theory where the continuum hypothesis fails. The work we describe is part of the Flypitch project, which aims to formalize the independence of the continuum hypothesis. Our results mark a major milestone towards that goal.

[1] Corresponding author.

Our formalization is written in the Lean 3 theorem prover. Lean is an interactive proof assistant under active development at Microsoft Research [12, 41]. It implements the Calculus of Inductive Constructions and has a similar metatheory to Coq, adding definitional proof irrelevance, quotient types, and a noncomputable choice principle. Our formalization makes as much use of the expressiveness of Lean's dependent type theory as possible, using constructions which are impossible or unwieldy to encode in HOL, much less ZF: Lean's ordinals and cardinals, which are defined as equivalence classes of well-ordered types, live one universe level up and play a crucial role in the forcing argument; the models of set theory we construct require as input an entire universe of types; our encoding of first-order logic uses parameterized inductive types to equate type-correctness with well-formedness, eliminating the need for separate well-formedness proofs.

The method of forcing with Boolean-valued models was developed by Solovay and Scott in '65-'66 [35, 38] as a simplification of Cohen's method. Some of these simplifications were incorporated by Shoenfield [40] into a general theory of forcing using partial orders, and it is in this form that forcing is usually practiced. While both approaches have essentially the same mathematical content (see e.g. [26, 23, 28]), there are several reasons why we chose Boolean-valued models for our formalization:

- **Modularity.** The theory of forcing with Boolean-valued models cleanly splits into several components (a general theory of Boolean-valued semantics for first-order logic, a library for calculations inside complete Boolean algebras, the construction of Boolean-valued models of set theory, and the specifics of the forcing argument itself) which could be formalized in parallel and then recombined.
- **Directness.** For the purposes of an independence proof, the Boolean-valued soundness theorem eliminates the need to produce a two-valued model. This approach also bypasses any requirement for the reflection theorem/Löwenheim-Skolem theorems, Mostowski collapse, countable transitive models, or genericity considerations for filters.
- **Novelty and reusability.** As far as we were able to tell, the Boolean-valued approach to forcing has never been formalized. Furthermore, while for the purposes of an independence proof, forcing with Boolean-valued models and forcing with countable transitive models accomplish the same thing, a general library for Boolean-valued semantics of a deeply embedded logic could be used for formal verification applications outside of set theory, e.g. to formalize the Boolean-valued semantics of stochastic λ-calculus [37, 4].
- **Amenability to structural induction.** As with Coq, Lean is able to encode extremely complex objects and reason about their specifications using inductive types. However, the user must be careful to choose the encoding so that properties they wish to reason about are accessible by structural induction, which is the most natural mode of reasoning in the proof assistant. After observing (1) that the Aczel-Werner encoding of ZFC as an inductive type is essentially a special case of the recursive *name* construction from forcing (c.f. Section 3), and (2) that the automatically-generated induction principle for that inductive type *is* $\in$-induction, it is easy to see that this encoding can be modified to produce a Boolean-valued model of set theory where, again, $\in$-induction comes for free.

We briefly outline the rest of the paper. In Section 1 we outline the method of Boolean-valued models and sketch the forcing argument. Section 2 discusses a deep embedding of first-order logic, including a proof system and the Boolean-valued soundness theorem. Section 3 discusses our construction of Boolean-valued models of set theory. Section 4 describes the formalization of the forcing argument and the construction of a suitable Boolean algebra for forcing $\neg\mathsf{CH}$. Section 5 describes the formalization of some transfinite combinatorics. We conclude with a reflection on our formalization and an indication of future work.

1 Outline of the proof

ZFC is a collection of first-order sentences in the language of a single binary relation $\{\in\}$, used to axiomatize set theory. The continuum hypothesis can be written in this fashion as a first-order sentence CH. A proof of CH is a finite list of deductions starting from ZFC and ending at CH. The soundness theorem says that provability implies satisfiability, i.e. if $\mathsf{ZFC} \vdash \mathsf{CH}$, then CH interpreted in any model of ZFC is true. Taking the contrapositive, we can demonstrate the unprovability (equivalently, the consistency of the negation) of CH by exhibiting a single model where CH is not true.

A model of a first-order theory T in a language L is in particular a way of assigning true or false in a coherent way to sentences in L. Modulo provable equivalence, the sentences form a Boolean algebra and "coherent" means the assignment is a Boolean algebra homomorphism (so $\vee$ becomes join, $\forall$ becomes infimum, etc.) into $\mathbf{2} = \{\mathsf{true}, \mathsf{false}\}$. The soundness theorem ensures that this homomorphism v sends a proof $\phi \vdash \psi$ to an inequality $v(\phi) \leq v(\psi)$. $\mathbf{2}$ may be replaced by any complete Boolean algebra $\mathbb{B}$, where the top and bottom elements $\top, \bot$ take the place of true and false. It is straightforward to extend this analogy to a $\mathbb{B}$-valued semantics for first-order logic, and in this generality, the soundness theorem now says that for any such $\mathbb{B}$, if $\mathsf{ZFC} \vdash \mathsf{CH}$, then for any $\mathbb{B}$-valued structure where all the axioms of ZFC have truth-value $\top$, CH does also. Then as before, to demonstrate the consistency of the negation of CH it suffices to find just one $\mathbb{B}$ and a single $\mathbb{B}$-valued model where CH is not "true".

This is where forcing comes in. Given a universe V of set theory containing a Boolean algebra $\mathbb{B}$, one constructs in analogy to the cumulative hierarchy a new $\mathbb{B}$-valued universe $V^{\mathbb{B}}$ of set theory, where the powerset operation is replaced by taking functions into $\mathbb{B}$. Thus, the structure of $\mathbb{B}$ informs the decisions made by $V^{\mathbb{B}}$ about what subsets, hence functions, exist among the members of $V^{\mathbb{B}}$; the real challenge lies in selecting a suitable $\mathbb{B}$ and reasoning about how its structure affects the structure of $V^{\mathbb{B}}$. While $V^{\mathbb{B}}$ may vary wildly depending on the choice of $\mathbb{B}$, the original universe V always embeds into $V^{\mathbb{B}}$ via an operation $x \mapsto \check{x}$, and while the passage of x to $\check{x}$ may not always preserve its original properties, properties which are definable with only bounded quantification are preserved; in particular, $V^{\mathbb{B}}$ thinks $\check{\mathbb{N}}$ is $\mathbb{N}$.

To force the negation of the continuum hypothesis, we use the Boolean algebra $\mathbb{B} := \mathrm{RO}(2^{\aleph_2 \times \mathbb{N}})$ of regular opens of the Cantor space $2^{\aleph_2 \times \mathbb{N}}$. For each $\nu \in \aleph_2$, we associate the $\mathbb{B}$-valued characteristic function $\chi_\nu : \mathbb{N} \to \mathbb{B}$ by $n \mapsto \{f \mid f(\nu, n) = 1\}$. This induces what $V^{\mathbb{B}}$ thinks is a new subset $\widetilde{\chi_\nu} \subseteq \mathbb{N}$, called a *Cohen real*, and furthermore, simultaneously performing this construction on all $\nu \in \aleph_2$ induces what $V^{\mathbb{B}}$ thinks is a function from $\check{\aleph_2}$ to $\mathcal{P}(\mathbb{N})$. After showing that $V^{\mathbb{B}}$ thinks this function is injective, to finish the proof it suffices to show that $x \mapsto \check{x}$ preserves cardinal inequalities, as then we will have squeezed $\check{\aleph_1}$ properly between $\mathbb{N}$ and $\mathcal{P}(\mathbb{N})$. This is really the technical heart of the matter, and relies on a combinatorial property of $\mathbb{B}$ called the *countable chain condition* (CCC), the proof of which requires a detailed combinatorial analysis of the basis of the product topology for $2^{\aleph_2 \times \mathbb{N}}$; we handle this with a general result in transfinite combinatorics called the Δ-*system lemma*.

So far we have mentioned nothing about how this argument, which is wholly set-theoretic, is to be interpreted inside type theory. To do this, it was important to separate the mathematical content from the metamathematical content of the argument. While our objective is only to produce a model of ZFC satisfying certain properties, traditional presentations of forcing are careful to stay within the foundations of ZFC, emphasizing that all arguments may be performed internal to a model of ZFC, etc., and it is not immediately clear what parts of the argument use that set-theoretic foundation in an essential way and require modification in the passage to type theory. Our formalization clarifies some of these questions.

Sources. Our strategy for constructing a Boolean-valued model in which CH fails is a synthesis of the proofs in the textbooks of Bell ([5], Chapter 2) and Manin ([27], Chapter 8). For the Δ-system lemma, we follow Kunen ([26], Chapters 1 and 5).

Viewing the formalization. The code blocks in this paper were taken directly from our formalization, but for the sake of formatting and readability, we sometimes omit or modify universe levels, type ascriptions, and casts. We refer the interested reader to our repository (see Supplemental Material on page 1) which contains a guide on compiling and navigating the source files of the project.

2 First-order logic

The starting point for first-order logic is a *language* of relation and function symbols. We represent a language as a pair of $\mathbb{N}$-indexed families of types, each of which is to be thought of as the collection of relation (resp. function) symbols stratified by arity:

```
structure Language : Type (u+1) :=
(functions : ℕ → Type u) (relations : ℕ → Type u)
```

2.1 (Pre)terms, (pre)formulas

The main novelty of our implementation of first-order logic is the use of *partially applied* terms and formulas, encoded in a parameterized inductive type where the $\mathbb{N}$ parameter measures the difference between the arity and the number of applications. The benefit of this is that it is impossible to produce an ill-formed term or formula, because type-correctness is equivalent to well-formedness. This eliminates the need for separate well-formedness proofs.

Fix a language L. We define the type of **preterms** as follows:

```
inductive preterm (L : Language.{u}) : ℕ → Type u
| var {} : ∀ (k : ℕ), preterm 0 -- notation ‘&’
| func : ∀ {l : ℕ} (f : L.functions l), preterm l
| app : ∀ {l : ℕ} (t : preterm (l + 1)) (s : preterm 0), preterm l
```

We use de Bruijn indices to avoid variable shadowing. A member of `preterm n` is a partially applied term. If applied to `n` terms, it becomes a term. We define the type of well-formed terms `term L` to be `preterm L 0`.

There are other methods to define well-typed terms, for example using a nested inductive type with a constructor (which replaces the second and third constructor in our definition)

```
| app : ∀ {l : ℕ} (f : L.functions l) (ts : vector term l), term
```

Here `vector term l` is a l-tuple of terms. Lean has limited support for nested inductive types, but defining definitions by recursion on a nested inductive type is inconvenient.

The type of **preformulas** is defined similarly:

```
inductive preformula (L : Language.{u}) : ℕ → Type u
| falsum {} : preformula 0 --  notation ⊥
| equal (t₁ t₂ : term L) : preformula 0 -- notation ≃
| rel {l : ℕ} (R : L.relations l) : preformula l
| apprel {l : ℕ} (f : preformula (l + 1)) (t : term L) : preformula l
| imp (f₁ f₂ : preformula 0) : preformula 0 -- notation ⟹
| all (f : preformula 0) : preformula 0 -- notation ∀'
```

A member of `preformula n` is a partially applied formula. If applied to `n` terms, it becomes a formula. The type of well-formed formulas `formula L` is defined to be `preterm L 0`. Implication is the only primitive binary connective and universal quantification is the only primitive quantifier. Since we use classical logic, we can define the other connectives and quantifiers from these. In particular, we define negation $\sim$ `f` to be `f` $\implies \bot$ and existential quantification $\exists'$ `f` to be $\sim \forall' \sim$ `f`. Note that implication and the universal quantifier cannot be applied to performulas that are not fully applied.

We choose this definition of `preformula` to mimic `preterm`. Of course, we could define an inductive type where the constructors `rel` and `apprel` were replaced by the single constructor

```
| rel : ∀ {l : ℕ} (f : L.relations l) (ts : vector term l), formula
```

This would not even result in a nested inductive type. However, we found it more convenient to adapt operations and proofs from `preterm` to `preformula` using our definition. Using vectors results in some extra proof steps for reasoning about vectors. Our approach also results in some extra proof steps, but they are the same as the steps in the corresponding proofs for preterms.

We define the usual operations of lifting and substitution for terms and formulas. We use the notation `t` $\uparrow'$ `n # m` to mean the preterm of preformula `t` where all variables which are at least `m` are increased by `n`. `t` $\uparrow'$ `n # 0` is abbreviated to `t` $\uparrow$ `n`. The substitution `t[s // n]` is defined to be the term or formula `t` where all variables that represent the `n`-th free variable are replaced by `s`. More specifically, if an occurrence of a variable `&(n+k)` is under `k` quantifiers, then it is replaced by `s` $\uparrow$ `(n+k)`. Variables `&m` for $m > n + k$ are replaced by `&(m-1)`.

Our proof system is a natural deduction calculus, and all rules are motivated to work well with backwards-reasoning. The type of proof trees is

```
inductive prf : set (formula L) → formula L → Type u
| axm     {Γ A} (h : A ∈ Γ) : prf Γ A
| impI    {Γ} {A B} (h : prf (insert A Γ) B) : prf Γ (A ⟹ B)
| impE    {Γ} (A) {B} (h₁ : prf Γ (A ⟹ B)) (h₂ : prf Γ A) : prf Γ B
| falsumE {Γ} {A} (h : prf (insert ∼A Γ) ⊥) : prf Γ A
| allI    {Γ A} (h : prf ((λ f, f ↑ 1) '' Γ) A) : prf Γ (∀' A)
| allE₂   {Γ} A t (h : prf Γ (∀' A)) : prf Γ (A[t // 0])
| ref     (Γ t) : prf Γ (t ≃ t)
| subst₂  {Γ} (s t f) (h₁ : prf Γ (s ≃ t)) (h₂ : prf Γ (f[s // 0])) :
          prf Γ (f[t // 0])
```

In `allI` the notation `(λ f, f ↑ 1) '' Γ` means lifting all free variables in Γ by one. A term of type `prf Γ A`, denoted $\Gamma \vdash$ `A`, is a proof tree encoding a derivation of A from Γ. We also define provability as the proposition stating that a proof tree exists.

```
def provable (Γ : set (formula L)) (f : formula L) : Prop :=
nonempty (prf Γ f)
```

Our current formalization does not use proof trees in an essential way, but we defined them so that we can define manipulations on proof trees (like detour elimination) in future projects. We prove various meta-theoretic properties about provability, like weakening and the substitution theorem.

```
def weakening (H₁ : Γ ⊆ Δ) (H₂ : Γ ⊢ A) : Δ ⊢ A
def substitution (H : Γ ⊢ A) : (λ f, f[s // n]) '' Γ ⊢ A[s // n]
```

2.2 Completeness

As part of our formalization of first-order logic, we completed a verification of the Gödel completeness theorem. Although our present development of forcing did not require it, we anticipate that it will be useful later to e.g. prove the downward Löwenheim-Skolem theorem for extracting countable transitive models. Like soundness, it also serves as a proof-of-concept and stress-test of our chosen encoding of first-order logic.

For our formalization, we chose the Henkin-style approach of constructing a canonical term model. In order to perform the argument, which normally involves modifying the language "in place" to iteratively add new constant symbols, we had to adapt it to type theory. Since our languages are represented by pairs of indexed types instead of sets, we cannot really modify them in-place with new constant symbols. Instead, at each step of the construction, we must construct an entirely new language in which the previous one embeds, and in the limit we must compute a directed colimit of types instead of a union. This construction induces similar constructions on terms and formulas, and completing the argument requires reasoning with all of them. As a result of our design decisions, only a few arguments required anything more than straightforward case-analysis and structural induction. The final statement makes no restrictions on the cardinality of the language.

2.3 Boolean-valued semantics for first-order logic

A **complete Boolean algebra** is a type $\mathbb{B}$ equipped with the structure of a Boolean algebra and additionally operations Inf and Sup (which we write as $\bigsqcap$ and $\bigsqcup$) returning the infimum and supremum of an arbitrary collection of members of $\mathbb{B}$. We use $\sqcap, \sqcup$, $\implies$, $\top$, and $\bot$ to denote meet, join, material implication, and top/bottom elements. For more details on complete Boolean algebras, we refer the reader to the textbook of Halmos-Givant [13].

▶ **Definition 1.** Fix a language L and a complete Boolean algebra $\mathbb{B}$. A **$\mathbb{B}$-valued structure** is an instance of the following `structure`:

```
structure bStructure :=
(carrier : Type u)
(fun_map : ∀{n}, L.functions n → vector carrier n → carrier)
(rel_map : ∀{n}, L.relations n → vector carrier n → 𝔹)
(eq : carrier → carrier → 𝔹)
(eq_refl : ∀ x, eq x x = ⊤)
(eq_symm : ∀ x y, eq x y = eq y x)
(eq_trans : ∀{x} y {z}, eq x y ⊓ eq y z ≤ eq x z)
(fun_congr : ∀{n} (f : L.functions n) (x y : vector carrier n),
  ⨅(map2 eq x y) ≤ eq (fun_map f x) (fun_map f y))
(rel_congr : ∀{n} (R : L.relations n) (x y : vector carrier n),
  ⨅(map2 eq x y) ⊓ rel_map R x ≤ rel_map R y)
```

Above, "`⨅(map2 eq x y)`" means "the infimum of the list whose ith entry is `eq` applied to `x[i]` and `y[i]`".

Note that Boolean-valued equality is not really an equivalence relation, but "$\mathbb{B}$ thinks it is". One complication which then arises in Boolean-valued semantics is keeping track of the congruence lemmas for formulas. However, as part of the soundness theorem shows, once these extensionality proofs are provided for the basic symbols in the language, they extend by structural induction to all formulas.

2.4 The soundness theorem

A soundness theorem says that a proof tree may be replayed to produce an actual proof in the object of truth-values. When the object of truth-values is `Prop`, this says that a proof tree compiles to a proof term. When the object of truth-values is a Boolean algebra, this says that the proof tree becomes an internal implication from the interpretation of the context to the interpretation of the conclusion:

```
lemma boolean_soundness {Γ : set (formula L)} {A : formula L}
  (H : Γ ⊢ A) : ∀ M, (⨅γ ∈ Γ, M[γ]) ≤ M[A]
```

Of course, we also formalized the ordinary soundness theorem. As a result of our design decisions, the proofs of both the ordinary and Boolean-valued soundness theorems were straightforward structural inductions.

3 Constructing Boolean-valued models of set theory

Throughout this section, we fix a universe level u and a complete Boolean algebra $\mathbb{B}$: `Type u`.

In set theory (see e.g. Jech [23] or Bell [5]), Boolean-valued models are obtained by imitating the construction of the von Neumann cumulative hierarchy via a transfinite recursion where iterations of the powerset operation (taking functions into $\mathbf{2} = \{\text{true}, \text{false}\}$) are replaced by iterations of the "$\mathbb{B}$-valued powerset operation" (taking functions into $\mathbb{B}$).

Since this construction by transfinite recursion does not easily translate into type theory, our construction of Boolean-valued models of set theory is instead a variation on a well-known encoding originally due to Aczel [1, 3, 2]. This encoding was adapted by Werner [42] to encode ZFC into Coq, whose metatheory is close to that of Lean. Werner's construction was implemented in Lean's `mathlib` by Carneiro [9]. In this approach, one takes a universe of types `Type u` as the starting point and then imitates the cumulative hierarchy by constructing the inductive type

```
inductive pSet : Type (u+1)
| mk (α : Type u) (A : α → pSet) : pSet
```

The Aczel-Werner encoding is closely related to the recursive definition of *names*, which is used in forcing to construct forcing extensions:

▶ **Definition 2.** Let P be a partial order (which one thinks of as a collection of forcing conditions). A *P-name* is a collection of pairs (y, p) where y is a P-name and $p : P$.

If P consists of only one element, then a P-name is specified by essentially the same information as a member of the inductive type `pSet` above. Conversely, specializing P to an arbitrary complete Boolean algebra $\mathbb{B}$, we generalize the definition of `pSet.mk` so that elements are recursively assigned Boolean truth-values:

```
inductive bSet (𝔹 : Type u) [complete_boolean_algebra 𝔹] : Type (u+1)
| mk (α : Type u) (A : α → bSet) (B : α → 𝔹) : bSet
```

Thus `bSet 𝔹` is the type of $\mathbb{B}$-names, and will be the underlying type of our Boolean-valued model of set theory. For convenience, if `x : bSet 𝔹` and `x := ⟨α, A, B⟩`, we put `x.type := α`, `x.func := A`, `x.bval := B`.

3.1 Boolean-valued equality and membership

In `pSet`, equivalence of sets is defined by structural recursion as follows: two sets x and y are equivalent if and only if for every $w \in x$, there exists a $w' \in y$ such that w is equivalent to w', and vice-versa. Analogously, by translating quantifiers and connectives into operations on $\mathbb{B}$, Boolean-valued equality is defined in the same way:

```
def bv_eq : ∀ (x y : bSet 𝔹), 𝔹
| ⟨α, A, B⟩ ⟨α', A', B'⟩ :=
              (⨅a : α, B a ⟹ ⨆a', B' a' ⊓ bv_eq (A a) (A' a')) ⊓
                (⨅a' : α', B' a' ⟹ ⨆a, B a ⊓ bv_eq (A a) (A' a'))
```

We abbreviate `bv_eq` with the infix operator $=^{\mathsf{B}}$. With equality in place, it is easy to define membership by translating "x is a member of y if and only if there exists a w indexed by the type of y such that $x = w$." As with equality, we denote $\mathbb{B}$-valued membership by $\in^{\mathsf{B}}$.

```
def mem : bSet 𝔹 → bSet 𝔹 → 𝔹
| a ⟨α' A' B'⟩ := ⨆a', B' a' ⊓ a =ᴮ A' a'
```

3.2 Automation and metaprogramming for reasoning in $\mathbb{B}$

As stressed by Scott [36], "A main point ... is that the well-known algebraic characterizations of [complete Heyting algebras] and [complete Boolean algebras] exactly mimic the rules of deduction in the respective logics." Indeed, that is really why the Boolean-valued soundness theorem is true. One thinks of the $\leq$ symbol in an inequality of Boolean truth-values as a turnstile in a proof state: the conjuncts on the left as a list of assumptions in context, and the quantity on the right as the goal. For example, given `a b : 𝔹`, the identity $(a \Rightarrow b) \sqcap a \leq b$ could be proven by unfolding the definition of material implication, but it is really just modus ponens; similarly, given an indexed family `a : I → 𝔹`, the equivalence `(⨆i, a i ≤ b) ↔ ∀ i, a i ≤ b` is just $\exists$-elimination.

Difficulties arise when the statements to be proved become only slightly more complicated. Consider the following example, which should be "`by assumption`":

```
∀ a b c d e f g: 𝔹, (d ⊓ e) ⊓ (f ⊓ g ⊓ ((b ⊓ a) ⊓ c)) ≤ a
```

or slightly less trivially, the following example where the goal is attainable by "just applying a hypothesis to an assumption"

```
∀ a b c d : 𝔹, (a ⟹ b) ⊓ c ⊓ (d ⊓ a) ≤ b
```

There are three ways to deal with goals like these, which approximately describe the evolution of our approach. First, one can try using the basic lemmas in `mathlib`, using the simplifier to normalize expressions, and performing clever rewrites with the deduction theorem.[2] Second, one can take the LCF-style approach and expand the library of lemmas with increasingly sophisticated derived inference rules. Third, one can make the following observation:

▶ **Lemma 3** (Yoneda lemma for posets). *Let $(P, \leq)$ be a partially ordered set. Let $a\ b : P$. Then $a \leq b$ if and only if $\forall \Gamma : P, \Gamma \leq a \to \Gamma \leq b$.*

[2] The deduction theorem in a Boolean algebra says that for all a, b and c, $a \sqcap b \leq c \iff a \leq b \Rightarrow c$.

This is a consequence of the Yoneda lemma for partially ordered sets, and its proof is utterly trivial. However, one side of the equivalence is much easier for Lean to reason with. Take the example which should have been "by assumption". The following proof, in which the user navigates down the binary tree of nested ⊓s, will work:

```
example {a b c d e f g : 𝔹} : (d ⊓ e) ⊓ (f ⊓ g ⊓((b ⊓ a)⊓ c)) ≤ a :=
by {apply inf_le_right_of_le, apply inf_le_right_of_le,
    apply inf_le_left_of_le, apply inf_le_right_of_le, refl}
```

But if we use the right-hand side of Lemma 3 instead, then after some preprocessing, assumption will literally work:

```
example {a b c d e f g : 𝔹} : (d ⊓ e) ⊓ (f ⊓ g ⊓((b ⊓ a)⊓ c)) ≤ a :=
by {tidy_context, assumption}
-- `tidy_context` applies `poset_yoneda`, introduces a hypothesis `H`,
-- uses `simp` at H to convert ⊓s to ∧s, and automatically splits
/- Goal state before `assumption`:
[...]
H_right_right_left_left : Γ ≤ b,
H_right_right_left_right : Γ ≤ a
⊢ Γ ≤ a -/
```

A key feature of Lean is that it is its own metalanguage, allowing for seamless in-line definitions of custom tactics. This feature was an invaluable asset, as it allowed the rapid development of a custom tactic library for simulating natural-deduction style proofs inside 𝔹 after applying Lemma 3. Boolean-valued versions of natural deduction rules like ∨/∧-elimination, instantiation of existentials, implication introduction, and even basic automation were easy to write. The result is that the user is able to pretend, with absolute rigor, that they are simply writing proofs in first-order logic while calculations in the complete Boolean algebra are being performed under the hood.

One use-case where automation is crucial is context-specialization. For example, suppose that after preprocessing with `poset_yoneda`, the goal is `Γ ≤ (a ⟹ b)`, and one would like to "introduce the implication", adding `Γ ≤ a` to the context and reducing the goal to `Γ ≤ b`. This is impossible as stated. Rather, the deduction theorem lets us rewrite the goal to `Γ ⊓ a ≤ b`, and now we may add `Γ ⊓ a ≤ a` to the context. So we may introduce the implication after all, but at the cost of specializing the context Γ to the smaller context `Γ' := Γ ⊓ a`. But now, in order for the user to continue the pretense that they are merely doing first-order logic, this change of variables must be propagated to the rest of the assumptions which may still be of the form `Γ ≤ _` – which is extremely tedious to do by hand, but easy to automate.

3.3 The fundamental theorem of forcing

The fundamental theorem of forcing for Boolean-valued models [17] states that for any complete Boolean algebra B, V^B is a Boolean-valued model of ZFC. Since, in type theory, a type universe `Type u` takes the place of the standard universe V, the analogous statement in our setting is that for every complete Boolean algebra 𝔹, `bSet 𝔹` is a Boolean-valued model of ZFC.

Bell [5] gives an extremely detailed account of the verification of the ZFC axioms, and we faithfully followed his presentation for this part of the formalization. Most of it is routine. We describe some aspects of `bSet 𝔹` which are revealed by this verification.

Check-names.

▶ **Definition 4.** From the definitions of `pSet` and `bSet`, one immediately sees that there is a canonical map `check : pSet → bSet 𝔹`, defined by

```
def check : pSet → bSet 𝔹
| ⟨α,A⟩ := ⟨α, (λ a, check (A a)), λ a, ⊤⟩
```

We call members of the image of `check` *check-names*,[3] after the usual diacritic notation $\check{x}$ for `check (x : pSet)`. These are also known as *canonical names*, as they are the canonical representation of standard two-valued sets inside a Boolean-valued model of set theory.[4]

The axiom of infinity. In `pSet`, ω is defined to be the collection of all finite von Neumann ordinals (via induction on $\mathbb{N}$), and `(ω : bSet 𝔹)` is $\check{\omega}$. While it is easy to show $\check{\omega}$ satisfies the axiom of infinity

```
def axiom_of_infinity_spec (u : bSet 𝔹) : 𝔹 :=
  (∅∈ᴮ u) ⊓ (⨅i_x, ⨆i_y, (u.func i_x ∈ᴮ u.func i_y))
```

it can furthermore be shown to satisfy the universal property of ω, which says that ω is a subset of any set which contains $\emptyset$ and is closed under the successor operation $x \mapsto x \cup \{x\}$.

The axiom of powerset.

▶ **Definition 5.** Fix a $\mathbb{B}$-valued set `x = ⟨α, A, b⟩`. Let `χ : α → 𝔹` be a function. The subset of `x` associated to `χ` is a $\mathbb{B}$-valued set $\widetilde{\chi}$ defined as follows:

```
def set_of_indicator {x} (χ : x.type → 𝔹) := ⟨x.type, x.func, χ⟩
```

The **powerset** $\mathcal{P}(x)$ of x is defined to be the following $\mathbb{B}$-valued set, whose underlying type is the type of all functions `x.type → 𝔹`:

```
def bv_powerset (u : bSet 𝔹) : bSet 𝔹 :=
⟨u.type → 𝔹, (λ f, set_of_indicator f), (λ f, set_of_indicator f ⊆ᴮ u)⟩
```

The axiom of choice. Following Bell, we verified Zorn's lemma, which is provably equivalent over ZF to the axiom of choice. As is the case with `pSet`, establishing the axiom of choice requires the use of a choice principle from the metatheory. This was the most involved part of our verification of the fundamental theorem of forcing, and relies on the technical tool of *mixtures*, which allow sequences of $\mathbb{B}$-valued sets to be "averaged" into new ones, and the *maximum principle*, which allows existentially quantified statements to be instantiated without changing their truth-value.

The smallness of $\mathbb{B}$. We end this section by remarking that the "smallness" (or more precisely, the fact that $\mathbb{B}$ lives in the same universe of types out of which `bSet 𝔹` is being built) is essential in making `bSet 𝔹` a model of ZFC. It is required for extracting the witness needed for the maximum principle, and is also required to even define the powerset operation, because the underlying type of the powerset is the function type of all maps into $\mathbb{B}$.

---

[3] This terminology is standard, c.f. [17, 28].

[4] We were pleased to discover Lean's support for custom notation allowed us to declare the Unicode modifier character `U+030C` (ˇ) as a postfix operator for `check`.

4 Forcing

4.1 Representing Lean's ordinals inside `pSet` and `bSet`

The treatment of ordinals in `mathlib` associates a class of ordinals to every type universe, defined as isomorphism classes of well-ordered types, and includes interfaces for both well-founded and transfinite recursion. Lean's ordinals may be represented inside `pSet` by defining a map `ordinal.mk : ordinal → pSet` via transfinite recursion; it is nothing more than the von Neumann definition of ordinals. In pseudocode,

```
def ordinal.mk : ordinal → pSet
| 0 := ∅
| succ ξ := pSet.succ (ordinal.mk ξ) -- (mk ξ ∪ {mk ξ})
| is_limit ξ := ⋃ η < ξ, (ordinal.mk η)
```

Composing by `check` (Definition 4) yields a map `check ∘ ordinal.mk : ordinal → bSet` $\mathbb{B}$. (We could just as well have defined `ordinal.mk' : ordinal → bSet` $\mathbb{B}$ analogously to `ordinal.mk` without reference to `check`, such that `ordinal.mk' = check ∘ ordinal.mk`; the point is that there is a link between the metatheory's notion of size and order with that of the forcing extension.)

Cardinals in Lean are defined separately from ordinals as bijective equivalence classes of types, but are canonically represented by ordinals which are not bijective with any predecessor. We let `aleph : ordinal → ordinal` index these representatives. For the rest of this section, unadorned alephs (e.g. "$\aleph_2$") will mean either an ordinal of the form `aleph ξ` or a choice of representative from the isomorphism class of well-ordered types, and checked alephs (e.g. "$\check{\aleph_2}$") will mean the `check ∘ ordinal.mk` of that ordinal.

4.2 The Cohen poset and the regular open algebra

Forcing with partial orders and forcing with complete Boolean algebras are related by the fact that every poset of forcing conditions can be embedded into a complete Boolean algebra as a dense suborder. This will be the case for our forcing argument: our Boolean algebra is the algebra of regular opens on $2^{\aleph_2 \times \mathbb{N}}$ (we identify this space with the subsets of $\aleph_2 \times \mathbb{N}$), and the poset of forcing condition embeds in this Boolean algebra as a dense suborder.

▶ **Definition 6.** The **Cohen poset** for adding $\aleph_2$-many Cohen reals is the collection of all finite partial functions $\aleph_2 \times \mathbb{N} \to \mathbf{2}$, ordered by reverse inclusion.

In the formalization, the Cohen poset is represented as a `structure` with three fields:

```
structure 𝒞 : Type :=
  (ins : finset (ℵ₂.type × ℕ))
  (out : finset (ℵ₂.type × ℕ))
  (H : ins ∩ out = ∅)
```

That is, we identify a finite partial function f with the triple $\langle$`f.ins, f.out, f.H`$\rangle$, where `f.ins` is the preimage of $\{1\}$, `f.out` is the preimage of $\{0\}$, and `f.H` ensures well-definedness. While f is usually defined as a finite partial function, we found that in practice f is really only needed to give a finite partial specification of a subset of $\aleph_2 \times \mathbb{N}$ (i.e. a finite set `f.ins` which *must* be in the subset, and a finite set `f.out` which *must not* be in the subset), and chose this representation to make that information immediately accessible.

▶ **Definition 7.** Let X be a topological space, and for any open set U, let $U^\perp$ denote the complement of the closure of U. The **regular open algebra** of a topological space X, written $\mathrm{RO}(X)$, is the collection of all open sets U such that $U = (U^\perp)^\perp$, equipped with the structure of a complete Boolean algebra, with $x \sqcap y := x \cap y$, $x \sqcup y := ((x \cup y)^\perp)^\perp$, $\neg x := x^\perp$, and $\bigsqcup x_i := ((\bigcup x_i)^\perp)^\perp$.

The Boolean algebra which we will use for forcing $\neg\mathsf{CH}$ is $\mathrm{RO}(2^{\aleph_2 \times \mathbb{N}})$. Unless stated otherwise, for the rest of this section, we put $\mathbb{B} := \mathrm{RO}(2^{\aleph_2 \times \mathbb{N}})$.

▶ **Definition 8.** We define the **canonical embedding** of the Cohen poset into $\mathbb{B}$ as follows:

```
def ι : 𝒞 → 𝔹 := λ p, {S | p.ins ⊆ S ∧ p.out ⊆ - S}
```

That is, we send each `c : 𝒞` to all the subsets which satisfy the specification given by `c`. This is a clopen set, hence regular. Crucially, this embedding is *dense*:

```
lemma 𝒞_dense {b : 𝔹} (H : ⊥ < b) : ∃ p : 𝒞, ι p ≤ b
```

Recalling that $\leq$ in $\mathbb{B}$ is subset-inclusion, we see that this is essentially because the image of $\iota : \mathcal{C} \to \mathbb{B}$ *is* the standard basis for the product topology. Our chosen encoding of the Cohen poset also made it easier to perform this identification when formalizing this proof.

4.3 Adding $\aleph_2$-many distinct Cohen reals

As we saw in Definition 5, for any $\mathbb{B}$-valued set x, characteristic functions into $\mathbb{B}$ from the underlying type of x determine $\mathbb{B}$-valued subsets of x. While the ingredients $\aleph_2$ and $\mathbb{N}$ for $\mathbb{B}$ are types and thus external to `bSet 𝔹`, they are represented nonetheless inside `bSet 𝔹` by their check-names $\check{\aleph_2}$ and $\check{\mathbb{N}}$, and in fact $\aleph_2$ *is* $\check{\aleph_2}$ `.type` and $\mathbb{N}$ *is* $\check{\mathbb{N}}$`.type`. Given our specific choice of $\mathbb{B}$, this will allow us to construct an $\aleph_2$-indexed family of distinct subsets of $\check{\mathbb{N}}$, which we can then convert into an injective function from $\check{\aleph_2}$ to $\mathcal{P}(\mathbb{N})$, *inside* `bSet 𝔹`.

▶ **Definition 9.** Let $\nu : \aleph_2$. For any $n : \mathbb{N}$, the collection of all subsets of $\aleph_2 \times \mathbb{N}$ which contain (ν, n) is a regular open of $2^{\aleph_2 \times \mathbb{N}}$, called the **principal open** $\mathbf{P}_{(\nu,n)}$ over (ν, n).

▶ **Definition 10.** Let $\nu : \aleph_2$. We associate to ν the $\mathbb{B}$-valued characteristic function $\chi_\nu : \mathbb{N} \to \mathbb{B}$ defined by $\chi_\nu(n) := \mathbf{P}_{(\nu,n)}$. In light of our previous observations, we see that each χ_ν induces a new $\mathbb{B}$-valued subset $\widetilde{\chi_\nu} \subseteq \check{\mathbb{N}}$. We call $\widetilde{\chi_\nu}$ a **Cohen real**.

This gives us an $\aleph_2$-indexed family of Cohen reals. Converting this data into an injective function from $\check{\aleph_2}$ to $\mathcal{P}(\mathbb{N})$ inside `bSet 𝔹` requires some care. One must check that $\nu \mapsto \widetilde{\chi_\nu}$ is externally injective, and this is where the characterization of the Cohen poset as a dense subset of $\mathbb{B}$ (and moving back and forth between this representation and the definition as finite partial functions) comes in. Furthermore, one has to develop machinery similar to that for the powerset operation to convert an external injective function `x.type → bSet 𝔹` to a $\mathbb{B}$-valued set which `bSet 𝔹` thinks is a injective function, while maintaining conditions on the intended codomain. Our custom tactics and automation for reasoning inside $\mathbb{B}$ made this latter task significantly easier than it would have been otherwise. We refer the interested reader to our formalization for details.

4.4 Preservation of cardinal inequalities

So far, we have shown for $\mathbb{B} = \mathrm{RO}(2^{\aleph_2 \times \mathbb{N}})$ that `bSet 𝔹` thinks $\check{\aleph_2}$ is smaller than $\mathcal{P}(\check{\mathbb{N}})$. Although Lean believes there is a strict inequality of cardinals $\aleph_0 < \aleph_1 < \aleph_2$, in general we

can only deduce that their representations inside `bSet` $\mathbb{B}$ are subsets of each other: $\top \leq \aleph_0^{\check{}} \subseteq^{\mathbb{B}} \aleph_1^{\check{}} \subseteq^{\mathbb{B}} \aleph_2^{\check{}}$. To finish negating CH, it suffices to show that `bSet` $\mathbb{B}$ thinks $\aleph_0^{\check{}}$ is strictly smaller than $\aleph_1^{\check{}}$, and that `bSet` $\mathbb{B}$ thinks $\aleph_1^{\check{}}$ is a strictly smaller than $\aleph_2^{\check{}}$. That is, for cardinals κ, we want that the passage from κ to $\kappa^{\check{}}$ to preserve cardinal inequalities.

▶ **Definition 11.** For our purposes, "X is strictly smaller than Y" means "there exists no function `f` such that for every `y` $\in$ `Y`, there exists an `x` $\in$ `X` such that `(x,y)` $\in$ `f`". Thus, "`X` is strictly smaller than `Y`" translates to the Boolean truth-value

```
-(⨆f, (is_func f) ⊓ ⨅y, y ∈ᴮ Y ⟹ ⨆x, x ∈ᴮ X ⊓ (x, y) ∈ᴮ f).
```

We abbreviate this with "$X \prec Y$".

The condition on an arbitrary $\mathbb{B}$ which ensures the preservation of cardinal inequalities is the *countable chain condition*.

▶ **Definition 12.** We say that $\mathbb{B}$ has the **countable chain condition** (CCC) if every antichain $\mathcal{A} : I \to \mathbb{B}$ (i.e. an indexed collection of elements $\mathcal{A} := \{a_i\}$ such that whenever $i \neq j, a_i \sqcap a_j = \bot$) has a countable image.

We sketch the argument that CCC implies the preservation of cardinal inequalities. The proof is by contraposition. Let κ_1 and κ_2 be cardinals such that $\kappa_1 < \kappa_2$, and suppose that $\kappa_1^{\check{}}$ is not strictly smaller than $\kappa_2^{\check{}}$. Then there exists some `f : bSet` $\mathbb{B}$ and some $\Gamma > \bot$ such that $\Gamma \leq$ `(is_func f)` $\sqcap$ `⨅y, y` $\in^{\mathbb{B}}$ $\kappa_1^{\check{}} \implies$ `⨆x, x` $\in^{\mathbb{B}}$ $\kappa_2^{\check{}}$ $\sqcap$ `(x,y)` $\in^{\mathbb{B}}$ `f`. Then one can show:

```
lemma AE_of_check_larger_than_check :
∀ β < κ₂, ∃ η < κ₁, ⊥ < (is_func f) ⊓ (η̌, β̌ ) ∈ᴮ f
```

The name of this lemma emphasizes that what has happened here is that, given this f and the assumption that it satisfes some $\forall$-$\exists$ formula inside `bSet` $\mathbb{B}$, we are able to extract, by virtue of $\kappa_1^{\check{}}$ and $\kappa_2^{\check{}}$ being check-names, a $\forall$-$\exists$ statement in the *metatheory*. Using Lean's choice principle, we can then convert this $\forall$-$\exists$ statement into a function $g : \kappa_2 \to \kappa_1$, such that for every β, $\bot <$ `(is_func f)` $\sqcap$ `(g(`β`)̌ ,` β `̌ )` $\in^{\mathbb{B}}$ `f`. Since $\kappa_2 > \kappa_1$, it follows from the infinite pigeonhole principle that there exists some $\eta < \kappa_1$ such that the $g^{-1}(\{\eta\})$ is uncountable. Define $\mathcal{A} : g^{-1}(\{\eta\}) \to \mathbb{B}$ by $\mathcal{A}(\beta) :=$ `(is_func f)` $\sqcap$ `(g(`β`)̌ ,` β `̌ )` $\in^{\mathbb{B}}$ `f`. This is an uncountable antichain because if $\beta_1 \neq \beta_2$, then the well-definedness part of `is_func f` ensures that, since $g(\beta_1) = g(\beta_2)$, the truth-value $\beta_1^{\check{}} = f(g(\beta_1)) \neq^{\mathbb{B}} f(g(\beta_2)) = \beta_2^{\check{}}$ is $\bot$.

Thus, conditional on showing that $\mathbb{B} = \mathrm{RO}(2^{\aleph_2 \times \mathbb{N}})$ has the CCC, we now have that cardinal inequalities are preserved in `bSet` $\mathbb{B}$. Combining this with the injection $\aleph_2^{\check{}} \preceq \mathcal{P}(\mathbb{N})$, we obtain:

```
theorem neg_CH : ⊤ = (ℕ ≺ (ℵ₁)̌ ⊓ (ℵ₁)̌ ≺ (ℵ₂)̌ ⊓ (ℵ₂)̌ ≼ 𝒫(ℕ))
```

The arguments sketched in Subsection 4.3 and Subsection 4.4 form the heart of the forcing argument. Their proofs involve taking objects in `Type u` and `bSet` $\mathbb{B}$, constructing corresponding objects on the other side, and reasoning about them in ordinary and $\mathbb{B}$-valued logic simultaneously to determine cardinalities in `bSet` $\mathbb{B}$. We have omitted many details from our discussion, but of course, all the proofs have been formally verified.

4.5 The unprovability of CH

We conclude this section by briefly describing how the previous results may be converted into a formal proof of the unprovability of CH. We work in a conservative expansion ZFC′ of ZFC with an expanded language $L_{\mathsf{ZFC}'}$ with symbols for pairing, union, powerset, and ω. We define ZFC′ to be precisely the ZFC axioms which were verified in the fundamental theorem of forcing, along with specifications for the new function symbols. CH can then be written as a deeply-embedded $L_{\mathsf{ZFC}'}$ sentence (note the use of de Bruijn indices for variables)

```
def CH : sentence L_ZFC′ := ¬ ∃′ ∃′ (ω ≺ &1) ⊓ (&1 ≺ &0) ⊓ (&0 ≼ 𝒫(ω))
```

where $\prec$ and $\preceq$ are abbreviations with the same meaning as in the previous section. Then proving `bSet 𝔹 ⊨ ZFC′ + ¬CH` is a straightforward matter of checking that sentences are interpreted correctly as Boolean truth values which we have already proved to be $\top$. Applying the contrapositive of the Boolean-valued soundness theorem yields the result.

5 Transfinite combinatorics and the countable chain condition

What remains now is to prove that $\mathrm{RO}(2^{\aleph_2 \times \mathbb{N}})$ has the CCC. There are several ways forward; we chose a very general proof using the Δ-system lemma to show more generally that the product of topological spaces satisfies the CCC if every finite subproduct does. Our proof follows Kunen [26].

5.1 The Δ-system lemma

▶ **Definition 13.** A family $(A_i)_i$ of sets is called a **Δ-system** (or a **sunflower** or **quasi-disjoint**) if there is a set r, called the **root** such that whenever $i \neq j$ we have $A_i \cap A_j = r$.

```
def is_delta_system {α ι : Type*} (A : ι → set α) :=
∃(root : set α), ∀{{x y}}, x ≠ y → A x ∩ A y = root
```

The Δ-system lemma states that if we have an uncountable family of finite sets, there is an uncounbtable subfamily which forms a Δ-system. In Lean this is formulated as follows. (`restrict A t` is the restriction of the collection `A` to `t`).

```
theorem delta_system_lemma_uncountable {α ι : Type*}
  (A : ι → set α) (h : cardinal.omega < mk ι)
  (h2A : ∀i, finite (A i)) : ∃(t : set ι),
  cardinal.omega < mk t ∧ is_delta_system (restrict A t)
```

This theorem follows from the following more general statement, taking $\kappa = \aleph_0$ and $\theta = \aleph_1$ (for cardinal numbers the operation `c ^< κ` or $c^{<\kappa}$ is the supremum of c^ρ for $\rho < \kappa$).

```
theorem delta_system_lemma {α ι : Type u} {κ θ : cardinal}
  (hκ : cardinal.omega ≤ κ) (hκθ : κ < θ) (hθ : is_regular θ)
  (hθ_le : ∀(c < θ), c ^< κ < θ) (A : ι → set α)
  (hA : θ ≤ mk ι) (h2A : ∀i, mk (A i) < κ) :
  ∃(t : set ι), mk t = θ ∧ is_delta_system (restrict A t)
```

We omit the proof, referring the interested reader to [26] or the formalization.

5.2 $\mathrm{RO}(2^{\aleph_2 \times \mathbb{N}})$ has the countable chain condition

▶ **Definition 14.** We say that a topological space X satisfies the countable chain condition if every family of pairwise disjoint open sets is countable.

We first give a sufficient condition for a product of topological spaces to satisfy the countable chain condition.

▶ **Theorem 15.** *If we have a family $(X_i)_{i\in I}$ of topological spaces, then $\prod_{i\in I} X_i$ has the countable chain condition if for every finite $J \subseteq I$ the product $\prod_{i\in J} X_i$ has the countable chain condition.*

Proof. For the proof, suppose we had an uncountable family of pairwise disjoint open subsets U_k of $\prod_{i\in I} X_i$. By shrinking U_k, we may assume that each U_k is a basic open set of the form $\prod_{i\in F_k} U_{k,i} \times \prod_{i\notin F_k} X_i$ for some finite set $F_k \subseteq I$ and $U_{k,i} \neq X_i$ open in X_i. Now the $(F_k)_k$ form a uncountable family of finite sets, so by the Δ-system lemma we know that there is an uncountable family K of indices such that $(F_k)_{k\in K}$ forms a Δ-system with root J. Now we can take the projections $\pi(U_k)$ onto $\prod_{i\in J} X_i$ for $k \in K$. We can show this forms an uncountable disjoint family of opens in $\prod_{i\in J} X_i$, contradicting the assumption. ◀

With this, the rest of the proof that $\mathbb{B} = \mathrm{RO}(2^{\aleph_2 \times \mathbb{N}})$ has the CCC is easy: since every finite product 2^J is a finite topological space, and so satisfies the CCC, it follows that the space $2^{\aleph_2 \times \mathbb{N}}$ satisfies the CCC. Also, if a topological space X satisfies the CCC then the algebra of regular opens satisfies the CCC, since every antichain of regular opens forms a family of disjoint open sets. Thus, we have shown:

```
theorem 𝔹_CCC : CCC (regular_opens (set(ℵ₂.type × ℕ)))
```

6 Related work

First-order logic, soundness, and completeness. There are many existing formalizations of first-order logic. Shankar [39] used a deep embedding of first-order logic to formalize incompleteness theorems. Harrison gives a deeply-embedded implementation of first-order logic in HOL Light [18] and a proof-search style account of the completeness theorem in [19]. Margetson [33] and Schlichtkrull [34] use the same argument for the completeness theorem in Isabelle/HOL, while Berghofer [6] (in Isabelle) and Ilik [22] (in Coq) use canonical term models.

Set theory and forcing. Set theory is a common target for formalization. Notably, a large body of formalized set theory has been completed in Isabelle/ZF, led by Paulson and his collaborators [32, 29, 30]. Most relevantly, this includes a formalization of the relative consistency of the axiom of choice with ZF [31]. Building on this, Gunther, Pagano, and Terraf have begun formalizing the basic ingredients of forcing [15, 16], taking the more conventional approach of generic extensions of countable transitive models.

Our tactic library for Boolean-valued logic was inspired by work of Hudon [21] on Unit-B, using similar techniques to embed a proof language for temporal logic [20]. It was pointed out to the authors that a trick similar to Lemma 3 had also been successfully applied in the Metamath library [8].

The work we have described in this paper relies heavily on Lean's `mathlib`. In particular, the extensive `set_theory` and `ordinal` libraries contained nearly everything we needed (including a treatment of cofinalities for the Δ-system lemma), with missing parts easily accessible through existing lemmas. These libraries were originally developed by Carneiro [9], in part to show that Lean proves the existence of infinitely many inaccessible cardinals.

7 Conclusions and future work

Reflections on the proof

As our formalization has shown, for the purposes of a consistency proof, one can perform forcing entirely outside of the set-theoretic foundations in which forcing is usually presented. There is no need to work inside an ambient model of set theory, or to even have a ground model of set theory over which one constructs a forcing extension. Instead, the recursive *name* construction applied to a universe of types is key. The type universe, with its classical two-valued logic and its own notion of ordinals, takes the place of the standard universe of sets. These external ordinals are then represented in the internal ordinals of the forcing extension by indexing the construction of von Neumann ordinals. With a clever choice of forcing conditions $\mathbb{B}$, this representation of ordinals will preserve cardinal inequalities and force an uncountable set beneath $\mathcal{P}(\mathbb{N})$.

In particular, `pSet`, being only another special case of the construction which produces `bSet` $\mathbb{B}$, is no longer a prerequisite for working with `bSet` $\mathbb{B}$, but merely a convenient tool for organizing the check-names – this is the only role it played in the proof. The check-names themselves were actually not necessary either: as we remarked, the canonical map `ordinal` $\to$ `bSet` $\mathbb{B}$ can be defined without reference to them. However, since in all of our sources, `pSet` additionally played the role of the universe of types, and an interface for it was readily available in `mathlib`, we started our formalization by following the usual arguments, implementing these simplifications as we became aware of them.

Lessons learned

- Originally, we thought set-theoretic arguments involving transfinite/ordinal induction, which are ubiquitous, would be difficult to implement. In practice, Lean's tools for well-founded recursion and the comprehensive treatment of ordinals in `mathlib` made the implementation of such arguments painless.
- Definitions and lemmas should be stated as generally as possible. This maximizes reusability, minimizes redundancy, and by exposing only the information required to complete the proof, improves the performance of automation.
- One should invest early in domain-specific automation. The formalization of the fundamental theorem was completed using only the first two strategies outlined in Subsection 3.2; the calculations, while tedious, were recorded in our sources and it seemed easier to follow them. If we had followed through on the observations around Lemma 3 and developed the custom tactic library earlier, we would have saved a significant amount of time.

Towards a formal proof of the independence of the continuum hypothesis

The work we have described in this paper was undertaken as part of the Flypitch project, which aims to produce a formal proof of the independence of the continuum hypothesis. As such, the obvious next goal is a formalization of the consistency of CH. Although it would be possible to do this using Boolean-valued models, we intend to develop the infrastructure necessary to support a proof by forcing with generic extensions, as well as Gödel's original proof by way of analyzing the constructible universe L.

Our work includes a formal proof of the unprovability of a version of CH from a version of the ZFC axioms in a conservative extension of the language of ZFC, but verifying this after completing the forcing argument (as in Subsection 4.5) is easy. What is more interesting is formalizing the equivalence of various common formulations of ZFC and CH, so that a

skeptical user may verify that their preferred version of CH is unprovable from their preferred version of ZFC. This would require formalizations of the conservativity of commonly-used extensions of ZFC, and of the equivalence of the various ways to say that one set is strictly smaller than another. The proof of the completeness theorem already required formalizing nontrivial conservativity statements, which shows that our framework is well-equipped to support such results.

Although the stated goal of our project is to achieve a formal proof of the independence of the continuum hypothesis, we also intend to develop reusable libraries for set theory and mathematical logic. We have completed a formalization of forcing, but are nowhere near completing a library which a set theorist could use to verify their research. Just as, more than 50 years ago, Cohen's proof marked the beginning of modern research in set theory, a formal proof of the independence of the continuum hypothesis will only mark the beginning of an integration of formal methods into modern research in set theory. This will require robust interfaces for handling the diverse range of forcing arguments and for reasoning about the consistency strengths of various extensions of ZFC, so that – to paraphrase Kanamori [24, 25] – deeply-embedded notions of truth and relative consistency become matters of routine manipulation as in algebra. Our work demonstrates that such tasks are well within the scope of modern interactive theorem provers.

References

1 Peter Aczel. The type theoretic interpretation of constructive set theory. In *Logic Colloquium*, volume 77, pages 55–66, 1978.

2 Peter Aczel. The type theoretic interpretation of constructive set theory: choice principles. In *Studies in Logic and the Foundations of Mathematics*, volume 110, pages 1–40. Elsevier, 1982.

3 Peter Aczel. The type theoretic interpretation of constructive set theory: inductive definitions. In *Studies in Logic and the Foundations of Mathematics*, volume 114, pages 17–49. Elsevier, 1986.

4 Giorgio Bacci, Robert Furber, Dexter Kozen, Radu Mardare, Prakash Panangaden, and Dana Scott. Boolean-valued semantics for the stochastic λ-calculus. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 669–678. ACM, 2018.

5 John L Bell. *Set theory: Boolean-valued models and independence proofs*, volume 47. Oxford University Press, 2011.

6 Stefan Berghofer. First-Order Logic According to Fitting. *Archive of Formal Proofs*, August 2007. , Formal proof development. URL: `http://isa-afp.org/entries/FOL-Fitting.html`.

7 Georg Cantor. Ein Beitrag zur Mannigfaltigkeitslehre. *Journal für die reine und angewandte Mathematik*, 84:242–258, 1878.

8 Mario Carneiro. Natural deduction in the Metamath Proof Explorer. `http://us.metamath.org/mpeuni/mmnatded.html`, 2014. Slides (`http://us.metamath.org/ocat/natded.pdf`).

9 Mario Carneiro. The type theory of Lean. In preparation (`https://github.com/digama0/lean-type-theory/releases`), 2019.

10 Paul J Cohen. The independence of the continuum hypothesis. *Proceedings of the National Academy of Sciences*, 50(6):1143–1148, 1964.

11 Paul J Cohen. The independence of the continuum hypothesis, II. *Proceedings of the National Academy of Sciences*, 51(1):105, 1964.

12 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, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015. `doi:10.1007/978-3-319-21401-6_26`.

13 Steven Givant and Paul Halmos. *Introduction to Boolean algebras*. Springer Science & Business Media, 2008.

14 Kurt Gödel. The consistency of the axiom of choice and of the generalized continuum-hypothesis. *Proceedings of the National Academy of Sciences*, 24(12):556–557, 1938.

15 Emmanuel Gunther, Miguel Pagano, and Pedro Sánchez Terraf. First steps towards a formalization of Forcing. *CoRR*, abs/1807.05174, 2018. arXiv:1807.05174.

16 Emmanuel Gunther, Miguel Pagano, and Pedro Sánchez Terraf. Mechanization of Separation in Generic Extensions. *CoRR*, abs/1901.03313, 2019. arXiv:1901.03313.

17 Joel David Hamkins and Daniel Evan Seabold. Well-founded Boolean ultrapowers as large cardinal embeddings. *arXiv preprint arXiv:1206.6075*, 2012.

18 John Harrison. Formalizing Basic First Order Model Theory. In Jim Grundy and Malcolm C. Newey, editors, *Theorem Proving in Higher Order Logics, 11th International Conference, TPHOLs'98, Canberra, Australia, September 27 - October 1, 1998, Proceedings*, volume 1479 of *Lecture Notes in Computer Science*, pages 153–170. Springer, 1998. doi:10.1007/BFb0055135.

19 John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

20 Simon Hudon. Temporal logic in Unit-B, 2018. URL: https://github.com/unitb/temporal-logic.

21 Simon Hudon, Thai Son Hoang, and Jonathan S. Ostroff. The Unit-B method: refinement guided by progress concerns. *Software & Systems Modeling*, 15:1091–1116, 2015.

22 Danko Ilik. *Constructive completeness proofs and delimited control*. PhD thesis, Ecole Polytechnique X, 2010.

23 Thomas Jech. *Set theory*. Springer Science & Business Media, 2013.

24 Akihiro Kanamori. The mathematical development of set theory from Cantor to Cohen. *Bulletin of Symbolic Logic*, 2(1):1–71, 1996.

25 Akihiro Kanamori. *The Higher Infinite: Large Cardinals in Set Theory from their beginnings*. Springer Science & Business Media, 2008.

26 Kenneth Kunen. *Set theory*, volume 102 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Co., Amsterdam-New York, 1980.

27 Yu I Manin. *A course in mathematical logic for mathematicians*, volume 53. Springer Science & Business Media, 2009.

28 Justin Tatch Moore. The method of forcing. *arXiv preprint arXiv:1902.03235*, 2019.

29 Lawrence C. Paulson. Set Theory for Verification: I. From foundations to functions. *J. Autom. Reasoning*, 11(3):353–389, 1993. doi:10.1007/BF00881873.

30 Lawrence C. Paulson. The Reflection Theorem: A Study in Meta-theoretic Reasoning. In Andrei Voronkov, editor, *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27-30, 2002, Proceedings*, volume 2392 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2002. doi:10.1007/3-540-45620-1_31.

31 Lawrence C. Paulson. The Relative Consistency of the Axiom of Choice - Mechanized Using Isabelle/ZF. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings*, volume 5028 of *Lecture Notes in Computer Science*, pages 486–490. Springer, 2008. doi:10.1007/978-3-540-69407-6_52.

32 Lawrence C. Paulson and Krzysztof Grabczewski. Mechanizing Set Theory. *J. Autom. Reasoning*, 17(3):291–323, 1996. doi:10.1007/BF00283132.

33 Tom Ridge and James Margetson. A Mechanically Verified, Sound and Complete Theorem Prover for First Order Logic. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 294–309. Springer, 2005. doi:10.1007/11541868_19.

34 Anders Schlichtkrull. *Formalization of logic in the Isabelle proof assistant*. PhD thesis, Technical University of Denmark, 2018.

35 Dana Scott. A Proof of the Independence of the Continuum Hypothesis. *Theory of Computing Systems*, 1(2):89–111, 1967.

36 Dana Scott. The Algebraic Intepretation of Quantifiers: intuitionistic and classical. *Andrzej Mostowski and Foundational Studies*, pages 289–312, 2008.

37 Dana Scott. Stochastic λ-calculi. *Journal of Applied Logic*, 12(3):369–376, 2014.

38 Dana Scott and Robert Solovay. Boolean algebras and forcing. Unpublished manuscript, 1967.

39 Natarajan Shankar. *Metamathematics, machines and Gödel's proof*, volume 38. Cambridge University Press, 1997.

40 Joseph R Shoenfield. Unramified forcing. In *Axiomatic set theory*, volume 13, pages 357–381. AMS Providence, RI, 1971.

41 Sebastian Ullrich. Lean 4: A Guided Preview. `https://leanprover.github.io/talks/vu2019.pdf`. Slides.

42 Benjamin Werner. Sets in types, types in sets. In *International Symposium on Theoretical Aspects of Computer Software*, pages 530–546. Springer, 1997.

Refinement with Time – Refining the Run-Time of Algorithms in Isabelle/HOL

Maximilian P. L. Haslbeck
Technische Universität München, Germany

Peter Lammich
The University of Manchester, England

Abstract

Separation Logic with Time Credits is a well established method to formally verify the correctness and run-time of algorithms, which has been applied to various medium-sized use-cases. Refinement is a technique in program verification that makes software projects of larger scale manageable.

Combining these two techniques for the first time, we present a methodology for verifying the functional correctness and the run-time analysis of algorithms in a modular way. We use it to verify Kruskal's minimum spanning tree algorithm and the Edmonds–Karp algorithm for network flow.

An adaptation of the Isabelle Refinement Framework [15] enables us to specify the functional result and the run-time behaviour of abstract algorithms which can be refined to more concrete algorithms. From these, executable imperative code can be synthesized by an extension of the Sepref tool [11], preserving correctness and the run-time bounds of the abstract algorithm.

2012 ACM Subject Classification Theory of computation → Program verification; Theory of computation → Separation logic; Theory of computation → Logic and verification

Keywords and phrases Isabelle, Time Complexity Analysis, Separation Logic, Program Verification, Refinement, Run Time, Algorithms

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.20

Supplement Material The formalization described in this paper is available at `https://www21.in.tum.de/~haslbema/Sepreftime`.

Funding *Maximilian P. L. Haslbeck*: DFG Grant NI 491/16-1
Peter Lammich: DFG Grant LA 3292/1 "Verifizierte Model Checker" and VeTSS grant "Formal Verification of Information Flow Security for Relational Databases"

Acknowledgements We want to thank Simon Wimmer and Armaël Guéneau, as well as the anonymous reviewers for useful suggestions to improve the paper.

1 Introduction

Recently the literature has seen various interactive verification efforts for run-time analysis of efficient algorithms and data structures: Charguéraud et al. [4] verify the union-find data structure, Zhan et al. [17] formalize amongst others the median of medians selection algorithm, Karatsuba's algorithm and splay trees, and most recently Guéneau et al. [8] verify a state-of-the-art incremental cycle detection algorithm.

While the largest of these developments fits on one page (Figure 1 in [8]) more ambitious projects have been tackled when only functional correctness is concerned: Esparza et al. [5] formalized a LTL-model checker, Fleury et al. [6] verified a SAT-solver, Wimmer et al. [16] formalized a timed automaton model checker, various graph algorithms have been verified [10, 13]. The list is growing. One key ingredient to manage the complexity of larger algorithm developments is to use refinement. It allows to separate reasoning about the abstract algorithmic idea from reasoning about implementation details. In the Isabelle world, the Isabelle Refinement Framework [15] can be used to express abstract algorithms and to

use step-wise refinement to form concrete algorithms. As a last step the Sepref tool [11] can be used to synthesize efficient executable imperative code while preserving correctness. Target languages for that tool are hybrid languages such as SML, Scala and more recently the purely imperative language LLVM [12]. Such verification efforts result in executable algorithms that are often competitive with real world implementations within one order of magnitude. However, only functional correctness is ensured and not run-time bounds.

This paper brings together run-time analysis and refinement. By extending the refinement approach to also reason about the run-time of algorithms in a modular and scalable way, we lay the ground for an additional run-time analysis of larger algorithms.

Our vision is to specify abstract algorithms and their run-time in terms of abstract operations with time bounds – say Edmonds–Karp algorithm uses at most $E * V$ find-augmenting-path operations. When we then refine an operation like find-augmenting-path to a more concrete BFS algorithm involving operations such as set membership test and map lookup, we can also refine the abstract compound algorithm to use the more refined operations. Just as for plain refinement we separate abstract run-time arguments from reasoning about run-times of concrete data structures. As a last step we synthesize executable imperative code which refines the abstract algorithm and thus obeys both the high-level correctness theorem and the run-time bound. This synthesis step is only successful when meaningful time bounds have been specified. For abstract programs with absurd run-time bounds it will just not be possible to synthesize real programs.

The main contributions of this paper are:

- We present a theory for refinement with time by creating NREST, the non-determinism monad with time, and tools for reasoning about programs in that monad (Section 2).
- We extend the Sepref Tool (Section 3.2) to synthesize executable imperative code in Imperative/HOL with time (Section 3.1) supporting imperative and amortized data structures seamlessly.
- We enable modular development of algorithms by providing a library of efficient amortized data structures and reusable algorithms with run-time guarantees (Section 4).
- We show the applicability of our approach to larger algorithm developments by use-cases such as Edmonds–Karp and Kruskal's algorithm (Section 5).

2 Non-determinism Monad With Time

In this section we introduce NREST, the timed non-determinism monad. It allows specifying the result and time consumption of programs. As this is an extension of the NRES monad of the Isabelle Refinement Framework, we follow Lammich [11] in some of our explanations.

2.1 Timed Non-determinism Monad

We want to specify the result of a computation together with its worst case execution time. We design the monad to permit three monadic effects: First, we allow non-deterministic selection from a set of computation results. This is a common technique in program refinement, used to hide implementation details of abstract algorithms. Second, we support failure in order to model non-termination and assertions. Last, we model upper bounds on the run time for each possible result.[1] A program in the timed non-determinism monad is defined over the type α *NREST*:

[1] Alternatively, one could use a set of pairs of result and time constraint. However, this would not be *fully abstract* wrt. upper bounds, in the sense that $\{(r,1),(r,3)\}$ would be equivalent to $\{(r,3)\}$.

α *NREST* = *RES* ($\alpha \Rightarrow$ *enat option*) | *FAIL*,

where *enat* is the type of extended natural numbers, i.e. $\mathbb{N} \cup \{\infty\}$ and $\alpha \Rightarrow \beta$ *option* is the standard idiom in Isabelle to model a map from α to β. The type α *NREST* describes non-deterministic results with time bounds, where *RES M* describes the non-deterministic choice of an element from the domain of M while consuming no more time units than M specifies for that element. *FAIL* describes a failed computation, which usually stems from an assertion that was not satisfied.

We define a *refinement ordering* on NREST by first lifting the ordering on *enat* to option with *None* as the bottom element, then pointwise to functions and finally to α *NREST*, setting *FAIL* as the top element. With that ordering, NREST forms a complete lattice where *RES* ($\lambda s.$ *None*) is the bottom element, and *FAIL* is the top element. Intuitively, $N \leq M$ means that program N *refines* program M, i.e. all results of N are also results of M, and further for each such result, N takes no more time than M does. Any program refines $FAIL$.

▶ **Example 1.** A program that reverses a list and whose run-time is at most four times the length of the list can be specified by: *rev_spec xs* = *RES* [*rev xs* $\mapsto$ 4∗|*xs*|]

Here, $[a \mapsto b]$ is syntactic sugar for ($\lambda x.$ `if` $x{=}a$ `then` *Some b* `else` *None*).

On the type NREST we define the following functions:

```
consume :: α NREST ⇒ enat ⇒ α NREST where
consume (RES M) t = RES (λx. case M x of None ⇒ None | Some t' ⇒ Some (t + t'))
consume FAIL t = FAIL
```

```
return :: α ⇒ α NREST where
return x = RES [ x ↦ 0 ]
```

```
bind :: α NREST ⇒ (α ⇒ β NREST) ⇒ β NREST  where
bind (RES M) f = Sup { consume (f x) t |x t. M x = Some t}
bind FAIL f = FAIL
```

The term `consume` M t describes the computation M prolonged by t time steps, `return` x is a computation that yields a single result x in no time, and `bind` m f is the sequential composition of two computations: First compute any result x of m, then any result y of $f\ x$. The time bounds for the final results have to be determined considering all possible ways how to reach them. If m or any reachable computation path of f fails the compound computation also fails. NREST together with `bind` and `return` forms a monad and `bind` as well as `consume` are monotonic w.r.t. the refinement ordering:

```
m ≤ m' ⟶ (∀x. f x ≤ f' x) ⟶ bind m f ≤ bind m' f'
m ≤ m' ⟶ t ≤ t' ⟶ consume m t ≤ consume m' t'
```

▶ **Example 2.** Let m = *RES* (λ*_::nat. Some 0*) and $f\ v$ = `consume` (`return` *0*) v. Program m computes any natural number in no time, and f takes a natural number v as argument and computes the result *0* in at most v steps. Now consider `bind` m f: Both m and f do not fail, and together compute the single result *0*. But there are computation paths (via any value v produced by m) with any natural number as a run-time. The supremum over all these is ∞. To sum it all up: `bind` m f = `consume` (`return` *0*) ∞. This illustrates why we had to choose *enat* for the range of the run-time bound, rather than the type of natural numbers.

Furthermore we define two derived operations:

$SPEC :: (\alpha \Rightarrow bool) \Rightarrow (\alpha \Rightarrow enat) \Rightarrow \alpha\ NREST$ **where**
$SPEC\ P\ t = RES\ (\lambda v.\ \texttt{if}\ P\ v\ \texttt{then}\ Some\ (t\ v)\ \texttt{else}\ None)$

$\texttt{assert} :: bool \Rightarrow unit\ NREST$ **where**
$\texttt{assert}\ P = (\texttt{if}\ P\ \texttt{then return}\ ()\ \texttt{else}\ FAIL)$

A computation that returns a result v if and only if $P\ v$ holds and takes at most $t\ v$ time is described by $SPEC\ P\ t$. The computation $\texttt{assert}\ P$ fails if the predicate P is not satisfied. For assertions we have the following rules:

$P \longrightarrow m \leq m' \longrightarrow \texttt{do}\ \{\ \texttt{assert}\ P;\ m\ \} \leq m'$
$(P \longrightarrow m \leq m') \longrightarrow m \leq \texttt{do}\ \{\ \texttt{assert}\ P;\ m'\ \}$

Here, we use a Haskell-like do notation as a convenient syntax for bind operations. The first rule is used to show that a program m with assertion P refines the program m'. It requires to prove P, in addition to the refinement $m \leq m'$. The second rule is used to show that a program m refines a program m' with an assertion. It allows one to assume P when proving the refinement $m \leq m'$. This way, facts that are proven on the abstract level are made available for proving refinement.

2.2 Recursive Programs

Non-recursive programs can be expressed by the monad operations and Isabelle/HOL's if and case-combinators. Recursion is encoded by a fixed point combinator *RECT*, such that *RECT F* is the greatest fixed point of the monotonic functor F, w.r.t. the flat ordering of timed result maps with *FAIL* as the top element. For any non-monotonic F, *RECT F* is set to *FAIL*:

$RECT :: ((\beta \Rightarrow \alpha\ NREST) \Rightarrow \beta \Rightarrow \alpha\ NREST) \Rightarrow \beta \Rightarrow \alpha\ NREST$ **where**
$RECT\ F\ x = (\texttt{if}\ mono2\ F\ \texttt{then}\ (gfp\ F\ x)\ \texttt{else}\ FAIL)$

Here, *mono2* denotes monotonicity w.r.t. both the flat ordering and the refinement ordering. The benefits of this are explained in more detail elsewhere [11]. Note that programs constructed by the combinators we introduced above are monotonic in that sense by construction. The combinator *RECT* is also monotonic w.r.t. the refinement ordering:

$mono2\ B \wedge (\forall F\ x.\ B\ F\ x \leq B'\ F\ x) \longrightarrow RECT\ B\ x \leq RECT\ B'\ x$

For all other combinators we can show similar monotonicity lemmas. Building on them, we also define while loops, foreach loops and a fold function to conveniently express tail recursion, folding over the elements of a finite set and folding over a list.

▶ **Example 3.** As a running example we consider the formalization of Kruskal's algorithm. To illustrate the expressive power of NREST we present the abstract algorithm in Figure 1a: the greedy algorithm to construct a minimum weight basis for a matroid. This abstract algorithm will later be instantiated for the cycle matroid, which yields the skeleton of Kruskal's algorithm. Already on this abstract level we can structure the algorithm and prove the functional correctness of the algorithmic idea, as well as its run-time – parameterized over the run-times of the abstract operations it performs.

```
1  minWeightBasis = do {
2   l ← SPEC (λL. sorted_wrt w L
3       ∧ distinct L ∧ set L = E)
4            (λ_. t_sc);
5   s ← RES [∅ ↦ t_eb];
6   T ← nfold l (λe T. do {
7    assert (e∉T ∧ indep T ∧ e∈c ∧ T⊆E);
8    b ← RES [indep (T∪{e})↦t_it];
9    if b then do {
10
11     RES [T∪{e}↦t_i]
12   } else
13     return T
14  }) s;
15  return T
16 }
```

(a) The greedy algorithm to construct a minimum weight basis of a Matroid in the NREST monad.

```
1  Kruskal = do {
2   l ← obtain_sorted_edge_list;
3
4
5   (djs0, fl0) ← initState;
6   (djs, fl) ← nfold l (λ(a,w,b) (djs, fl). do {
7    assert (a∈Domain djs ∧ b∈Domain uf);
8    b ← RES [¬djs_cmp djs a b ↦ t_it];
9    if b then do {
10     assert ((a,w,b)∉set fl);
11     addEdge djs a b fl
12   } else
13    return (djs,fl)
14  }) (djs0, fl0);
15  return fl
16 }
```

(b) A further refinement for the Kruskal algorithm, where an additional disjoint sets data structure is passed around.

Figure 1 Two examples of algorithms in the the timed non-determinism monad.

In line two the algorithm obtains a list of the elements of the carrier set E (later this will be the set of edges of an undirected graph) sorted w.r.t. some weight function w. Starting from an empty independent set, we iteratively add elements if they leave the set T independent (i.e. create no cycle in the graph case). For all operations that may cost time, we reserve some time parameter of type *nat* or functions to *nat*: here t_{sc}, t_{eb}, t_{it} and t_i stand for sorted carrier set time, empty basis time, independence test time and insertion time.

We can give the specification for this algorithm, and state the refinement theorem:

$$minWeightBasis \leq SPEC\ minBasis\ (\lambda\_.\ t_{sc} + t_{sb} + |E| * (t_{it} + t_i))$$

where *minBasis* S is true iff S is a minimum weight basis. How to prove such a refinement in a mechanized way is the subject of the next section.

2.3 Generalizing the Weakest Precondition

First let us consider refinement goals with a result on the right hand side: $c \leq RES\ Q$

That is, we want to prove that a program c meets specification Q. Note that program c might be a composed program using the combinators defined above. In order to come up with meaningful rules for these combinators we first need to generalize the above goal.

Instead of asking only *whether* a program satisfies the specification, we also ask "*how much*" it satisfies the specification, i.e. how much slack time is between the specified and actual run-time. As a mental model, we place the "slack time" *in front* of the actual run-time and call it the *latest starting time* such that executing c always terminates before the deadline $Q :: \alpha \Rightarrow enat\ option$, and denote it as $lst\ c\ Q :: enat\ option$.

If program c does not fulfill a specification Q then there is no such time and $lst\ c\ Q = None$, otherwise its value is the latest feasible starting time. Before we give the definition of *lst*, let us explore what we can do with it. We obtain the following equality:

$$c \leq RES\ Q \longleftrightarrow Some\ 0 \leq lst\ c\ Q$$

and we can prove the following equation for the bind operator:

$$lst\ (\texttt{bind}\ m\ f)\ Q = lst\ m\ (\lambda y.\ lst\ (f\ y)\ Q)$$

Intuitively it says: The latest starting time for the compound computation `bind` $m\ f$ to satisfy Q is the latest starting time for m in order to meet the latest starting time such that $f\ y$ meets the specification Q.

To determine $lst\ c\ Q$, we need to consider the differences between the specified and the actual run-time for every result of c and take the most conservative one:

$$lst\ c\ Q = Inf\ r.\ minus\ Q\ c\ r$$

Operation $minus :: (\alpha \Rightarrow enat\ option) \Rightarrow \alpha\ NREST \Rightarrow \alpha \Rightarrow enat\ option$ formalizes taking the difference. We have the following cases:

- c fails: then c may never be executed and thus there is no valid latest starting time, i.e. $minus\ Q\ c\ r = None$.
- $c{=}RES\ C$ and $C\ r = None$: as C will never produce the result r, it can be ignored, i.e. the result is the top element: $Some\ \infty$.
- $c{=}RES\ C$ and $C\ r = Some\ m$ and $Q\ r = None$: r is specified to not be obtained, but when starting c we obtain r, thus there is no valid starting time for C: $minus\ Q\ c\ r = None$.
- $c{=}RES\ C$ and $C\ r = Some\ m$ and $Q\ r = Some\ n$: if more time is needed than specified ($n < m$) there is no valid latest starting time and we return $None$, otherwise the difference is returned ($Some\ (n{-}m)$).

We can get some more intuition when unfolding lst in the above equality:

$$\begin{aligned}
& c \leq RES\ Q \\
\longleftrightarrow\ & Some\ 0 \leq lst\ c\ Q && (= Inf\ r.\ \ minus\ Q\ c\ r\ ) \\
\longleftrightarrow\ & \forall r.\ \ Some\ 0 \leq minus\ Q\ c\ r
\end{aligned}$$

The infimum is just a compact version of saying that the difference of Q and c on *any* result r is non-negative. By abusing notation and following the intuition of minus one can restate the last line as "$\forall r.\ c\ r \leq Q\ r$". In essence it says, that c meets specification Q, iff for any r the time that it takes to calculate r for c is at most the time that Q reserved for that result.

2.4 Sound proof rules for the latest starting time calculus

Instead of solving problems of the form $c \leq RES\ Q$ we solve problems of the more general form $Some\ t \leq lst\ c\ Q$. This general form allows us to state syntax directed rules in a uniform way, which would not be possible otherwise.

From the equality for lst on `bind` we can derive an introduction rule for `bind`:

$$Some\ t \leq lst\ M\ (\lambda y.\ lst\ (f\ y)\ Q) \longrightarrow Some\ t \leq lst\ (\texttt{bind}\ M\ f)\ Q$$

For the other combinators we have:

$$\begin{aligned}
&(\forall r{\in}M.\ Some\ (t + M\ r) \leq Q\ r) \longrightarrow Some\ t \leq lst\ (RES\ M)\ Q \\
&Some\ t \leq Q\ x \longrightarrow Some\ t \leq lst\ (\texttt{return}\ x)\ Q \\
&(\forall x.\ P\ x \longrightarrow Some\ (t + t'\ x) \leq Q\ x) \longrightarrow Some\ t \leq lst\ (SPEC\ P\ t')\ Q \\
&Some\ (t + t') \leq lst\ M\ Q \longrightarrow Some\ t \leq lst\ (\texttt{consume}\ M\ t')\ Q
\end{aligned}$$

For the fold operation $nfold :: \beta\ list \Rightarrow (\beta \Rightarrow \alpha \Rightarrow \alpha\ NREST) \Rightarrow \alpha \Rightarrow \alpha\ NREST$ we have the following rule:

$$
\begin{aligned}
&I\ []\ l_0\ s_0 \\
&\wedge\ (\forall x\ l_1\ l_2\ s.\ l_0{=}l_1 \cdot [x] \cdot l_2 \wedge I\ l_1\ ([x] \cdot l_2)\ s \\
&\qquad \longrightarrow Some\ 0 \leq lst\ (f\ x\ s)\ (emb\ (I\ (l_1 \cdot [x])\ l_2)\ t_{body})) \\
&\wedge\ (\forall s.\ I\ l_0\ []\ s \longrightarrow Some\ (t + t_{body} * |l_0|) \leq Q\ s) \\
&\longrightarrow Some\ t \leq lst\ (nfold\ l_0\ f\ s_0)\ Q
\end{aligned}
$$

Here, $emb\ P\ t = (\lambda x.\ \texttt{if}\ P\ x\ \texttt{then}\ Some\ t\ \texttt{else}\ None)$, *nfold* is defined in a straightforward manner and the invariant I is a predicate that takes as its first argument the list of already processed elements, then the list of elements still to be processed and finally a state s. For showing that $nfold\ l_0\ f\ s_0$ meets its specification Q with slack time t, one has to show that an invariant I holds initially, the body preserves the invariant and takes at most t_{body} time steps and the invariant in the end implies the desired specification. As we fold over a finite list, a termination argument is not required.

We also define a rule for *RECT* and based on that one for while loops. With the above rules and analogous rules for `assert` and the combinators `if` and `case`, we construct a syntax directed verification condition generator that exhaustively applies those rules.

▶ **Example 4.** After annotating the loop in the abstract program from Figure 1b with $body_{time} = t_{it} + t_i$ and a suitable invariant $I = \lambda l_1\ l_2\ T.\ I_{mwb}\ (T,\ set\ l_2)$ (where $I_{mwb}(T, E)$ implies $minBasis\ T$ for the whole carrier set E), we run the VCG on the refinement theorem of Example 3 and obtain eleven verification conditions. One of these is the invariant preservation of the first branch of the if-expression, i.e. when adding an element e:

$$
\begin{aligned}
&sorted\_wrt\ w\ l \wedge distinct\ l \wedge set\ l = E \wedge l = l_1 \cdot [e] \cdot l_2 \wedge indep\ (T \cup \{e\}) \\
&\wedge\ I_{mwb}(T,\ set\ ([e] \cdot l_2)) \longrightarrow I_{mwb}(T \cup \{e\},\ set\ l_2)
\end{aligned}
$$

This verification condition is one of the central ones in the correctness proof and can be discharged with an interactive proof.

2.5 Data Refinement

In the process of refining an abstract algorithm to a more concrete one, a usual task is to replace abstract data structures by concrete ones, for example to replace sets by lists. Consider the then branch in the algorithm in Figure 1a: instead of using a set to collect the elements of a basis, we want use a list. We have the following refinement in mind. Given that a list l represents a set T (denoted by $(l,T){\in}list\_set\_rel$), the resulting lists of the program on the left hand side refine the resulting sets produced by the right hand side program:

$$(l,T){\in}list\_set\_rel \longrightarrow RES\ [l \cdot [x] \mapsto it] \leq \Downarrow(list\_set\_rel)\ RES\ [T{\cup}\{x\} \mapsto it]$$

Given a refinement relation R, i.e. a relation that relates concrete elements with abstract elements, the concretization function $\Downarrow$R maps abstract results to concrete results w.r.t. R. Note that, if R is single-valued any concrete result is mapped to at most one abstract result.

$$
\begin{aligned}
&\Downarrow R\ FAIL = FAIL \\
&\Downarrow R\ (RES\ X) = RES\ (\lambda c.\ Sup\ \{X\ a |\ a.\ (c,a){\in}R\})
\end{aligned}
$$

Data refinement is orthogonal to introducing the time counting, as it only acts on the domain of the maps, not on their values. We can lift all monotonicity lemmas to also include the data refinement, e.g. for the bind operation we obtain the following rule:

$M \leq \Downarrow R'\ M' \wedge (\forall x\ x'.\ (x,\ x') \in R' \longrightarrow f\ x \leq \Downarrow R\ (f'\ x')) \longrightarrow \texttt{bind}\ M\ f \leq \Downarrow R\ (\texttt{bind}\ M'\ f')$

Analogous rules can be proven for *RECT*, *nfold*, `assert`, and the other combinators.

2.6 Setting Up a VCG for Refinement

In practice, one mostly is confronted with two kinds of refinement goals: first, goals w.r.t. a specification $c \leq RES\ Q$, which we already considered, and second, refinement of two abstract algorithms that are structurally similar (c.f. Figure 1). For the latter case, one simulates the two programs in lock step and uses the monotonicity lemmas mentioned in the last section to divide and conquer the problem. Collecting these rules we construct an automated refinement solver, which we illustrate with an example:

▶ **Example 5.** Consider the two programs in Figure 1. The concrete program *Kruskal* is a specialized minimum weight basis algorithm for the cycle matroid, where the elements of the matroid are edges in an undirected graph, represented by a tuple (a, w, b) of its end nodes a and b and weight w. Programs *obtain_sorted_edge_list* and *addEdge* are compound programs. We want to show the following refinement relation:

$Kruskal \leq \Downarrow\ list\_graph\_rel\ minWeightBasis$

where *list_graph_rel* relates a set of abstract edges in the graph with a list of edge tuples representing them. When showing this refinement, several other intermediate refinement relations are used, e.g. $((djs,fl),T) \in djs\_graph\_rel$ which relates the abstract edge set T to the list of edges fl and its corresponding disjoint-sets data structure. The main part of this refinement proof is to show that testing independence if we add an edge (a, w, b) (i.e. checking cycle-freedom) can be implemented by comparing the equivalence classes of a and b.

Note that *addEdge* has to do two things: update the disjoint-sets data structure and add the edge tuple to the list. We specify this program abstractly, and reserve time t_{iu} and t_{il} for the two actions. In the refinement proof we need to prove that $t_{iu} + t_{il} \leq t_i$. Similarly, the sum of the costs in *obtain_sorted_edge_list* must be smaller than t_{sc}.

The VCG for refinement simulates the two programs side by side, using the monotonicity lemmas to split the problem into smaller parts, and showing the refinements of those smaller parts. One such part is the goal $addEdge\ djs\ a\ b\ fl \leq \Downarrow list\_graph\_rel\ (RES\ [T \cup \{e\} \mapsto t_i])$ (with *list_graph_rel* motivated as above).

3 Refinement to Imperative/HOL with Time

In this section we introduce the time-aware monad of Imperative/HOL [17], which we then use as the target monad of the adapted Sepref tool [11] with NREST as the source monad.

3.1 Imperative/HOL with Time

Imperative/HOL with time [17] incorporates Atkey's [1] idea to include *time credits* in separation logic into the Imperative/HOL [2] framework. In essence, it enables reasoning about imperative programs and their run-time in Isabelle/HOL. While all the details can be found in Section 2.1 of [17], we will give an abstract explanation here that suffices for our purposes.

A procedure in the monad takes a heap as input and can either fail or return a tuple consisting of a return value, a new heap and a natural number, specifying the number of computation steps used. The type of a procedure with result type α is given by:

```
datatype α Heap = Heap (heap ⇒ (α × heap × nat) option)
```

The bind operator as well as fix point iteration, while and other combinators are defined in a straightforward manner. The term $(h,c) \Rightarrow (r,\ h',\ t)$ expresses that procedure c started on heap h does not fail and takes time t to produce result r and heap h'.

While heaps themselves do not form a separation algebra, there is an abstraction function α that maps a pair of heap and time credits to an abstract heap. Abstract heaps together with suitable definitions of disjointness and heap addition form a separation algebra. An assertion P, i.e. a mapping from an abstract heap to bool, being true for a heap h and time credits n is denoted by $\alpha(h,n) \models P$. There are basic assertions for an abstract heap containing an array without time credits $(a \mapsto_a xs)$, references without time credits $(r \mapsto_r v)$ and time credits ($n).

The *separating conjunction* $P{*}Q$ expresses that the heap and time credits can be partitioned into two disjoint parts satisfying assertions P and Q respectively. The strength of separation logic is, that this disjointness enables modular reasoning, which also carries over to reasoning about time credits.

Hoare triples are defined in the following way:

```
1 <P> c <λr. Q r>_t =
2  (∀h n. α (h,n) ⊨ P ⟶ (∃h' t r. (c,h) ⇒ (r, h', t)
3                                  ∧ α (h', n − t) ⊨ Q r ∗ true ∧  t ≤ n) )
```

where the assertion *true* is true for any heap, thus enabling garbage collection of heap elements and time credits. The Hoare triple $<P> c <\lambda r.\ Q\ r>_t$ denotes that procedure c started from a heap satisfying P terminates with a return value r in a resulting heap that satisfies $Q\ r * true$. In particular it states that the starting heap holds enough time credits n in order to pay for the cost t of executing the procedure c (see line 3).

The cost model assigns most basic commands (e.g. accessing or updating a reference, getting the length of an array) to consume one unit of computation time. Commands that operate on an entire array take $n+1$ units of computation, where n is the length of the array. Examples for basic commands are:

```
<a ↦_a xs ∗ $1 ∗ ↑(i < |xs|)> Array.upd i x a <λr. a ↦_a xs[i:=x] ∗ ↑(r = a)>_t
<$(n+1)> Array.new n x <λr. r ↦_a replicate n x>_t
```

where $\uparrow P$ is a pure assertion, which is valid for an empty heap if P holds globally, $xs[i:=x]$ denotes a list xs updated at position i with value x, and *replicate* $n\ x$ denotes a list of n elements x.

In Section 4.2 we review available and new infrastructure and automation for proving valid Hoare triples of procedures in the time-aware monad of Imperative/HOL.

3.2 Adapted Sepref

As a next step we want to automatically synthesize programs in the time-aware Imperative/HOL monad from abstract algorithms in the NREST monad. This step is performed by an adaptation of the Sepref tool [11]. Note that, the original tool refines NRES to vanilla Imperative/HOL; adapting it includes many but rather straightforward modifications. During that process we identified common patterns and constraints on the source and target monad. It is future work to come up with a generalized Sepref tool. The core of the tool is the *translation phase*, where the concrete program is synthesized. We focus on that phase as the other phases can be adapted in a straightforward manner.

The translation works by symbolically executing the abstract program, thereby synthesizing a structurally similar concrete program. During the symbolic execution, the relation between the abstract and concrete variables is modeled by refinement assertions. The *synthesis predicate* guiding the "Heap-monad to Non-determinism Refinement" is denoted by $hnr\ \Gamma\ m_\dagger\ \Gamma'\ R\ m$: it means that the concrete program $m_\dagger$ implements the abstract program m, where Γ contains the refinements for the variables before the execution, Γ' contains the refinements after the execution, and R is the refinement assertion for the result of m. For example, a `bind` is processed by the following synthesis rule:

```
1  hnr Γ m† Γ' R_x m ∧
2    (∀x x†. hnr (R_x x x† * Γ') (f† x†) (R'_x x x† * Γ'') R_y (f x))
3  ⟶ hnr Γ (do {x† ← m†; f† x†}) Γ'' R_y (do {x ← m; f x})
```

To refine $x \leftarrow m;\ f\ x$, we first execute m, synthesizing the concrete program $m_\dagger$ (line 1). The state after m is $R_x\ x\ x_\dagger\ *\ \Gamma'$, where x is the result created by m. From this state, we execute $f\ x$ (line 2). The new state is $R'_x\ x\ x_\dagger\ *\ \Gamma''\ *\ R_y\ y\ y_\dagger$, where y is the result of $f\ x$.

While executing the abstract program, not only a concrete program is created, but also the set of refinement assertions Γ evolves: It contains all the data structures (pure or on the heap) that the concrete program maintains.

All the other combinators (*RECT*, `while`, `if`, `case` ...) have similar rules that are used to decompose an abstract program into parts, synthesize corresponding concrete parts recursively and combine them afterwards.

At the leaves of this decomposition one has to find "atomic" operations, with a suitable synthesis rule. An example could be the rule for the specification of the compare operation of a disjoint-sets data structure as in the concrete Kruskal program in Figure 1b:

```
hnr (is_uf R' R * nat_assn a' a * nat_assn b' b) (uf_cmp R a b)
    (is_uf R' R * nat_assn a' a * nat_assn b' b)
    bool_assn (RES [djs_cmp R' a' b' ↦ itt] )
```

The program *uf_cmp* in the time-aware Imperative/HOL monad refines the abstract compare operation *djs_cmp*. If the parameters fulfill the correct refinement assertions, i.e. R is a concrete union-find implementation of the abstract equivalence relation R', as well as $a' = a$ and $b' = b$, then the result of the concrete operation is equal (*bool_assn*) to the result of the abstract one, and the parameters are still in the refinement relations as before.

3.3 Heap-monad to Non-determinism Refinement (HNR)

Now we present how we can link NREST with the Imperative/HOL monad via a suitable synthesis predicate.

```
1  hnr Γ c Γ' R m ≡ m ≠ FAIL ⟶
2   (∀h n. α(h, n) ⊨ Γ ⟶ (∃h' t r. (c,h) ⇒ (r, h', t)
3                                ∧ (∃t_a r_a. α(h', (n+t_a)−t) ⊨ Γ' * R r_a r * true
4                                    ∧ consume (return r_a) t_a ≤ m  ∧ n+t_a≥t)))
```

If the abstract program m does not fail, procedure c started from a heap satisfying Γ produces a heap satisfying Γ' and a result r which relates to an abstract result r_a via relation R. The abstract result r_a is a valid result of m and has at least t_a time units reserved for it. Together with the time credits on the heap n this pays for the execution cost t (line 4).

In particular, the execution cost t is paid for by the time units t_a specified by the abstract program and by time credits n that are hidden in the data structures on the heap. One can see, that amortized data structures seamlessly integrate into the framework: only amortized run-time costs are visible to the abstract algorithm, while the actual run-time and potential is hidden in the implementation.

In order to verify that this definition makes sense, observe what we can prove for it: First, this definition enables us to prove soundness of the synthesis rule for `bind` from above. Second, as a final step in an algorithm analysis we would like to extract a Hoare triple for the concrete program we synthesized. The run-time of final algorithms that we analyze is typically not dependent on the result, but only on the input. For programs with specifications of that special form $SPEC\ P\ (\lambda\_.\ t)$ we can extract a standard Hoare triple from a valid synthesis predicate and vice versa:

$$hnr\ \Gamma\ c\ \Gamma'\ R\ (SPEC\ P\ (\lambda\_.\ t)) \longleftrightarrow <\Gamma\ *\ \$\ t> c <\lambda r.\ \Gamma' * (\exists_A\ r'.\ R\ r'\ r * \uparrow(P\ r'))>_t$$

While during reasoning the abstract time bound needs to depend on the result (in order to prove the synthesis rule for bind correct), when proving the run-time of an algorithm, in most cases the final run-time only depends on the input parameters.

Based on that definition we can provide sound synthesis rules for all the combinators as well as a frame and a consequence rule. To illustrate how the hnr-approach allows to use amortized data structures seamlessly, consider the first case-study in Section 5.1.

4 Modular Algorithms and Proof Development

Using our methodology, algorithm design and analysis can be modularized in two ways:

First, separating the implementation details of data structures from the abstract arguments of algorithms enables focusing on one part of the problem at a time. Both levels have their own language (time-aware Imperative/HOL and the NREST monad), and the interface is realized by abstract operations (e.g. *mop_append_list*) and *hnr* rules. Sepref is employed to automatically synthesize concrete algorithms from abstract ones. On the abstract level we reserve some amount of time for each abstract operation, whose details will get filled in once one decides which data structure and concrete operation to use, then yielding a sound upper bound on the run-time. A collection of abstract operations and their implementations by efficient data structures will be given in the next subsection.

Second, the refinement calculus of NREST programs enables to formulate abstract algorithms that can be reused as components in larger developments. One example is a generic BFS component, that is used as a sub-component in the Edmonds–Karp algorithm. Also abstract algorithms, such as the minimum weight basis algorithm can be formulated on general matroids, and then later be instantiated for the cycle matroid yielding a blue-print for Kruskal's algorithm.

4.1 Library of Operations and Algorithms

Table 1 lists abstract data structures with their abstract operations and the implementations we currently provide in the *Timed Imperative Isabelle Collections Framework* (TIICF). Note: it is easy to extend this list. As an example for a generic re-usable algorithm we provide breadth first search, which is used in the formalization of the Edmonds–Karp algorithm.

Table 1 This table shows the abstract data structures with abstract operations that we provide implementations for in the TIICF. Amortized run-time bounds are marked with an asterisk (*).

abstract	operations	run-time	concrete
matrix	create; lookup, update	$O(n^2)$; $O(1)$	array
set/map	create; insert, lookup, delete, update	$O(1)$; $O(\log n)$	red-black tree
		$O(n)$; $O(1)$	array
list	create, append; lookup, update	$O(1)^*$; $O(1)$	dynamic array
disjoint sets	create; union, find	$O(n)$; $O(\log n)$	union-find

4.2 Methodology and Automation

The process of formalizing an algorithm is supported by automation in four stages. We present those from the most abstract to the concrete:

First, when proving the refinement of a specification in the NREST monad to an abstract algorithm the generation of verification conditions is automated. They can be discharged by automatic tactics or interactive proof.

Second, abstract algorithms are refined to structurally similar concrete algorithms. Here a lock step simulation is carried out automatically by the refinement condition generator. An example is to show the refinement between the programs in Figure 1.

Third, the adapted Sepref tool automatically synthesizes a program in the time-aware Imperative/HOL monad from a given abstract algorithm containing only abstract operations with available *hnr* rules. Automatic proving of side-conditions is performed in a limited way. Usually, preconditions of concrete operations are provided as an `assert` in the abstract algorithm.

Finally, for showing that concrete implementations of abstract operations are correct and satisfy the given time bounds one has to show *hnr* predicates. In essence, these are Hoare triples in time-aware Imperative/HOL. Zhan et al. [17] develop a methodology for proving functional correctness and (amortized) run-time claims and provide a setup for automation. One novel component is a special routine for handling time credits during frame inference. Lammich [11] provides *sep_auto* – a strong automation for vanilla Imperative/HOL – which we extend by the above mentioned time frame inference routine to also handle programs in the time-aware case. Both approaches can be used in order to establish correct Hoare triples of basic data structures and form a library of algorithms and data structures which can be used as abstract operations in more advanced algorithms.

5 Case Studies

In this section we present three case studies:

The first one considers the abstract operation "appending an element to the end of a list" and illustrates three stages of the verification process: implementing the operation for a concrete data structure in Imperative/HOL with time, designing a synthesis predicate relating the concrete with the abstract operation and using it in an abstract algorithm.

The latter two case studies describe the verification of more involved algorithms where refinement helps structuring the development: Kruskal's minimum spanning tree algorithm and the Edmonds–Karp algorithm for maximum network flow.

5.1 Amortized Dynamic Array and Remove Duplicates

Let us consider the following abstract operation, appending an element to the end of a list:

```
mop_push_list t x xs = RES [xs · [x] ↦ t xs]
```

The operation is specified in the NREST monad, with a parameter t that represents the run-time of the operation, here parametrized in the list xs. For an implementor, this leaves open the possibility to provide an implementation whose time consumption depends on xs, e.g. on its length. Let us turn to an implementation of that operation on a dynamic array.

Implementation

An *abstract dynamic list* is represented by a pair of a carrier list bs and a fill level n. The corresponding abstract list as is the list bs restricted to the first n elements:

```
dyn_abs (bs,n) as ⟷ as = take n bs ∧ n<|bs|
```

We define a function *push_array_fun* on abstract dynamic lists that doubles the length of the list if it is full and then appends an element. We prove its functional correctness:

```
dyn_abs (bs,n) as ⟶ dyn_abs (push_array_fun x (bs,n)) (as · [x])
```

Recall that $p \mapsto_a xs$ denotes a heap containing an array at address p with content xs. Based on this, one can define an assertion

```
dyn_array_raw (bs, n) (p, m) = (p ↦a bs ∗ ↑(m = n))
```

relating an abstract dynamic list with a concrete *dynamic array* represented by a pair of address p and fill level m.

For the functional *push_array_fun* we define a corresponding procedure *push_array* which appends an element to the back of a dynamic array, doubling the length if it is exceeded. We can now show the following raw Hoare triple, with worst-case run-time linear in the fill level of the dynamic array, as we might have to double the array. The explicit numbers in the run-time stem from the concrete implementation of *push_array* and the cost model of time-aware Imperative/HOL.

```
n ≤ length bs ⟶
<dyn_array_raw (bs, n) p ∗ $(5∗n + 9)>
 push_array x p
<λp'. dyn_array_raw (push_array_fun x (bs, n)) p'>t
```

We now incorporate the potential ($\Phi(bs,n) = 10 * n - 5 * |bs|$) into an assertion for a compound data structure *dyn_array* and prove the following Hoare triple with amortized constant run-time:

```
dyn_array r p = dyn_array_raw r p ∗ $(Φ r)
```

```
n ≤ length bs ⟶
<dyn_array (bs, n) p ∗ $19>
 push_array x p
<λp'. dyn_array (push_array_fun x (bs, n)) p'>t
```

Note that for showing the latter amortized Hoare triple it does not suffice to employ the raw Hoare triple, rather *push_array* must be unfolded again.

As a final step we compose the refinements of abstract lists to abstract dynamic lists (*dyn_abs*) and further to dynamic arrays (*dyn_array*) and obtain *dyna_assn*:

```
dyna_assn as p = (∃_A bs n. dyn_array (bs,n) p * ↑(dyn_abs (bs,n) as))
```

where the list and fill level of the abstract dynamic array are hidden behind an existential quantifier. Then we obtain the final Hoare triple of the procedure:

```
<dyna_assn as p * $19> push_array x p <λp'. dyna_assn (as · [x]) p'>_t
```

Together with the definition of *mop_push_list* we can state and prove the synthesis predicate for the append operation:

```
19 ≤ t xs' ⟶ hnr (dyna_assn xs' p * Id x' x) (push_array x p)
                 (Id x' x) dyna_assn (mop_push_list t x' xs')
```

Usage

The abstract operation *mop_push_list* can now be used when specifying an abstract algorithm. Then a concrete time function t can be specified, which is used to determine the overall cost of the algorithm. In this example we choose (*λ_. 23*), which is not a tight bound but enough to later allow synthesizing a concrete program using dynamic arrays.

Consider the following program to remove duplicates from a list.

```
1  remdups_impl as = do {
2    ys ← mop_empty_list 12;
3    S ← mop_set_empty 1;
4    (zs,ys,S) ← whileT (λ(xs,ys,S). |xs| > 0) (λ(xs,ys,S). do {
5      assert (|xs| > 0 ∧ |xs| + |ys| ≤ |as| ∧ |S| ≤ |ys|);
6      (x,xs) ← return (hd xs, tl xs);
7      b ← mop_set_member (λ_. rbt_search_t (|as| + 1) + 1) x S;
8      if b then
9        return (xs,ys,S)
10     else do {
11       S ← mop_set_insert (λ_. rbt_insert_t (|as| + 1)) x S;
12       ys ← mop_push_list (λ_. 23) x ys;
13       return (xs,ys,S)
14     }
15   }) (as,ys,S);
16   return ys
17 }
```

The program uses *mop_push_list* from above as well as other abstract operations with corresponding reserved run-time functions. For example insertion into a set:

```
mop_set_insert t x S =  RES [S ∪ {x} ↦ t S]
```

For each operation in the program some time is reserved. The overall run-time of the program is then a function of these reserved quantities.

Let *remdups_t n = n*(60 + rbt_search_t (n+1) + rbt_insert_t (n+1)) + 20.*

Note that for the set operations the reserved time in *remdups_t* is not parametrized in the size of the set they operate on, but in an over-approximation of it: the length of the input. Our automation can prove the following refinement theorem and asymptotic bound:

$$remdups\_impl\ as \leq SPEC\ (\lambda ys.\ set\ ys = set\ as \wedge distinct\ ys)\ (\lambda\_.\ remdups\_t\ |as|)$$
$$remdups\_t \in \Theta(\lambda n.\ n * \log n)$$

When synthesizing an Imperative/HOL program, the synthesis rules will be applied and their preconditions must be discharged. For the *mop_push_list* this boils down to the trivial check $16 \leq 23$. Note that in that process only the advertised cost of the dynamic array is concerned, while the amortization is hidden at this level.

Let us consider a more interesting operation. The synthesis rule of the red-black tree implementation of *mop_insert_set* is the following:

$$rbt\_insert\_t\ (card\ S + 1) \leq t\ S \longrightarrow$$
$$hnr\ (Id\ x'\ x * rbt\_set\_assn\ S\ p)\ (rbt\_set\_insert\ x\ p)$$
$$(Id\ x'\ x)\ rbt\_set\_assn\ (mop\_set\_insert\ t\ x'\ S)$$

where *rbt_set_assn S p* relates a set S with a red-black tree at address p. During synthesis the Sepref tool has to check whether there is enough reserved time for the set insertion.

$$|S| \leq |ys| \wedge |xs| + |ys| \leq |as|$$
$$\longrightarrow rbt\_insert\_t\ (|S| + 1) \leq (\lambda\_.\ rbt\_insert\_t\ (|as| + 1))\ S$$

The goal can be discharged with the knowledge from the assertions and the monotony of *rbt_insert_t*.

Once more, note that amortized data structures seamlessly can be modeled using time credits, and this comfort extends to also be available for the abstract algorithm. At the abstract level, an amortized data structure behaves just as a normal data structure does.

5.2 Kruskal

Kruskal's algorithm was verified in the standard Refinement Framework in parallel to the research reported on in this paper. It can be found in the archive of formal proofs [9]. As a case study, we port it to NREST, adding the run-time claims.

The proof development follows this general structure: first we define the abstract algorithm for minimum weight basis in matroids (c.f. Figure 1a) and verify it. Then we instantiate it with the cycle matroid for forests in undirected graphs and refine the algorithm with the usage of equivalence classes. Figure 1b shows the last-but-one stage in the step-wise refinement process. In a last step we fix the vertices to be natural numbers and the domain of the disjoint-set data structure to be the set from `{0,...,M}`, with M being the maximal vertex in the graph. After that, we use the implementation of the union-find data structure from the TIICF to synthesize a concrete algorithm with the Sepref tool.

Provided a procedure that obtains a list of edges of a graph in linear time, a $O(n * \log n)$ sorting algorithm and a union-find data structure with logarithmic find and union operations we obtain a concrete algorithm that calculates the minimum weight spanning forest for the graph in time $O(E * \log E + M + E * \log M)$, with E being the number of edges and M being the maximal vertex in the graph.

We have only proven the logarithmic bounds for the union-find data structure for this case-study. Charguéraud et al. [4] verified a union-find data structure with amortized run-time $O(\alpha(M))$ (where M is the size of the domain of the disjoint-set data structure and α is the inverse Ackermann function) in Coq.

When developing this case study, we learned that the correctness arguments can be plainly reused, and that adding the proofs of the run-time claims does not interfere, as they only evoke additional verification conditions and leave the ones concerned with functional correctness unchanged. However, it is necessary to add more assertions in the algorithms that speak about the sizes of the data structures used. This reasoning is mostly done on the abstract level, but the information has to be passed to the concrete algorithm via assertions. In the Sepref translation phase, this information is needed to discharge the preconditions of the *hnr* predicates, which demand that enough time has been reserved to execute the step.

5.3 Edmonds–Karp: Reuse

Before starting this project we had the following working hypothesis:

"Formalizations in the standard Refinement Framework can be easily extended to also verify the run-time behaviour. In this process, most of the formalization can be reused, and termination arguments can be translated into run-time arguments."

We conducted this extension to the Edmonds–Karp algorithm [13, 14] as a case-study. The result is two-fold: For procedures where the reasoning on the run-time of the algorithm is already well prepared making this claim explicit is straightforward, for procedures where only termination has been shown only coarse bounds can be shown with little effort. Fine tuned run-time bounds require substantial work.

The Edmonds–Karp algorithm repeatedly tries to increase the flow by searching for an augmenting path in the residual graph and terminates successfully if no such path exists. The search is conducted by a breadth-first search (BFS) on the residual graph. The structure of the development follows the original proof [13, 14]; we only give an abstract overview here:

First, an abstract BFS is defined and verified to return the shortest path from some start node to some end node. The algorithm is parametrized on some graph $G{=}(V{,}E)$ and some procedure that provides the successors of a node in that graph. The run-time of the BFS consequently depends on $|V|$ and $|E|$ as well as the run-time of the successor procedure and the allotted run-times for the data-structure operations used.

Second, an abstract Edmonds–Karp algorithm is defined assuming a procedure to find the shortest path in a graph. For that algorithm functional correctness is proven as well as the correct run-time bound depending on the underlying network, the run-time of the shortest-path algorithm and the run-times of the operations that maintain the residual graph.

Finally, by implementing the operations on the residual graph, in particular its successor function, the abstract algorithms can be interpreted and we obtain a concrete algorithm in NREST together with a refinement theorem and a compound run-time function. For that algorithm we synthesize a program in timed Imperative/HOL together with a correctness theorem and a run-time bound in $O(V * E * (E + V))$. Residual graphs are represented by matrices, for which we provide an array implementation in the TIICF, with linear-time initialization, and constant-time update and lookup operations.

Lammich et al. [14] already quite explicitly work out the bound $O(V * E)$ for the outer loop iterations of the Edmonds–Karp algorithm. We were able to reuse the whole proof and additionally embed the result into our time aware non-determinism monad, thus making the run-time claim less ad-hoc. On the other hand the inner BFS is only proven to terminate via a terminating lexicographic ordering. Plainly using this leads to a valid but very coarse run-time bound. Establishing the tight $O(E + V)$ bound involves some amortized argument on the abstract level and was a considerable verification effort, but again orthogonal to the functional correctness proof, which in turn can be reused with no change.

6 Conclusion

6.1 Related Work

Lammich pioneered the Sepref tool [11] and it has been used to verify several interesting algorithms and software projects [13, 6, 16]. It was recently adapted to synthesize programs in LLVM [12] instead of Imperative/HOL. Coming up with a generic Sepref tool that is parametrized in the target and source language, as well as extending the LLVM semantics to run-time are interesting future projects.

As already mentioned, time-aware Imperative/HOL is due to Zhan et al. [17], which builds upon Atkey's [1] idea to use Time Credits in Separation Logic.

In the Coq community similar theory [3, 7] and the run-time analysis of interesting algorithms [8] and data structures [4] have been formalized.

To the best of our knowledge, we are the first to combine run-time analysis with refinement.

6.2 Limitations and Future Work

In particular, we are not satisfied with the parametrization of operations with timing functions. We envision not only counting one currency ($) representing one computation step in the final concrete algorithm, but to have currencies for abstract operations. Say one abstract algorithm A incurs cost of one "*A-dollar*" $\$_A$ and can be implemented by an algorithm using several operations C_1 and C_2 costing some $\$_{C_1}$ and some $\$_{C_2}$. Refining algorithms that use several calls to A should then routinely yield a refinement with costs in terms of $\$_{C_1}$ and $\$_{C_2}$. A target monad of Sepref then would also allow different actions and respective currencies. Refining abstract operations into this target would exchange these currencies in a sound way, such that ultimately upper bounds on the usage of these currencies are obtained.

In this paper we only study upper bounds of run-time of algorithms. This should be relaxed in two ways: First, consider other quantities, e.g. stack usage, or energy usage. Second, not only upper bounds can be reasoned about, also lower bounds are feasible. A refinement relation on lower bounds seems to be straightforward. Also combining this in a pair of *enat*s and keeping track of lower as well as upper bounds seems to be feasible.

We already mentioned, that Lammich's LLVM semantics could be extended to counting the number of operations. Obviously, it is future work to extend the collection of efficient data structures and reusable algorithms, as well as lowering the barriers to verify run-time arguments by providing more automation.

6.3 Conclusion

In this paper, we have combined the refinement approach of algorithm verification with techniques to verify the run-time of algorithms: We extended the Isabelle Refinement Framework to express the result and time consumption of abstract algorithms as well as the Sepref tool to synthesize executable imperative programs for such abstract algorithms. This setup makes it possible to carry out the verification of algorithms such as Edmonds–Karp and Kruskal in a modular way. Separating concerns into the abstract algorithmic idea and the implementation details of data structures makes larger proof developments feasible.

Our use-cases indicate that for additionally verifying run-time arguments for algorithms whose functional correctness has already been shown within the vanilla Isabelle Refinement Framework, formalizations can be reused to a large extent. We think that even larger developments can be tackled this way, both verifying functional correctness and the run-time analysis of such algorithms.

References

1. Robert Atkey. Amortised Resource Analysis with Separation Logic. In *ESOP*, volume 6012, pages 85–103. Springer, 2010.
2. Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkok, and John Matthews. Imperative functional programming with Isabelle/HOL. *Lecture Notes in Computer Science*, 5170:134–149, 2008.
3. Arthur Charguéraud. Characteristic Formulae for the Verification of Imperative Programs. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 418–430, New York, NY, USA, 2011. ACM. doi:10.1145/2034773.2034828.
4. Arthur Charguéraud and François Pottier. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *Journal of Automated Reasoning*, pages 1–35, 2017.
5. Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A fully verified executable LTL model checker. In *International Conference on Computer Aided Verification*, pages 463–478. Springer, 2013.
6. Mathias Fleury, Jasmin Christian Blanchette, and Peter Lammich. A verified SAT solver with watched literals using Imperative HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 158–171. ACM, 2018.
7. Armaël Guéneau, Arthur Charguéraud, and François Pottier. A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification. In *European Symposium on Programming (ESOP)*, 2018.
8. Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. Formal Proof and Analysis of an Incremental Cycle Detection Algorithm. In *International Conference on Interactive Theorem Proving*. Springer, 2019. URL: http://gallium.inria.fr/~agueneau/publis/gueneau-jourdan-charguer aud-pottier-2019.pdf.
9. Maximilian P.L. Haslbeck, Peter Lammich, and Julian Biendarra. Kruskal's Algorithm for Minimum Spanning Forest. *Archive of Formal Proofs*, February 2019. , Formal proof development. URL: http://isa-afp.org/entries/Kruskal.html.
10. Peter Lammich. Verified efficient implementation of Gabow's strongly connected component algorithm. In *International Conference on Interactive Theorem Proving*, pages 325–340. Springer, 2014.
11. Peter Lammich. Refinement to Imperative/HOL. In *International Conference on Interactive Theorem Proving*, pages 253–269. Springer, 2015.
12. Peter Lammich. Generating Verified LLVM from Isabelle/HOL. In *International Conference on Interactive Theorem Proving*. Springer, 2019.
13. Peter Lammich and S Reza Sefidgar. Formalizing the edmonds-karp algorithm. In *International Conference on Interactive Theorem Proving*, pages 219–234. Springer, 2016.
14. Peter Lammich and S. Reza Sefidgar. Formalizing the Edmonds-Karp Algorithm. *Archive of Formal Proofs*, August 2016. , Formal proof development. URL: http://isa-afp.org/entries/EdmondsKarp_Maxflow.html.
15. Peter Lammich and Thomas Tuerk. Applying data refinement for monadic programs to Hopcroft's algorithm. In *International Conference on Interactive Theorem Proving*, pages 166–182. Springer, 2012.
16. Simon Wimmer and Peter Lammich. Verified model checking of timed automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 61–78. Springer, 2018.
17. Bohua Zhan and Maximilian P. L. Haslbeck. Verifying asymptotic time complexity of imperative programs in Isabelle. In *International Joint Conference on Automated Reasoning*, pages 532–548. Springer, 2018.

Virtualization of HOL4 in Isabelle

Fabian Immler
School of Computer Science, Carnegie Mellon University, USA
fimmler@cs.cmu.edu

Jonas Rädle
Fakultät für Informatik, Technische Universität München, Germany
raedle@in.tum.de

Makarius Wenzel
Augsburg, Germany
https://sketis.net

Abstract

We present a novel approach to combine the HOL4 and Isabelle theorem provers: both are implemented in SML and based on distinctive variants of HOL. The design of HOL4 allows to replace its inference kernel modules, and the system infrastructure of Isabelle allows to embed other applications of SML. That is the starting point to provide a virtual instance of HOL4 in the same run-time environment as Isabelle. Moreover, with an implementation of a virtual HOL4 kernel that operates on Isabelle/HOL terms and theorems, we can load substantial HOL4 libraries to make them Isabelle theories, but still disconnected from existing Isabelle content. Finally, we introduce a methodology based on the transfer package of Isabelle to connect the imported HOL4 material to that of Isabelle/HOL.

2012 ACM Subject Classification Software and its engineering → Interoperability; Theory of computation → Higher order logic

Keywords and phrases Virtualization, HOL4, Isabelle, Isabelle/HOL, Isabelle/ML

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.21

Supplement Material Our implementation is available online, it works with Isabelle2019 and the following development version of the official HOL4 repository.
https://github.com/immler/hol4isabelle
https://isabelle.in.tum.de/website-Isabelle2019
https://github.com/HOL-Theorem-Prover/HOL/commit/7e03303e51f

Funding *Fabian Immler*: This material is based upon work supported by the Air Force Office of Scientific Research under grant number FA9550-18-1-0120. Any opinions, finding, and conclusion or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force.

Acknowledgements Dagstuhl Seminar 18341 *Formalization of Mathematics in Type Theory* [2] hosted the working group "Interoperability of systems" (by Mario Carneiro), which provided an excellent environment to discuss early experiments and helped to structure this approach. We thank the HOL4 team (especially Michael Norrish) for support and accepting changes to HOL4.

1 Introduction: Interoperability of Theorem Provers

Suppose you chose Isabelle/HOL as your favorite theorem prover, like many other people did, e.g., in the Isabelle Archive of Formal Proofs (AFP) [3]. Unfortunately, by committing to one prover, you miss out on all of the great developments in others. For example, you cannot re-use the substantial work on the CakeML compiler [12], which is done in HOL4.

Interoperability between theorem provers, particularly HOL-based systems, has a long history (see also Section 7). But the problem has not been solved satisfactorily so far: none of the previous approaches has managed to import a huge project like CakeML in a scalable way and such that the result can be reused in a truly idiomatic manner in the target system.

Here we propose a novel and unorthodox approach to combine Isabelle/HOL and HOL4.

Main Ideas

We observe that HOL4 is designed with a modular and replaceable kernel, and that both provers are implemented in SML; Isabelle turns the underlying platform into a sophisticated environment of managed Isabelle/ML. The main ideas are:

- Run HOL4 inside the run-time environment of Isabelle/ML.
- Replace the kernel of HOL4 by a kernel that acts as a proxy to the kernel of Isabelle/HOL.
- Keep imported HOL4 libraries unchanged, without connections to existing Isabelle/HOL libraries at first.
- Connect theory content via Isabelle's `transfer` package.

Challenges: HOL4 versus Isabelle/HOL

We briefly review aspects of HOL4 and Isabelle that are relevant for combining them in a single run-time environment. Isabelle/Pure [17] is a generic logical framework (minimal higher-order logic), and Isabelle/HOL [15] a big library of Isabelle (classical higher-order logic with many add-on tools). HOL4 [19] is a proof assistant specifically for classical higher order logic. Both Isabelle and HOL4 are implemented in SML and run atop Poly/ML [14].

HOL4 provides a small abstract interface to its logical kernel (the modules for types, terms, and theorems); this opens the possibility to choose between different kernels implementations. Isabelle's inference kernel provides abstract types and constructors similar to the ones required by the kernel interface of HOL4.

The general idea of our approach seems straightforward: We replace the kernel of one HOL-based system with the kernel of another HOL-based system. But the implementation is not trivial, there are both conceptual and engineering difficulties to master.

The *conceptual difficulty* concerns differences in how HOL4 and Isabelle maintain logical declarations (i.e. update signatures of theories): HOL4 keeps a table of declared types and constants in a global mutable state variable that changes linearly over time. In contrast, Isabelle operates on a universal context as a purely functional value, following the DAG-structure of theories and the block-structure of proofs. We address this by providing an ML environment in which global state is virtualized inside Isabelle's universal context.

The *engineering difficulty* concerns details about the mapping of HOL4 inferences to Isabelle/HOL: they must conform precisely to the behavior expected from the HOL4 kernel interface. There are some further side-conditions, like implicit state in HOL4 terms, and different policies for names of variables and constants.

Contributions and Findings

We provide a working implementation of a virtual ML environment for HOL4 (Section 2), that is well integrated in Isabelle's Prover IDE (Isabelle/jEdit) and manages global state implicitly (Section 3). We present the implementation of a kernel for the virtual HOL4 that acts as a proxy to Isabelle/HOL theories, theorems, terms, and types (Section 4). We further propose a methodology (Section 5) to connect the imported HOL4 formalization to existing libraries in Isabelle/HOL and illustrate this idea on a small example.

Our measurements (Section 4.3) indicate that this approach of combining the two provers is worth continuing towards big applications: The performance losses in virtualized HOL4 and the proxy to Isabelle inferences are quite small, with a constant slowdown on most of the basis library of HOL4.

2 Virtualization in Isabelle/ML and ML Environments

Isabelle/ML is based on Poly/ML, but it isolates programs from low-level access to the compiler and run-time system: instead of direct mutation of the toplevel environment, there are functional updates on a universal context [21, §1.1]: it contains all logical declarations, add-on data for proof tools, and the ML environment (ML types, values, signatures, structures, functors, infixes). This managed environment of Isabelle/ML imposes restrictions on user-space programs, but allows pervasive parallelism with live editing of a running program in the Prover IDE, including implicit "undo" of ML toplevel declarations.

Such *virtualization* of ML is possible thanks to special operations provided by Poly/ML, most notably `PolyML.compiler`: it augments the running program with new declarations in the static ML environment, and new evaluations on the run-time heap (using native machine-code). This has been integrated with the universal context management of Isabelle for theories and proofs. Isabelle commands like `ML` or `ML_file` augment the environment according to the structure of theory documents in a thread-safe manner, i.e. ML code within independent theories (according to their DAG structure) and proofs (according to their block structure) can run in parallel without conflicts.

We have added a further dimension of *named ML environments*, to support other ML applications within Isabelle (notably HOL4): the ML function `ML_Env.setup` takes a fresh name and some operations to turn source text into ML tokens and (optional) antiquotations. There are two predefined ML environments: `Isabelle` refers to regular Isabelle/ML in the context of Isabelle/Pure (with some token syntax extensions and antiquotations), and `SML` refers to official Standard ML starting from the initial basis (without syntax add-ons).

The meaning of Isabelle commands like `ML_file` has been modified to depend on the context option `ML_environment`: it specifies the names of input and output environments in the form "env_1`>`env_2" (where "env`>`env" may be abbreviated by "env" alone). This allows to build up ML modules in independent name spaces, and to move material between them on demand. For example, the Isabelle/ML operation `writeln` for managed output of messages (with optional Prover IDE markup) can be made accessible in plain SML like this:[1]

```
declare [[ML_environment = "Isabelle>SML"]]
ML "val println = writeln"
```

That covers the static phase of ML declarations, but there is also the dynamic phase for program evaluation. To fit this into the purely functional context model of Isabelle, each ML execution context has a private (thread-local) variable to access its Isabelle context. The system provides the initial value, which may be changed by the running program via `Context.>>` of type `(context -> context) -> unit`. Afterwards, the system takes back the result. Thus old-style ML toplevel scripts with implicit mutation become plain functions on the context. Here is an example for Isabelle/ML:

[1] `TextIO.print` is available in the `SML` environment, too, but its output shows up on `stdout` and is not easily accessible to end-users in the Prover IDE. It is also not possible to "undo" such physical output.

```
fun declare_const name ty =
  Context.>> (Context.map_theory (Sign.add_consts [(Binding.name name, ty, NoSyn)]));
declare_const "c" propT;
declare_const "d" (propT --> propT);
```

As the effect of updating the context by `Context.>>` is managed by Isabelle, going back to an earlier situation in the theory means that the updates are still absent. So we get a form of implicit "undo", simply by returning to an old version of the (immutable) context.

3 ML Environment for HOL4

We use `ML_Env.setup` from Section 2 to define a fresh ML environment with the name "`HOL4`", hereafter referred to as the HOL4-environment.

▶ **Definition 1** (HOL4-environment). *The named ML environment "`HOL4`" in Isabelle.*

Subsequently we describe our setup of the HOL4-environment. Its initial basis is augmented to turn `ref` cells into variables managed in the Isabelle context (Section 3.1 and 3.2). Moreover, the lexical syntax is changed to support HOL4 quotations of terms and types (Section 3.3).

3.1 Global State

SML provides a special type `'a ref` for mutable reference cells pointing to values of type `'a`. (There is also the `Array` structure, for vectors of `ref` cells, but that is unused in HOL4.) The main operations on `'a ref` are `ref: 'a -> 'a ref` to initialize a new cell, `! : 'a ref -> 'a` to get the cell's content, and infix `:= : 'a ref * 'a -> unit` to change the cell's content. We imitate that in our structure `Context_Var`: it provides type `'a var` and operations `new`, `get`, `put` with analogous signatures. It is implemented via a data slot in the Isabelle context [21, §1.1.4], holding a map from integers to the universal sum type in SML. The operation `new: 'a -> 'a var` allocates a new index in the table and provides type-safe injections and projections for stored values of the particular type `'a`.

Now the meaning of HOL4 programs shall be changed to refer to this managed variable space, whenever `'a ref` operations are seen, but this type cannot be redefined in SML user-space. Since we manage the ML environment anyway, we simply map the name "`ref`" for types and values to our counterparts `Context_Var.var` and `Context_Var.new`, respectively. The other operations can be redefined by conventional declarations in SML.

A remaining problem is the use of `ref` as a datatype constructor in pattern matching, instead of `!` as selector. To keep our language manipulation simple, we eliminated the (rare) uses of that feature of SML in the HOL4 repository: there was no problem with rewriting, e.g., `fun lookup (ref v) = v` manually to `fun lookup r = !r`.

In summary, the module `Context_Var` manages the overall state of *all* context variables of the running ML program: henceforth it can represent the global "mutable" application state within the HOL4-environment.

▶ **Definition 2** (HOL4-state). *The value of the table in* `Context_Var` *in a given universal context is called the* HOL4-state.

3.2 Local State

The above approach, which maps all uses of `ref` to `Context_Var`, is functionally correct, but there is a catch regarding performance and memory consumption. Conventional `ref` cells are subject to garbage collection in ML, and do not need an explicit operation to free

allocated memory. In contrast, variables that were allocated once via `Context_Var.new` remain accessible in the context and will not be garbage-collected automatically.

This is particularly problematic for strictly local program variables, i.e., reference cells that are private to particular function invocations, and typically used to simplify or speed up the implementation via imperative features. Such ad-hoc variables do not survive termination of the function, and are better not made persistent in the Isabelle context.

Following this observation, we distinguish *local state* variables from global state variables that are managed in the context. This works by marking the variables in the original HOL4 sources, using the type `Uref.t` that is merely a clone of `ref` in HOL4. Now we can distinguish the two kinds of references in the virtualized Isabelle environment: unmarked `ref` becomes our `Context_Var.var`, and `Uref.t` becomes `Unsynchronized.ref` of Isabelle/ML [21, §0.7.9].

How to distinguish the two kinds of variables in the vast body of HOL4 sources? Our pragmatic approach is based on run-time profiling. For each static occurrence of `Context_Var` its number of dynamic allocations is reported. Then we inspected the worst "memory leaks" to judge their role in the program: the result is recorded in the official HOL4 sources.

3.3 Quotation Filter

HOL4 extends SML syntax with *quotations* to allow embedding of logical entities (types and terms) with their own syntax into ML.[2] HOL4 quotations come in different forms, we will illustrate the concept only for terms. A *term quotation* consists of a string delimited by two single back-quotes, e.g., ``` ``string`` ```. The HOL4 `QuoteFilter` expands this to ML source `Parse.Term [QUOTE "string"]`. It means that the string is parsed at run-time at the spot where this is inlined into the ML source, using the syntax of the implicit theory.

Our HOL4-environment in Isabelle should support quotations, too, so we include the `QuoteFilter` directly into it. We also want to use the rich capabilities of the Isabelle Prover IDE: this requires original source positions passed on to the generated ML text. The Isabelle setup of `PolyML.compiler` (Section 2) turns results from static analysis by the ML compiler into markup that Isabelle/jEdit presents to the user as colors, popups, hyperlinks etc.

`QuoteFilter` is implemented with the lexical analyzer generator ML-Lex [1]. We made minor modifications to that in the HOL4 repository, to have it return precise position information (together with the expanded text). Consequently, the Isabelle/HOL4 source text provides ML IDE annotations both for SML and the result of inlined quotations (but not inside the HOL4 term language, unlike Isabelle). The example in Figure 1 illustrates this Prover IDE experience: The embedded HOL4 term `t` has ML type `KernelTypes.term`; this can be inspected by hovering with the mouse cursor over `t`.

3.4 Implicit "Undo" of Changes in HOL4-state

We can now illustrate the advantage of implicit "undo" with the HOL4-state. Assume you declare a constant `foo`, realize that you spelled it wrong and want to redeclare it as `fop`. When interacting with the HOL4 toplevel, you need to keep the global state of constants in mind, delete `foo` (make it inaccessible in the term language), and introduce `fop` like this:

```
> Theory.new_constant("foo", Type.bool);
> Theory.delete_const "foo";
> Theory.new_constant("fop", Type.bool);
```

[2] This is similar to *antiquotations* in Isabelle, but the terminology is reversed, because the outer source is the Isabelle theory and proof language, which may *quote* ML, which may *antiquote* the term language.

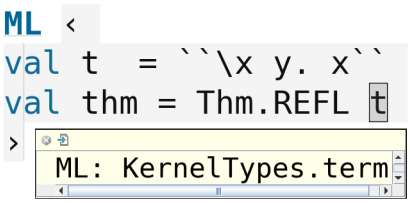


Figure 1 A Quotation in the HOL4-environment and IDE markup in Isabelle/jEdit.

With virtual state management by Isabelle, however, it suffices to edit the document, changing `foo` to `fop` and the previous declaration of `foo` simply disappears (see Figure 2).

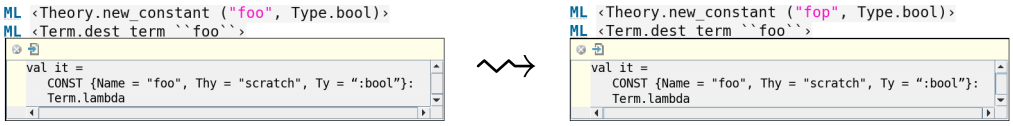


Figure 2 Implicit "undo" of operations on virtual state in HOL4 in Isabelle/jEdit. After editing `foo` to `fop`, `fop` is no longer registered as a constant `CONST`, but is parsed as a variable `VAR`.

3.5 The Standard Kernel in the Virtualized Environment

To test that everything is properly set up, we load the original HOL4 sources (with the standard kernel) in the previously described HOL4-environment. So in this case, HOL4 can be seen as an isolated application running in the HOL4-environment, without any connection to the logic of Isabelle/HOL whatsoever. We did this for the following build-sequences:

- `core`: e.g., natural numbers, datatypes, lists, and bossLib
- `more`: e.g., integers, topology, n-bit vectors
- `large`: e.g., real numbers, probability theory, temporal logic, floating point numbers

Figure 4 shows performance figures of virtualized HOL4 vs. original HOL4. Broadly summarized, virtualized HOL4 is about 1.5 times as slow.

4 An Isabelle/HOL Kernel for HOL4

Now that we have demonstrated that the HOL4-environment provides a suitable environment to run HOL4 in Isabelle, let us take a look at what is required to have HOL4 produce actual Isabelle/HOL theorems. This requires an implementation of the actual logical kernel, i.e., modules for types, terms, and theorems, which we describe in Section 4.1. But it also requires modifications to the theory management of HOL4 to properly integrate in the virtualized HOL4-environment with Isabelle's theory management, which we describe in Section 4.2. We report on performance measurements in Section 4.3.

4.1 Logical Kernel

HOL4 and Isabelle/HOL both follow the LCF-approach, which means they define an abstract datatype of theorems with inference rules as (type-safe) operations [4]. Note that Isabelle/HOL's types, terms, and theorems are actually implemented in Isabelle/Pure.

The *HOL4-Kernel* is the collection of ML modules for types, terms, and theorems in HOL4. HOL4 prescribes an interface (ML signatures) to the HOL4-Kernel, which makes it possible to select different implementations of the HOL4-Kernel at compile time. HOL4 comes with a *standard* HOL4-Kernel, where terms are represented with de-Bruijn indices and explicit substitutions, as well as an *experimental* HOL4-Kernel with named bound variables.

In the subsequent implementation of our *Isabelle* HOL4-Kernel, types, terms, and theorems of the HOL4-Kernel interface are implemented by their counterparts in Isabelle/Pure.

4.1.1 Types

Besides constructor names, Isabelle's type language only differs from the standard HOL4-Kernel in that it is many-sorted and includes schematic type variables. Therefore, all types produced by our kernel have the base sort `HOL.type` and do not include schematic type variables. So the type structure is mainly a copy of the standard HOL4-Kernel with constructors replaced and some small adaptations.

The Isabelle/Pure theorem module does not use these (unchecked) types directly, but rather operates on values of the abstract type `ctyp`, which represents types that are well-formed wrt. some theory. Certifying types is an expensive operation, and since the Isabelle HOL4-Kernel should never need to produce malformed types, it would be beneficial to use certified types as our underlying type representation.

But unfortunately this is impossible, because the HOL4-Kernel (and subsequently all of HOL4) requires that `hol_type` is an ML equality type. Isabelle/Pure's abstract type `ctyp` does not satisfy this requirement and it is unrealistic to remove this requirement from the HOL4-Kernel.

There are two occurrences in the Isabelle HOL4-Kernel where certification of types is necessary: The first occurrence is instantiation of type variables, `INST_TYPE` in HOL4 that maps to `Thm.instantiate_frees`). In this case it is reasonable to expect that the size of these types is small, so that we can ignore this fine point. The second occurrence is construction of variables `Term.mk_var : (string * hol_type) -> term`. Variables are constructed so frequent that re-certification can be prohibitively expensive. We work around this by introducing a cache where already certified types can be looked up.

4.1.2 Terms

The HOL4-Kernel interface fixes an abstract interface to well-typed terms. In Isabelle/Pure, well-formed and well-typed terms are an abstract subtype `cterm` (for certified `term`) of a datatype of preterms[3]. Figure 3 compares the representation of `preterm`s in Isabelle/Pure and terms in the standard HOL4-Kernel. Constants `Const` and free variables `Free/Fv` are constructed from a name and a type, bound variables `Bound/Bv` are represented with de-Bruijn indices. In Isabelle/Pure, λ-abstraction `Abs` takes information on how to display the bound variable with a string, the type of the bound variable, and a body. The standard HOL4-Kernel maintains the invariant that `Abs` only occurs with a free variable `Fv` as first argument, thereby representing the same information as Isabelle/Pure. Function application of function `f` and argument `x` is written as infix operation `f $ x` or combination `Comb`. `Var` in Isabelle represents schematic (unifiable) variables, but this is not needed for HOL4.

[3] This actually is the datatype `term` in Isabelle, but to avoid confusion, we call it `preterm` here.

```
datatype preterm =
  Const of string * typ
| Free  of string * typ
| Bound of int
| Abs   of string * typ * preterm
| $     of preterm * preterm
| Var   of indexname * typ
```

(a) Preterms in Isabelle/Pure.

```
datatype term =
  Const of kernelid * hol_type
| Fv    of string * hol_type
| Bv    of int
| Abs   of term * term
| Comb  of term * term
| Clos  of term Subst.subs * term
```

(b) Internal terms in the standard HOL4-Kernel.

Figure 3 Different term representations in Isabelle and HOL4.

HOL4's standard kernel has an explicit constructor `Clos` for closures, terms with an environment attached to them. For rewriting-heavy applications (e.g., the CakeML bootstrapping), `Clos` might be performance-critical. Nevertheless, we decided to ignore this feature for the moment, because e.g., the experimental kernel does not feature closures, either. Should one wish to achieve the same asymptotic complexity for rewriting with explicit closures, one could add a special Let-construct to Isabelle/HOL. Pattern-matches on `preterm`s in the Isabelle HOL4-Kernel would then need to introduce a special case that tests for the presence of this special Let constant (just like the standard HOL4-Kernel has a special case for `Clos`).

While the Isabelle/Pure interface to `cterm` is rather minimal, it exposes some primitives for building abstractions and applications without having to re-certify the result. This allows us to base our implementation of the HOL4-Kernel term structure on certified terms, avoiding the expensive operation of certification as much as possible.

One slight complication was that, at the beginning of our work, the type of terms was declared as an `eqtype` in the HOL4-Kernel signature, and thus could not be instantiated with an abstract type. The HOL4 developers had already started to work (with an independent motivation) towards removing the `eqtype` constraint, and this removal is now completed.

We encountered another problem: the HOL4-Kernel sometimes produces malformed terms, in particular terms containing loose bound variables. These terms result from `break_abs`, which destructs an abstraction without turning the variable bound under the abstraction into a free variable. We work around this by using free variables with special names to represent the loose variables. These uniquely named variables are also useful to efficiently destruct abstractions in the regular way (i.e. by turning the bound variable into a free variable), which may otherwise involve renaming, and to ensure that Isabelle/Pure does not rename variables using its own convention, which is different from that of the HOL4-Kernel.

We cannot pattern-match on the abstract type `cterm` but would like our implementation to stay close to that in the standard HOL4 kernel, which heavily uses pattern matching on terms. We therefore match on the underlying `preterm` of a `cterm` and carry out the actual operation on the result of destructing the `cterm` according to the matched pattern. To illustrate this technique, here is part of the implementation of the operation `trav`, which applies a function `f : cterm -> unit` to all constants and free variables in a term.

```
fun trav f ct =
  let fun trv (Free _) ct = f ct
        | trv (Rator $ Rand) ct =
          let val (cRator,cRand) = dest_comb ct
          in (trv Rator cRator ; trv Rand cRand)
          end
  ...
  in trv (preterm_of ct) ct end
```

Here we access the underlying preterm of `ct` using the Isabelle/Pure function `preterm_of`, which is cheap, and then give both to the internal function `trv`, which pattern matches on the term and either applies `f` at the appropriate places or destructs the `cterm` according to the matched pattern with `dest_comb`.

4.1.3 Axiomatization of HOL

The HOL4-Kernel axiomatizes higher order logic. For the Isabelle HOL4-Kernel, we obviously do not want to add new axioms, but rather map calls that axiomatize to existing constants, axioms, and theorems in Isabelle/HOL.

The HOL4-Kernel has builtin type operators for functions `->`, booleans `bool`, and an inductive type `ind`, which we map directly to the corresponding type operators from the axiomatization of Isabelle/HOL for functions ⇒, booleans *bool*, and an inductive type *ind*.

The HOL4-Kernel has builtin constants for equality `= : 'a -> 'a -> bool`, Hilbert choice `@ : ('a -> bool) -> 'a`, and implication `==> : bool -> bool -> bool`. The axiomatization of Isabelle/HOL also contains equality *(=) :: 'a ⇒ 'a ⇒ bool*, implication *(⟶) :: bool ⇒ bool ⇒ bool*, and Hilbert choice *Eps :: ('a ⇒ bool) ⇒ 'a*, so we can map to those directly, as well.

The HOL4-Kernel introduces axioms for defined constants (`T`, `F`, `ONE_ONE`, `ONTO`):

```
("BOOL_CASES_AX", ``!t. (t=T) \/ (t=F)``)
("ETA_AX",        ``!t:'a->'b. (\x. t x) = t``)
("SELECT_AX",     ``!(P:'a->bool) x. P x ==> P ($@ P)``)
("INFINITY_AX",   ``?f:ind->ind. ONE_ONE f /\ ~ONTO f``)
```

Without extending the set of axioms in Isabelle/HOL, we map those axioms to theorems that we proved explicitly in Isabelle/HOL.

4.1.4 Theorems

The HOL4-Kernel theorem module modifies the internal representation of theorems in a soundness-critical way. In contrast to that, the Isabelle/Pure-based implementation simply defers operations to (trusted) Isabelle/Pure primitives.

Usually, Isabelle/Pure inferences cannot be used directly, but require some adaptation of interfaces. This is because of the distinction between meta-logic Isabelle/Pure and object-logic Isabelle/HOL. For example, the HOL4-Kernel primitive `MK_COMB : thm -> thm -> thm` is supposed to yield a theorem `f x = g y` from theorems `f = g` and `x = y`. Isabelle/Pure provides a similar inference `Thm.combination : thm -> thm -> thm`, but for meta-equality. I.e., it produces *f x ≡ g y* from theorems *f ≡ g* and *x ≡ y*. The Isabelle/HOL axiomatization states that HOL-equality *(=)* reflects Pure-equality *(≡)*, so we can insert inference steps to convert theorems with Pure-equality to (and from) theorems with HOL-equality.

Overall, we do not use the builtin unification of Isabelle/Pure, but always compute explicit instantiations, hoping that this is the most efficient implementation.

4.2 Theory Management

HOL4 uses the dedicated Holmake tool to manage dependencies of source files. Moreover, it can compile theory files from script files: this requires special attention when trying to incorporate this in the HOL4-environment and cooperate with the Isabelle HOL4-Kernel.

4.2.1 HOL4-Scripts and HOL4-Theory Files

Theories in HOL4 are structured along a concept called *theory segments*. A segment records logical declarations like types, constants, and theorems, together with pointers to parent segments. The theory represented by a segment is the union of all the logical declarations of the segment and its parents. A theory segment is constructed in different stages:

- One starts from a so-called *script*. A script contains all the ML-declarations that define types, constants, prove theorems, or e.g., augment syntax.
- When compiling a script, all changes that the script makes w.r.t. to the current (logical) theory are recorded and saved in a special file-format. Compilation of a script will also generate *theory files*.
- Theory files are ML modules that contain all the information required to load the recorded changes and apply them to the current (logical) theory.

Note that re-importing a HOL4 theory from the file-system does *not* reconstruct theorems by kernel inferences, instead it trusts the imported statement with an oracle. We do not want to reproduce this part of the workflow in Isabelle/HOL, in particular because we do not want to increase the trusted code base by theorem import (and essentially all of HOL4).

Instead we remember the theorem values that are created when running the script. HOL4 offers a hook that allows users to register custom code to be called upon exporting a theory, and we use this to store the theorems as abstract ML values in the universal Isabelle context.

We provide a small wrapper around HOL4's module that reads theories from the file-system. This wrapper does not assume theorems via an oracle, it rather looks up the theorem values that were previously stored in the Isabelle context.

In the HOL4 system, scripts are run in a separate process, and the only artifacts that they produce are the generated theory files. This means that in our case, after running a script, the HOL4-state needs to be reset: the script modified the underlying HOL4 theory, but these changes are not supposed to persist. In order to do so, we simply remember the HOL4-state before compiling the script in Isabelle/ML and update the HOL4-state afterwards to the previously remembered state.

4.2.2 Holmake

Holmake is the main tool to manage dependencies of script files, theory files and other ML files for HOL4. Upon invocation in a directory containing HOL4 source files, Holmake computes dependencies between files, and compiles and runs plain ML code, proof scripts, and generated theory files.

We can compile and run Holmake in the `HOL4` ML environment, but its overall setup is – due to the previous discussion about storing theorems when running scripts – too alien for us to use it in the HOL4-environment. Instead, we write custom code that emulates the behavior of Holmake reasonably well. Our emulation builds on a part of Holmake, the *Holdep* tool. From the output of Holdep, we recurse over the dependencies. With additional dependencies for Theory files (they depend on Script files), our emulation is sufficiently close so that it "does the right thing"[4] for a large part of HOL4's source directories.

On several occasions, HOL4 saves a "heap", i.e., the global state of the Poly/ML process after loading a number of SML modules and theories. In our virtual environment, this simply amounts to keeping the HOL4-state instead of resetting it to the previous value.

[4] Quoting a comment in the implementation of Holmake.

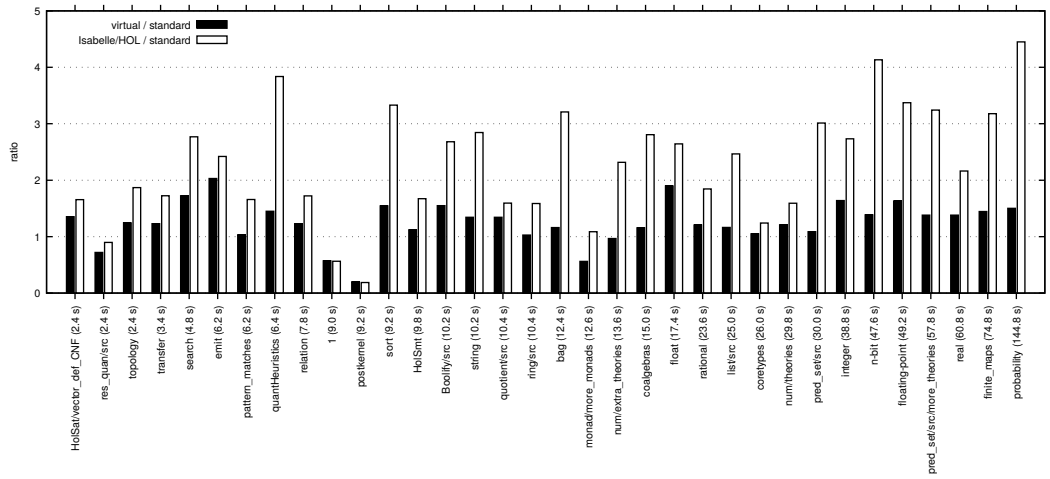


Figure 4 Performance measurements: HOL4 source directory on the x-axis, sorted by elapsed time (in parentheses) to build that directory with standard HOL4. Bars show the time (relative to standard HOL4) to build that directory with virtualized standard HOL4-Kernel (black) and the Isabelle HOL4-Kernel (white).

4.3 Performance Measurements

In this section we report on performance measurements. We investigate whether the overhead incurred by virtualization and adapting kernel interfaces is reasonably moderate and how well it scales with the size of the application.

In Figure 4, we compare the elapsed time for building theories in HOL4 source directories from the `core`, `more`, and `large` build sequences. We exclude the directories that take less than 2 seconds to build in standard HOL4. The reported times are the minimum out of 5 measurements for each directory. The experiments were run (single threaded, because Isabelle and HOL4 have different schemes for parallelization) on a laptop computer with Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz and 32 GB of RAM, running Windows 10.

We observe that the virtual HOL4-Kernel takes about 1.5 times as long as standard HOL4. We interpret this as an indication that we have a good handle on the management of global and local state (Sections 3.1 and 3.2).

Some directories build faster with the virtual HOL4-Kernel. This is likely due to different bootstrapping in the virtual HOL4-environment: more dependencies are already pre-loaded. Another reason could be fewer IO operations in the HOL4-environment, because we virtualize access to the file-system by in-memory data lookup and storage.

The final observation concerning Figure 4 is that the slowdown of the Isabelle HOL4-Kernel is never by more than a factor of 4.5 and usually lies between one and three. We believe that this is a moderate overhead for the adaptations between inferences of Isabelle/Pure and the HOL4-Kernel interface (described in Section 4). Moreover this overhead seems to be constant w.r.t the size of the development, so we expect it to scale to even bigger applications.

Let us now compare our approach of transporting theorems from HOL4 to Isabelle/HOL with OpenTheory (see also Section 7). With the OpenTheory approach, HOL4 scripts are run with a HOL4-Kernel that produces OpenTheory files (`.art`). These files are post-processed with the OpenTheory tool (`opentheory info --article`), which, e.g., deletes unwanted constants. The resulting file can then be imported into Isabelle/HOL, once the importer is set up with information on how to map HOL4 type operators and constants

Table 1 Performance in comparison with OpenTheory. Absolute time and time relative to standard HOL4 (row 1) are reported for transporting HOL4 theories `relation` and `real_topology` to Isabelle/HOL via OpenTheory (rows 2.1-2.3) and our Isabelle HOL4-Kernel (row 3).

		`relation`		`real_topology`	
		absolute [s]	relative	absolute [s]	relative
1	standard HOL4	1.8	1.0	68.4	1.0
2.1	HOL4 OpenTheory kernel (`.art`)	10.2	5.7	546	8.0
2.2	`opentheory info --article` (`.ot.art`)	31.2	17.3	2416	35.3
2.3	Isabelle OpenTheory import	0.9	0.5		
3	Isabelle HOL4-Kernel	6.1	3.4	310	4.5

to their Isabelle/HOL counterparts. In Table 1, we report our performance measurements for exporting and importing the HOL4 theory `relation` and `real_topology`. We chose `relation` because it has few dependencies and therefore allowed us to set up the OpenTheory importer with moderate effort. We chose `real_topology` in order to investigate performance on a large theory: regarding build time, it is the largest theory in the basis library.

The actual import is faster than the original build of the theory `relation`. But producing OpenTheory files is slower than our Isabelle HOL4-Kernel. Post-processing of OpenTheory files is very expensive, it takes more than 17 times as long for the small `relation` theory and even 35 times as long for the large `real_topology` theory.

4.4 Debugging

When attempting to build HOL4 using our Isabelle HOL4-Kernel, we came across many failures that were related either to implementation errors or to unexpected behavior of the Isabelle/Pure primitives we use. Debugging these failures was often a challenge: Errors frequently occurred far from their root cause, especially when program flow in HOL4 is controlled by exceptions. In order to help with the debugging process, we implemented yet another kernel that performs every operation using both the standard HOL4-Kernel and our Isabelle HOL4-Kernel simultaneously, comparing their results. This yields an error at the earliest point where the behavior of the standard kernel and the Isabelle kernel diverge and therefore points directly to discrepancies in the implementation (together with concrete arguments that caused the bad behavior).

5 Transfer

In order to keep the Isabelle HOL4-Kernel as simple and maintainable as possible, we do not make any attempts at transforming the imported definitions or theorems to somewhat more idiomatic concepts in Isabelle/HOL. For example, apart from the axiomatization in Section 4.1.3, we do not map types/constants from HOL4 to existing types/constants in Isabelle/HOL. We also do not use the Isabelle/HOL datatype package, but simply use the constructions performed by HOL4.

Overall, we get a completely separate formalization of closely related concepts. E.g., both Isabelle and HOL4 define natural numbers and lists. We realign those in a post-hoc fashion, and Isabelle's `transfer` package [5, 13] is a powerful, flexible, and efficient tool perfectly suited for these needs.

Subsequently, we propose an approach that allows the user to obtain an idiomatic Isabelle/HOL formalization from the imported HOL4 libraries. This requires some user interaction, but arguably there has to be some human interaction to judge what an "idiomatic" definition looks like. We provide infrastructure to make this process as comfortable as possible. In particular, we enable the user to mix HOL4 syntax and Isabelle/HOL syntax in order to state and prove theorems that relate Isabelle/HOL concepts to HOL4 concepts.

The running example to illustrate our approach will be lists. The HOL4 formalization defines the type of lists as a datatype:

```
Datatype.Hol_datatype ‘list = NIL | CONS of 'a => list‘
```

In the Isabelle HOL4-Kernel (Section 4), we adopt the naming scheme that identifiers `id` from a HOL4 theory segment `seg` are mapped to Isabelle identifiers *`seg___id`*. This means that in the Isabelle/HOL foundation, the above HOL4-datatype definition results in the definition of a type *`'a list___list`* and constants *`list___NIL`*, *`list___CONS`* (the datatype definition is in the segment `list`).

```
typedecl 'a list___list
consts list___NIL::'a list___list
consts list___CONS::'a ⇒ 'a ⇒ 'a list___list
```

In the rest of this section, we show how to relate these constants to the "idiomatic", existing datatype constructors of lists in Isabelle/HOL:

```
datatype 'a list = Nil | Cons 'a "'a list"
```

5.1 Mixing HOL4 and Isabelle Entitites

We first describe how a user can mix HOL4 and Isabelle-syntax. This takes inspiration from an experiment by Hupel [6] that allows the embedding of ML values into a formal context. We provide special syntax for Isabelle, that allows the user to write *`HOL4‹expr›`* in inner syntax, and *`expr`* will be parsed by the HOL4 parser and return the corresponding Isabelle/HOL term. For example, a property *`P`* of two-element HOL4-lists can be expressed as *`P (HOL4‹length [x, y]›)`*, which will be parsed as *`P (list___length (list___CONS x (list___CONS y list___NIL)))`*.

Theorems produced by the Isabelle HOL4-Kernel do not even show up in the Isabelle/Isar namespace, they are only stored as ML-values in the universal context in the HOL4-environment. In order to comfortably refer to HOL4 theorems in Isabelle/Isar, we provide an attribute *`[[hol4_thm segment.THEOREM]]`* that refers to the theorem `THEOREM` from the HOL4-segment `segment`.

5.2 Transfer Rules

The main tool for proving theorems along isomorphisms is the `transfer` package [5, 13]. The central element for setting up the `transfer` package are *transfer rules*. For the purposes of this presentation, transfer rules are of the form *`R c d`* for relations *`R :: 'a => 'b => bool`* and constants *`c :: 'a`*, *`d :: 'b`*. For simplicity, we assume that *`R`* is bi-total and bi-unique, i.e., the graph of a bijection between *`'a`* and *`'b`*, but the `transfer` package is sufficiently flexible to deal with partial and quotient relations as well. We say that *`R c d`* is a transfer rule for constant *`d`*.

Relators are used to construct relations for compound types, in particular the function relator with infix syntax `===>` relates functions *`f`* and *`g`* that yield *`S`*-related results *`f x`*, *`g x`* for *`R`*-related arguments *`x`*, *`y`*. Written as a transfer rule: *`(R ===> S) f g`*.

Given a set of transfer rules and a theorem, the `transfer` package looks up transfer rules for each of the constants that occur in the theorem and composes them (recall that the transfer relations encode bijections) to obtain a theorem for an isomorphic (along the transfer rules) theorem. E.g., given a transfer rule `(R ===> (=)) P Q` for predicates `P :: 'a => bool` and `Q :: 'b => bool`, a transfer rule `R c d` to transfer a constant `d::'b` to `c::'a`, and a theorem `Q d`, the transfer package will produce the theorem `P c`. To clarify the intuition, `Q` and `d` will be constants imported from HOL4, and `P` and `c` related idiomatic constants in Isabelle/HOL.

The basis of our setup consists of transfer rules for all of the constants defined in the HOL4 `bool` theory. To give an example, the transfer rule for universal quantification relates the HOL4 all-quantifier `!` with the Isabelle/HOL all-quantifier `All` for `(A ===> (=))`-related predicates, given a bijection `A`.

```
lemma [transfer_rule]: "((A ===> (=)) ===> (=)) All HOL4<(!)>" if "bi_total A"
```

We prove similar rules for e.g., conjunction, implication, negation, True, and False. The proofs are straightforward, because the definitions in HOL4 and Isabelle/HOL both follow the axiomatization from Section 4.1.3.

5.3 Setting up Transfer for Lists

We now proceed with setting up transfer rules for HOL4-lists. The HOL4 datatype package constructs (among others) an induction theorem:

```
list_INDUCT = ‘!P. P [] /\ (!t. P t ==> !h. P (h::t)) ==> !l. P l‘
```

In the Isabelle/HOL foundation, we can look up this theorem with the attribute `[[hol4_thm <list.list_INDUCT>]]`. The resulting theorem looks like this:

```
bool___! (λP. bool___/\ (P list___NIL)
  (bool___! (λt.  P t ⟶ bool___! (λh.  P (list___CONS h t)))) ⟶ bool___! P)
```

It can be transferred (automatically, using the `untransferred` attribute) along the transfer rules for the constants from the HOL4 `bool` theory (`bool_!` and `bool___/\`) in order to prove a proper Isabelle/HOL induction rule, mixing in HOL4-syntax for the HOL4-constants `NIL` and `CONS`:

```
lemma hol4_list_induction[case_names NIL CONS, induct type]:
  "P x" if "P HOL4<NIL>" and "(⋀x xs. P xs ⟹ P (HOL4<CONS> x xs))"
using [[hol4_thm listTheory.list_INDUCT>, untransferred, of P x]] by simp
```

In order to establish a connection between lists in Isabelle/HOL and lists in HOL4, we define a function (in Isabelle/HOL) from Isabelle/HOL lists to their HOL4-counterparts.

```
fun convert_list :: "'a list ⇒ 'a list___list"
where "convert_list [] = HOL4<NIL>"
    | "convert_list (x#xs) = HOL4<CONS x> (convert_list xs)"
```

With the induction rule for HOL4-lists, as well as injectivity of `CONS` and disjointness of `NIL` and `CONS` – those rules are also provided by HOL4 – we can easily prove injectivity and surjectivity of `convert_list`. Therefore the relation `(λx y. convert_list x = y)` is bi-total and bi-unique and it can be used as a transfer relation for lists of elements of the same type.

The `transfer` package provides some more support, in particular automation to set up parametric transfer rules:

```
setup_lifting ‹Quotient (=) convert_list convert_list' (λx y. convert_list x = y)›
  parametric ‹(list_all2 A ===> list_all2 A ===> (=)) (=) (=)›
```

This defines a relator `rh4_list::('a ⇒ 'b ⇒ bool) ⇒ 'a list ⇒ 'b list ⇒ bool`, where `rh4_list A is hs` expresses that `is` and `hs` are of the same shape and their elements are (pointwise) in relation `A`. With this setup, we can prove transfer rules for `NIL` and `CONS`

```
lemma [transfer_rule]: rh4_list A Nil (HOL4‹NIL›)
lemma [transfer_rule]: (A ===> rh4_list A ===> rh4_list A) Cons HOL4‹CONS›
```

With these rules proved, the `transfer` package can be used to automatically transfer every theorem that involves `NIL, CONS` and constants from HOL4 theory `bool`.

5.4 Primitive Recursive Definitions

HOL4 does not expose generic recursors for primitive recursive functions. A convenient way to transfer constants that are defined by primitive recursion in HOL4 is to define (in Isabelle/HOL) a recursor for HOL4 lists in terms of the recursor `rec_list` for Isabelle/HOL lists. The command **lift_definition** provides infrastructure to define constants in terms of isomorphic constants (here the isomorphism is between between `list` and `list___list` and has been set up by the previous **setup_lifting** command).

```
lift_definition rec_hol4_list::
  "'a ⇒ ('b ⇒ 'b list___list ⇒ 'a ⇒ 'a) ⇒ 'b list___list ⇒ 'a"
  is rec_list::"'a ⇒ ('b ⇒ 'b list ⇒ 'a ⇒ 'a) ⇒ 'b list ⇒ 'a"
  parametric list.rec_transfer .
```

Then one only needs to express the respective constants in terms of this recursor, e.g., for appending lists:

```
lemma "append xs ys = rec_list ys (λ x _.  Cons x) xs"
lemma "HOL4‹APPEND› xs ys = rec_hol4_list ys (λ x _.  HOL4‹CONS› x) xs"
```

These lemmas follow directly from the definition of the recursors and the involved functions. Then the transfer rule for `APPEND` (`"(rh4_list A ===> rh4_list A ===> rh4_list A) append (HOL4‹APPEND›)"`) can be proved automatically by the `transfer_prover` (after unfolding the above equivalences), because there are transfer rules for each of the involved constants.

5.5 Example: Transfer Theorems

We demonstrate the possibility of transferring theorems from HOL4 to Isabelle/HOL and back on a derived example. Assume that someone proved `FERMAT` in HOL4, which we simulate by an axiomatization in the virtual HOL4:

```
val FERMAT = Theory.new_axiom ("FERMAT",
``!a b c n. SUC (SUC 0) < n ==> ~(SUM (MAP (\x. SUC x ** n) [a; b]) = SUC c ** n)``)
```

A significant result! Because we proved transfer rules for all of the constants occurring in the theorem statement, we can import this result with a single invocation of `untransferred`.

```
lemma fermat: "Suc (Suc 0) < n ⟹ (∑x←[a, b].  (Suc x) ^ n) ≠ Suc c ^ n"
  using [[hol4_thm fermatTheory.FERMAT, untransferred]] by simp
```

We can even communicate this result back to the virtual HOL4 system: First of all, we need to prove the lemma (in Isabelle/HOL) in a HOL4-friendly format. Again, we have transfer rules for all of the constants, so a single invocation of `transfer` allows us to prove the lemma.

```
lemma fermat_hol4: "HOL4‹! a b c n.  SUC (SUC 0) < n ==>
  ~(SUM (MAP (\x.  SUC x ** n) [a; b]) = SUC c ** n)›"
by transfer (use fermat_isabelle in simp)
```

Then `val fermat_hol4 = @{thm fermat_hol4}` can be used as a regular theorem value in the virtual HOL4 environment and prove the round-tripped theorem `FERMAT2`.

```
val FERMAT2 = store_thm("FERMAT2", ‘‘! a b c n.
    SUC (SUC 0) < n ==> ~(SUM (MAP (\x. SUC x ** n) [a; b]) = SUC c ** n)‘‘,
  METIS_TAC [fermat_hol4]);
```

5.6 Discussion: Manual Interaction

Our proposed approach clearly involves some amount of manual interaction. First, suitable definitions in Isabelle/HOL need to be identified or defined. Then, suitable transfer rules need to be proved for each of these definitions. But (and that is an important aspect), the amount of manual interaction is proportional to the number of constant definitions, and not to the size or number of results that one wants to transfer.

This manual effort is certainly well invested for larger applications, in particular because intermediate constructions need not be set up in order to transfer final results. For example, there is no need to set up transfer for all intermediate languages in the CakeML stack to transfer a final correctness theorem that involves only machine-code and CakeML-syntax.

6 Conclusion

Thanks to the proper setup of a virtual ML environment, we have managed the daunting task of taming mutable state in a large application program: HOL4. Many big and small problems had to be overcome; it is great to see such a collaboration between different systems already work so well. This effort has induced changes to the internals of both HOL4 and Isabelle, which were overseen by the respective experts and incorporated into their repositories.

All involved systems profit from this work: HOL4 has yet another kernel, and remaining issues of the emulation could point to HOL4 code that needs further improvement. Isabelle now has a systematic treatment of alternative ML environments, with user-defined static basis and token language. Since virtual HOL4 runs inside Isabelle/jEdit [20], we could even see that as a viable IDE for HOL4 in the near future, although its user community is still very content with more traditional vi and Emacs interfaces.

We imagine fruitful interoperability: For example, HOL4 tactics could be used for Isabelle proofs, or HOL4 users could work with Isabelle/HOL formalizations (e.g., from the AFP) in the HOL4-environment.

A full import of the CakeML project in Isabelle/HOL is still future work, but it could yield a much larger user-base for the CakeML formalization, when all the tools of HOL4 and Isabelle/HOL can be combined in a single environment. The material on CakeML by Hupel in the AFP [7] is already awaiting to be formally connected.

7 Related Work

In 1995, Slind implemented the TFL package [18] generically, such that the ML sources worked both for Isabelle/HOL and HOL4. A few years later, both sides were maintained independently and diverged: today Isabelle has still a legacy `recdef` command and HOL4 a substantially extended `Definition` function, both based on TFL. Despite its limited success in bridging the gap between Isabelle/HOL and HOL4, the TFL package shares the key idea of our approach to load original ML sources into the other proof assistant.

More conventional export and import facilities write internal data structures to the file-system (essentially a trace of the inference kernel and theory content) and load them into the other system. A notable example is the HOL(Light) to Isabelle converter: the first version by Skalberg and Obua [16] had scalability problems due to massive amounts of XML data written to a Unix file-system. This has been greatly improved by Kaliszyk and Krauss [10]: the HOL-Light standard library is loaded into Isabelle/HOL in a few minutes.

OpenTheory by Hurd [8] is a similar approach based on kernel traces, but its theory and proof representations follow a published standard format. This has been designed to cover all members of the HOL family, but this excludes Isabelle/HOL with its distinctive deviations in the primitive logic (e.g. support for type-classes with overloaded definitions). Consequently, the OpenTheory importer for Isabelle did not get beyond experimental state so far, and an exporter never worked out. We also see fundamental problems in scalability to really large libraries: the OpenTheory standard library [9] is rather small compared to applications seen today, e.g. in Isabelle AFP [3], or CakeML [12].

More ambitious export-import projects even attempt to bridge the gap between HOL-Light and Coq [11]. This introduces new questions on the logic, but the fundamental problems of scalability and systems engineering remain the same. It should be noted that our approach is closely related to the original idea behind LCF [4]: instead of handing around proof terms, we merely run a program in a controlled manner to get to the intended theory content.

References

1 Andrew W Appel, James S Mattson, and David Tarditi. A lexical analyzer generator for Standard ML. *Distributed with Standard ML of New Jersey*, 1989.

2 Andrej Bauer, Martín Escardó, Peter L. Lumsdaine, and Assia Mahboubi. Formalization of Mathematics in Type Theory (Dagstuhl Seminar 18341). *Dagstuhl Reports*, 8(8):130–155, 2019. doi:10.4230/DagRep.8.8.130.

3 Manuel Eberl, Gerwin Klein, Tobias Nipkow, Larry Paulson, and René Thiemann (editors). The Archive of Formal Proofs. Online Journal. URL: https://www.isa-afp.org.

4 M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

5 Brian Huffman and Ondřej Kunčar. Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs*, pages 131–146, Cham, 2013. Springer International Publishing.

6 Lars Hupel. Splicing runtime ML values into Isar. isabelle-users mailing list. 09 Jun 2015, https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2015-June/msg00076.html.

7 Lars Hupel. CakeML. *Archive of Formal Proofs*, 2018, 2018. URL: https://www.isa-afp.org/entries/CakeML.html.

8 J. Hurd. OpenTheory: Package Management for Higher Order Logic Theories. In G. Dos Reis and L. Théry, editors, *Programming Languages for Mechanized Mathematics Systems*, pages 31–37. ACM, 2009.

9 Joe Hurd. The OpenTheory Standard Theory Library. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods – Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2011. doi:10.1007/978-3-642-20398-5.

10 Cezary Kaliszyk and Alexander Krauss. Scalable LCF-Style Proof Translation. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, pages 51–66, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

11 Chantal Keller and Benjamin Werner. Importing HOL Light into Coq. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving*, pages 307–322, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

12 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 179–192, 2014. doi:10.1145/2535838.2535841.

13 Ondřej Kunčar. *Types, Abstraction and Parametric Polymorphism in Higher-Order Logic*. Dissertation, Technische Universität München, München, 2016. URL: http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20160408-1285267-1-5.

14 David Matthews. The Poly/ML implementation of Standard ML. Website. URL: https://www.polyml.org.

15 Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

16 Steven Obua and Sebastian Skalberg. Importing HOL into Isabelle/HOL. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning*, pages 298–302, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

17 Lawrence C Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media, 1994.

18 Konrad Slind. Function definition in higher-order logic. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics*, pages 381–397, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

19 Konrad Slind and Michael Norrish. A brief overview of HOL4. In *International Conference on Theorem Proving in Higher Order Logics*, pages 28–32. Springer, 2008.

20 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, Vienna, Austria*, volume 8558 of *Lecture Notes in Computer Science*. Springer, 2014. URL: https://www21.in.tum.de/~wenzelm/papers/itp-pide.pdf.

21 Makarius Wenzel. *The Isabelle/Isar Implementation*, 2019. URL: http://isabelle.in.tum.de/website-Isabelle2019/dist/Isabelle2019/doc/implementation.pdf.

Generating Verified LLVM from Isabelle/HOL

Peter Lammich
The University of Manchester, UK
peter.lammich@manchester.ac.uk

Abstract

We present a framework to generate verified LLVM programs from Isabelle/HOL. It is based on a code generator that generates LLVM text from a simplified fragment of LLVM, shallowly embedded into Isabelle/HOL. On top, we have developed a separation logic, a verification condition generator, and an LLVM backend to the Isabelle Refinement Framework.

As case studies, we have produced verified LLVM implementations of binary search and the Knuth-Morris-Pratt string search algorithm. These are one order of magnitude faster than the Standard-ML implementations produced with the original Refinement Framework, and on par with unverified C implementations. Adoption of the original correctness proofs to the new LLVM backend was straightforward.

The trusted code base of our approach is the shallow embedding of the LLVM fragment and the code generator, which is a pretty printer combined with some straightforward compilation steps.

2012 ACM Subject Classification Theory of computation → Program verification; Theory of computation → Logic and verification; Theory of computation → Separation logic

Keywords and phrases Isabelle/HOL, LLVM, Separation Logic, Verification Condition Generator, Code Generation

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.22

Supplement Material `http://www21.in.tum.de/~lammich/isabelle_llvm`

Funding We received funding from DFG grant LA 3292/1 "Verifizierte Model Checker" and VeTSS grant "Formal Verification of Information Flow Security for Relational Databases".

Acknowledgements We thank Maximilian P. L. Haslbeck and Simon Wimmer for proofreading and useful suggestions.

1 Introduction

The Isabelle Refinement Framework [33, 26, 27] features a stepwise refinement approach to verified algorithms, using the Isabelle/HOL theorem prover [42, 41]. It has been successfully applied to verify many algorithms and software systems, among them LTL and timed automata model checkers [15, 6, 48], network flow algorithms [32, 31], a SAT-solver certification tool [29, 30], and even a SAT solver [16]. Using Isabelle/HOL's code generator [18], the verified algorithms can be extracted to functional languages like Haskell or Standard ML. However, the code generator only provides partial correctness guarantees, i.e., termination of the generated code cannot be proved. Moreover, the generated code is typically slower than the same algorithms implemented in C or Java.

The original Refinement Framework [33, 26] could only generate purely functional code. The first remedy to the performance problem was to introduce array data structures that behave like functional lists on the surface, but are implemented by destructively updated arrays behind the scenes, similar to Haskell's now deprecated DiffArray. While this gained some performance, the array implementation itself was not verified, such that we had to trust its correctness. Moreover, an array access still required a significant amount of overhead compared to a simple pointer dereference in C.

The next step towards more efficient verified implementations was the Sepref tool [27]. It generates code for Imperative HOL [7], which provides a heap monad inside Isabelle/HOL, and a code generator extension to generate code that uses the stateful arrays provided by ML, or the heap monad of Haskell. The Sepref tool performs automatic data refinement from abstract data types like maps or sets to concrete implementations like hash tables, which can be placed on the heap and destructively updated. Moreover, it provides tools [28] to assist in the definition of new data structures, exploiting "free theorems" [45] that it obtains from parametricity properties of the abstract data types. Using Imperative HOL as backend, we gained some additional performance: For example, the GRAT tool [29, 30] provides a verified checker for UNSAT certificates in the DRAT format [47]. It is faster than the unverified state-of-the art checker DRAT-TRIM [47], which is written in C. However, the GRAT tool spends most of its run time in an unverified certificate preprocessor. Nevertheless, optimizing the verified part of the code is important: The very same technique was also implemented in Coq, using purely functional data structures [12, 11]. There, the verified code was actually the bottleneck[1].

This paper presents a next step towards efficient verified algorithms: A refinement framework to generate verified code in LLVM intermediate representation [35] with total correctness guarantees. LLVM is an imperative intermediate language with a powerful and well-tested optimizing compiler. We first formalize the semantics of Isabelle-LLVM, a simple imperative language shallowly embedded into Isabelle/HOL, and designed to be easily translated to actual LLVM text (§2). On top of Isabelle-LLVM, we build a separation logic and a verification condition generator, which allows convenient reasoning about Isabelle-LLVM programs (§3). Finally, we modify the Sepref tool to target Isabelle-LLVM instead of Imperative/HOL (§4), connecting the Refinement Framework to our LLVM code generator. This only affects the last refinement step, such that most parts of existing verifications can be reused. As case studies (§5), we verify a binary search algorithm and adopt an existing formalization [19] of the Knuth-Morris-Pratt string search algorithm [24]. The resulting LLVM code is significantly faster than the corresponding Standard-ML code and on par with unverified C implementations. The paper ends with the discussion of future work (§6) and related work (§7). The Isabelle theories described in this paper are available as supplement material (URL displayed in paper header).

2 Isabelle-LLVM

2.1 State Monad

The basis of Isabelle-LLVM is a state-error monad, which we use to conveniently model the preconditions of instructions, their effect on memory, as well as arbitrary recursive programs. We define the algebraic data types:

```
('a,'s) M = M (run: 's ⇒ ('a,'s) mres)        ('a,'s) mres = NTERM | FAIL | SUCC 'a 's
```

An entity of type ‹('a,'s) M› contains a function ‹run› that maps a start state of type ‹'s› to a *monad result* that indicates either nontermination, a failure, or a successful execution with a result of type ‹'a› and a new state. We define the standard monad combinators:

[1] Later, the checker was rewritten in ACL2, also using imperative data structures [11, 20].

```
return x  = M (λs. SUCC x s)                    get   = M (λs. SUCC s s)
fail      = M (λ_. FAIL)                        set s = M (λ_. SUCC () s)
bind m f  = M (λs. case run m s of SUCC x s ⇒ run (f x) s | r ⇒ r)
assert Φ  = if Φ then return () else fail
```

That is, ‹`return` x› returns result ‹x› without changing the state, ‹`fail`› aborts the computation, ‹*get*› returns the current state, and ‹*set s*› updates the current state. Finally, ‹`bind` m f› first executes ‹m›, and then ‹f› with the result of ‹m›. If ‹m› fails or does not terminate, the whole bind fails or does not terminate. The derived ‹`assert` Φ› combinator can be conveniently used to abort the computation if some precondition is violated, e.g., on division by zero.

We use do-notation, i.e. ‹`do` { x←m; $f\ x$ }› is short for ‹`bind` m (λx. $f\ x$)›. Moreover, we define a flat chain complete partial order [37] on ‹*mres*›, with ‹⊥ := *NTERM*›. For a monotonic function ‹*F* :: ('*a* ⇒ ('*b*,'*s*) *M*) ⇒ '*a* ⇒ ('*b*,'*s*) *M*›, ‹*REC F*› is the least fixed point. As functions defined using the monad combinators are monotonic by construction [25], we can define arbitrary recursive computations. The partial function package [25] provides automation for monotonicity proofs and for defining simple recursive functions. Mutual recursion still requires some manual effort, though it could be automated, too.

2.2 Memory Model

We use a high-level memory model that does not directly expose the bit-level representation of values and assumes an infinite supply of memory. The memory is modeled as a list of blocks. Each block is either deallocated, or it is a list of values. A value is a pair of values, a pointer, or an integer. We model memory by the following data types[2]:

```
memory = MEMORY (block list)          block     = val list option
val = PAIR val val | PRIM primval     primval = PV_INT lint | PV_PTR rptr
```

Here, the type ‹*lint*› is a fixed bit width word type with a two's complement semantics, as used by LLVM, and pair corresponds to a 2-element structure in LLVM. The type ‹*rptr*› is either null or an address. An address is a path through the memory structure to a value:

```
rptr  = NULL | ADDR nat nat (va_dir list)          va_dir = PFST | PSND
```

An address consists of a *block index*, a *value index*, and a *value address*, which is a list of directions to either descend into the first or the second value of a pair.

For the rest of this paper, we will use the state monad with a memory as state. Thus, we define the type ‹'*a llM* = ('*a*,*memory*) *M*›. It is straightforward to define functions ‹*load* :: *rptr* ⇒ *val llM*› and ‹*put* :: *val* ⇒ *rptr* ⇒ *unit llM*› to read/write a value from/to a pointer, or fail if the pointer is invalid. For the actual store function, we check that the structure of the value does not change, i.e. pairs remain pairs, pointers remain pointers, and words of width w remain words of width w:

```
store x p = do { y ← load p; assert (vstruct x = vstruct y); put x p }
where
vstruct (PAIR a b) = VS_PAIR (vstruct a) (vstruct b)
vstruct (PRIM (PV_PTR _)) = VS_PTR
vstruct (PRIM (PV_INT w)) = VS_INT (width w)
```

[2] We have slightly simplified the presentation. The actual implementation defines the concepts memory, block, and value in a modular fashion, in order to ease future extensions.

Similarly, we define an allocate and a free function:

```
allocn v n = do {
  blocks ← get;
  set (blocks@[Some (replicate n v)]);
  return (ADDR |blocks| 0 []) }
```

```
free (ADDR bi 0 []) = do {
  blocks ← get;
  assert (bi < |blocks| ∧ blocks!bi ≠ None);
  set (blocks[bi:=None]) }
free _ = fail
```

Here, ‹$l_1 @ l_2$› concatenates two lists, ‹$|l|$› is the length of list ‹l›, ‹$l!i$› is the ith element of ‹l›, and ‹$l[i:=x]$› replaces the ith element of ‹l› by ‹x›. The allocate function takes an initial value and a block size, appends a new block to the memory, and returns a pointer to the start of the new block (value index 0, and value address []). The free function expects a pointer to the start of a block, checks that this block is not already deallocated, and then deallocates the block by setting it to ‹*None*›.

2.3 Towards a Shallow Embedding

While we explicitly model values in memory by the type ‹*val*›, we model values in registers in a more shallow fashion: We identify LLVM registers with Isabelle variables that have a type of shape ‹$T = T \times T \mid n\ word \mid T\ ptr$›. Here, ‹$\times$› is Isabelle's product type, ‹$n\ word$› is the n bit word type from Isabelle's word library[3], and ‹$'a\ ptr$› is a pointer with an attached phantom type for the value pointed to (‹$'a\ ptr = PTR\ rptr$›). For each type ‹$'a$› of shape ‹T›, we define the functions:

```
to_val    :: 'a ⇒ val          struct_of  :: 'a itself ⇒ vstruct
from_val :: val ⇒ 'a           init       :: 'a
```

such that

```
from_val o to_val = id                vstruct (to_val x) = (struct_of TYPE('a))
to_val init = zero_initializer (struct_of TYPE('a))
```

Here, ‹$TYPE('a) :: 'a\ itself$› reflects type ‹$'a$› into a term. The functions ‹*to_val*› and ‹*from_val*› inject a T-shaped type ‹$'a$› into a value with structure ‹$struct\_of\ TYPE('a)$›. Moreover, ‹$init::'a$› corresponds to the all-zeroes value, i.e., the value where all pointers are null pointers, and all integers are 0.

2.4 Instructions

In a next step, we define the instructions of Isabelle-LLVM. Each instruction is identified with an Isabelle constant. For example, the load instruction is modeled by:

```
ll_load :: 'a ptr ⇒ 'a llM
ll_load (PTR p) = do {
  v ← load p;
  assert (vstruct v = struct_of TYPE('a));
  return (from_val v) }
```

[3] For convenient notation, we use the type ‹$n\ word$› as if it were a type depending on a variable n. Isabelle/HOL is not dependently typed. Instead, n is actually a type variable with type-class ‹*len*›, which provides a function ‹$len\_of :: 'a::len\ itself \Rightarrow nat$› to extract the length as a term.

It loads a value from the specified pointer, checks that its structure matches the expected type ‹'a›, and then converts the value to ‹'a›.

For allocation and deallocation, we provide the instructions:

```
ll_malloc :: 'a itself ⇒ n word ⇒ 'a ptr llM        ll_free :: 'a ptr ⇒ unit llM
```

Note that LLVM does not contain a heap manager. Instead, we assume that the generated code will be linked with the C standard library, and let the code generator produce calls to ‹*calloc*› and ‹*free*›. We also define instructions to access the elements of a pair, to offset a pointer, and to advance a pointer into a pair. The code generator maps these instructions to the corresponding LLVM instructions ‹*getelementptr*›, ‹*insertvalue*›, and ‹*extractvalue*›.

Integer instructions are defined on the ‹*n word*› type. For example, we define:

```
ll_udiv :: n word ⇒ n word ⇒ n word
ll_udiv a b = do { assert (b ≠ 0); return (a div b) }
```

where ‹*div*› is the unsigned division from Isabelle's word library. Note the use of assertions to exclude undefined behavior, e.g., division by zero.

2.5 Modeling Control Flow

Next, we put together instructions to form procedure bodies. We only allow structured control flow via if-then-else, while, procedure calls, and sequential composition: The body of a procedure is modeled by an Isabelle term of type ‹'a *llM*› and shape ‹*block*›, where

```
block = do { var ← cmd; block } | return var
cmd = ll_<opcode> arg* | proc_name arg* | llc_if arg block block | llc_while block block
arg = var | number | null | init
```

with

```
llc_if :: 1 word ⇒ 'a llM ⇒ 'a llM ⇒ 'a llM
llc_if b t e = if b=1 then t else e

llc_while :: ('a ⇒ 1 word llM) ⇒ ('a ⇒ 'a llM) ⇒ 'a ⇒ 'a llM
llc_while b c s = do {ctd ← b s; llc_if ctd (do {s ← c s; llc_while b c s}) (return s)}
```

That is, a block is a list of commands whose results are bound to variables, terminated by a return instruction. A command is either an instruction, a procedure call, or an if-then-else or while statement. The arguments of instructions and procedure calls, as well as the condition of an if-then-else statement, must be variables or constants (i.e., numbers, the null pointer, or a zero-initialized value). The condition of a while statement is modeled as a block returning a ‹*1 word*›, such that it can be re-evaluated prior to each loop iteration. A program is represented by a set of (monomorphic) theorems of the shape ‹$proc_i\ x_1\ \ldots\ x_n = cmd$›, where the ‹$proc_i$› are Isabelle functions, the ‹x_i› are variables, and all free variables on the right hand side are among the ‹x_i›.

▶ **Example 1.** Figure 1 shows the Isabelle specification of a procedure named ‹*fib*›, which takes a 64 bit word argument, and returns a 64 bit word. Our semantics can be directly executed inside Isabelle. The following Isabelle command evaluates ‹*fib*› on the first few natural numbers, and an empty memory:

```
value ‹map (λn. run (fib n) (MEMORY [])) [0,1,2,3]›
(* output: [SUCC 0 (MEMORY []), SUCC 1 ..., SUCC 1 ..., SUCC 2 ...] *)
```

```
fib:: 64 word ⇒ 64 word llM
fib n = do {
  t ← ll_icmp_ule n 1;
  llc_if t (return n) (do {
    n1 ← ll_sub n 1;
    a  ← fib n1;
    n2 ← ll_sub n 2;
    b  ← fib n2;
    c  ← ll_add a b;
    return c
  }) }
```

Figure 1 Isabelle-LLVM program.

```
define i64 @fib(i64 %x) {
  start:
    %t = icmp ule i64 %x, 1
    br i1 %t, label %then, label %else
  then:
    br label %ctd_if
  else:
    %n_1 = sub i64 %x, 1
    %a   = call i64 @fib (i64 %n_1)
    %n_2 = sub i64 %x, 2
    %b   = call i64 @fib (i64 %n_2)
    %c   = add i64 %a, %b
    br label %ctd_if
  ctd_if:
    %x1a = phi i64 [ %x, %then ], [ %c, %else ]
    ret i64 %x1a }
```

Figure 2 Generated LLVM text.

2.6 Code Generation

The LLVM intermediate representation [35] is a strongly typed control flow graph (CFG) based intermediate language that uses single static assignment (SSA) form [13]. A procedure is a list of basic blocks, the first block in the list being the entry point of the procedure. A basic block is a list of instructions, finished by a terminator instruction that determines the next basic block to execute (or to return from the current procedure). Each non-void instruction defines a fresh register containing its result. A register can only be accessed in the part of the CFG that is dominated by its definition. To transfer values from registers to other parts of the CFG, ϕ-instructions are used. A ϕ-instruction must be located at the start of a basic block. It lists, for each possible predecessor block, an accessible register in this predecessor block. The ϕ-instruction evaluates to the value of the register from those predecessor block from which execution was actually transferred. The result of the ϕ-instruction is bound to a fresh register, which can then be accessed from the current basic block.

It is straightforward to map an Isabelle-LLVM program to an actual LLVM program. Each equation of the form ‹*proc* x_1 .. x_n = *block*› is mapped to an LLVM function named ‹*proc*›. A block is mapped to a control flow graph. Instructions and procedure calls are directly mapped to LLVM instructions and calls. An ‹$x \leftarrow$ *llc_if b t e*› is translated to conditional branching, using a ϕ-instruction to define the result register ‹x› when joining the control flow. An ‹$x \leftarrow$ *llc_while b c s*› is translated similarly.

▶ **Example 2.** Figure 2 displays the output of our code generator for the ‹*fib*› constant displayed in Figure 1.

2.6.1 Mapping the Memory Model

Mapping the abstract memory model of Isabelle-LLVM to actual LLVM is slightly more involved. For example, recall the ‹*ll_malloc :: 'a itself ⇒ n word ⇒ 'a ptr llM*› instruction. It has to be mapped to the function ‹*void* calloc(size_t, size_t)*› from the C standard library.

For this, we have to parameterize the code generator with the architecture dependent size of the ‹*size_t*› type. Next, we have to obtain the size of type ‹*'a*› and cast the ‹*n word*› parameter to ‹*size_t*›. Here, our code generator will refuse downcast, as this might result in bits being dropped. Finally, we have to cast the returned ‹*void**› to the correct return type. Moreover, the ‹*calloc*› function returns ‹*null*› if not enough memory is available. In contrast, our semantics always returns a new block of memory. We insert code to terminate the program in a defined way if it runs out of memory. The relation between our semantics and the actual LLVM program then becomes: Either the program terminates with an out-of-memory condition, or it behaves as modeled by the semantics. Our current implementation prints an error message and terminates the process with exit code 1 if it runs out of memory.

A similar issue arises when comparing pointers: LLVM does not have instructions for pointer comparison. Instead, pointers have to be cast to integers, which can then be compared. However, this requires to know the bit-width of a pointer, which we cannot model in our semantics that admits unboundedly many different pointers. Instead, we model the instructions ‹*ll_ptrcmp_eq*› and ‹*ll_ptrcmp_ne*›, and let the code generator generate the cast to integers and the integer comparison.

2.7 Preprocessing

In the previous sections we have described the semantics of Isabelle-LLVM and its translation to actual LLVM. However, Isabelle-LLVM programs have to adhere to a very restrictive shape (cf. §2.5), which makes them easy to map to actual LLVM code, but tedious to write directly. Thus, we implement a preprocessor that tries to automatically transform user-specified equations to valid Isabelle-LLVM. While the preprocessing is highly incomplete, i.e., it cannot convert every equation to a well-shaped one, it works well in practice, allowing for concise specifications. Note that the preprocessor *proves* the new equations from the original ones. Thus, errors in the preprocessor cannot affect soundness: Either, it fails to prove the equations, or it produces ill-shaped equations, which the code generator will reject.

The user specifies an initial set of constants, which must be instantiated to monomorphic types, i.e., must not contain any type variables. For each constant, the preprocessor then gathers the defining equation, instantiates it to the actual monomorphic type of the constant, transforms it by inlining and fixed point unfolding, and then repeats the process for any new constant occurring on the right-hand side of the transformed equation. Note that a constant is identified by its name and type, such that a constant with the same name can occur multiple times in the final Isabelle-LLVM program. The code generator will disambiguate the names. At the end, we have a set of monomorphic equations that define all constants that occur in the final program, and can be passed to the actual code generator. We now describe the inlining and fixed point unfolding transformations.

2.7.1 Inlining

Inlining first applies user defined rewrite rules and then flattens nested expressions, converting function calls to the shape ‹$r \leftarrow f\ x_1\ \dots\ x_n$› or ‹$r \leftarrow$ `return` $(f\ x_1\ \dots\ x_n)$›, where the x_i are either constants, variables, or *monadic* arguments of type ‹$\dots \Rightarrow \_\ llM$›. Subterms of type ‹$\_\ llM$› are recursively flattened. We iterate the rewriting and flattening steps until a fixed point is reached.

▶ **Example 3.** Consider the following definition of the constant ‹*fib'*›:

```
fib' :: m word ⇒ m word llM
fib' n = if n ≤ 1 then return n
       else do { n₁ ← fib' (n − 1); n₂ ← fib' (n − 2); return (n₁ + n₂) }
```

When started with ‹*fib′ :: 64 word ⇒ 64 word llM*›, the preprocessor automatically translates this equation to the equation displayed in Figure 1. During the translation, it uses the following inlining rules:

```
if b then c else t = llc_if (from_bool b) c t        return (a + b) = ll_add a b
return (from_bool (a≤b)) = ll_icmp_ule a b           return (a − b) = ll_sub a b
```

Our default setup contains similar rules for the other operations, as well as rules to map tuples and case-distinctions over tuples to ‹*insertvalue*› and ‹*extractvalue*› instructions.

2.7.2 Fixed-Point Unfolding

The preprocessor generates recursive functions from fixed-point combinators. It examines the right hand side of an equation for patterns ‹*p*› for which it has an unfold rule of the form ‹*p = F p*›. It then defines a new constant ‹*f x*$_1$ *...* *x*$_n$ *= F* (*f x*$_1$ *...* *x*$_n$)›, where the ‹*x*$_i$› are the free variables in the pattern ‹*p*›. Finally, it replaces ‹*p*› by ‹*f x*$_1$ *...* *x*$_n$› in the equation. This way, specifications with fixed point combinators are automatically transformed to a set of recursive equations, as required by the code generator.

For example, the ‹*llc_while*› combinator is defined as a fixed point (cf. §2.5). Using its definition as an unfold rule, the preprocessor will automatically convert while loops into tail calls. This allows for using while-loops without trusting their translation in the code generator. A configuration option in our tool lets the user choose between direct while-loop translation or unfolding into a tail call.

▶ **Example 4.** Consider the following program:

```
euclid :: 64 word ⇒ 64 word ⇒ 64 word
euclid a b = do {
  (a,b) ← llc_while
    (λ(a,b) ⇒ ll_cmp (a ≠ b))
    (λ(a,b) ⇒ if (a≤b) then return (a,b−a) else return (a−b,b))
    (a,b);
  return a }
```

From this, the preprocessor proves the following two equations (before inlining):

```
euclid a b = do {
    (a, b) ← euclid₀ (a, b);
    return a }
euclid₀ s = do {
    ctd ← case s of (a, b) ⇒ ll_cmp (a ≠ b);
    llc_if ctd (do {
      s ← case s of (a, b) ⇒ if a ≤ b then return (a, b − a) else return (a − b, b);
      euclid₀ s
    }) (return s) }
```

That is, it defined a new constant ‹*euclid*$_0$› to replace the while loop by tail recursion.

3 Verification Condition Generator

The next step towards generating verified LLVM programs is to establish a reasoning infrastructure. In this section, we describe our separation logic [43] based verification condition generator. Note that, while applying complex operations on the proof state, at the end, our VCG conducts a proof that goes through Isabelle's inference kernel. Thus, bugs in the VCG cannot cause unsoundness.

3.1 Separation Algebra

The first step to obtain a separation logic is to define a separation algebra on a suitable abstraction of the memory. A separation algebra [8] is a structure with a zero, a disjointness predicate $a\#b$, and a disjoint union $a+b$. Intuitively, elements describe parts of the memory. Zero describes the empty memory, $a\#b$ means that a and b describe disjoint parts of the memory, and $a+b$ describes the memory described by the union of a and b. For the exact definition of a separation algebra, we refer to [8, 22]. We note that separation algebras naturally extend over functions, pairs, and option types.

We abstract a value by a partial function from value addresses (‹*va_dir list*›) to primitive values, such that the addresses in the domain of the function are independent, i.e., no address is the prefix of another address:

typedef *aval* = { *m :: vaddr* ⇒ *'a option. ∀va,va'∈dom m. va≠va'* ⟶ *indep va va'* }
val_α :: val ⇒ *aval*
val_α (*PRIM x*) = [[] ↦ *x*]
val_α (*PAIR x y*) = *PFST* · *val_α x* + *PSND* · *val_α y*

Here, ‹$[k \mapsto v]$› is the partial function that maps ‹k› to ‹v›, and ‹$i \cdot a$› prepends the item ‹i› to all addresses in the domain of ‹a›. It is straightforward (though technically involved) to show that abstract values form a separation algebra, where the empty map is zero, maps are disjoint iff their domains are pairwise independent, and union merges two maps.

A natural abstraction of a block (‹*val list*›) would be a function from indexes to abstract values, mapping invalid indexes to 0. However, this abstraction does not contain enough information to reason about deallocation. In order to deallocate a block, we have to own the whole block. However, from the abstraction, we cannot infer the size of the block, and thus we cannot specify an assertion that ensures that we own the whole block. A remedy (which the author has seen in [1]) is to additionally abstract a block to its size. Thus, abstract blocks have the type ‹*ablock* = (*nat* ⇒ *aval*) × *nat option*›. The option type is required to make the second elements of the tuples a separation algebra. We use the trivial separation algebra here, where two elements are only disjoint if at least one of them is ‹*None*›. Finally, we define ‹*amemory* = *nat* ⇒ *ablock*›, and a function ‹*α :: memory* ⇒ *amemory*› that abstracts memory by a function from block indexes to abstract blocks, mapping deallocated or invalid indexes to zero.

3.2 Basic Reasoning Infrastructure

Predicates of type ‹*assn* = *amemory* ⇒ *bool*› are called *assertions*. The *weakest precondition* of a program ‹*c :: 'a llM*›, a *postcondition* ‹*Q :: 'a* ⇒ *assn*›, and a memory ‹*s*› is defined as:

wp c Q s = (∃*r s'*. *run c s* = *SUCC r s'* ∧ *Q r* (*α s'*))

Intuitively, ‹*wp c Q s*› states that program ‹*c*›, if run on memory ‹*s*›, terminates successfully with the result ‹*r*›, and the abstraction of the new state ‹*s'*› satisfies ‹*Q*›.

For assertions ‹*P*› and ‹*Q*›, the *separating conjunction* ‹*P*∗*Q*› describes a memory that can be split into two disjoint parts described by ‹*P*› and ‹*Q*›, respectively:

$$(P * Q)\ s = \exists s_1\ s_2.\ s_1 \perp s_2 \wedge s = s_1 + s_2 \wedge P\ s_1 \wedge Q\ s_2$$

Validity of a *Hoare triple* ‹{*P*} *c* {*Q*}› is defined as follows:

$$\models \{P\}\ c\ \{Q\} = \forall F\ s.\ (P*F)\ (\alpha\ s) \longrightarrow wp\ c\ (\lambda r\ s'.\ (Q\ r * F)\ s')\ s$$

That is, if the memory can be split into a part described by the *precondition* ‹*P*›, and a *frame* described by ‹*F*›, then command ‹*c*› will succeed, and the new memory consists of a part described by the postcondition ‹*Q*› and the unchanged frame. Our Hoare triples satisfy the frame rule: ‹$\models \{P\}\ c\ \{Q\} \implies \models \{P * F\}\ c\ \{\lambda r.\ Q\ r * F\}$› for all ‹*F*›.

3.3 Basic Rules

Once we have set up the separation algebra and the abstraction function, we can prove Hoare triples for the basic operations of our memory model. For example, we prove the following rules for ‹*allocn*› and ‹*free*›:

$$\models \{\square\}\ allocn\ v\ n\ \{\lambda p.\ malloc\_tag\ n\ p * range\ \{0..<n\}\ (\lambda\_.\ v)\ p\}$$
$$\models \{malloc\_tag\ n\ p * \exists blk.\ range\ \{0..<n\}\ blk\ p\}\ free\ p\ \{\lambda\_.\ \square\}$$

where ‹$\square = \lambda s.\ s=0$› describes the empty memory, ‹*malloc_tag n p*› asserts that ‹*p*› points to the beginning of a block, and the size field of this block's abstraction is ‹*n*›, and ‹*range I f p*› describes that for all ‹$i \in I$›, ‹$p + i$› points to value ‹$f\ i$›. Intuitively, ‹*allocn*› creates a block of size ‹*n*›, initialized with values ‹*v*›, and a tag. If one possesses both, the whole block and the tag, it can be deallocated by free. For the Isabelle-LLVM memory instructions, we obtain the following rules:

$$\models \{n\neq 0\}\ ll\_malloc\ TYPE('a)\ n\ \{\lambda p.\ range\ \{0..<n\}\ (\lambda\_.\ init)\ p * malloc\_tag\ n\ p\}$$
$$\models \{range\ \{0..<n\}\ blk\ p * malloc\_tag\ n\ p\}\ ll\_free\ p\ \{\lambda\_.\ \square\}$$
$$\models \{pto\ x\ p\}\ ll\_load\ p\ \{\lambda r.\ r=x * pto\ x\ p\}$$
$$\models \{pto\ xx\ p\}\ ll\_store\ x\ p\ \{\lambda\_.\ pto\ x\ p\}$$

Here, ‹*pto x p*› describes that p points to value x, and we write predicates as if they were assertions on the empty memory, e.g., ‹$n\neq 0$› instead of ‹$\lambda s.\ s=0 \wedge n\neq 0$›. We prove similar rules for the other instructions.

3.4 Automating the VCG

In order to efficiently prove Hoare triples, some automation is required. We provide a verification condition generator with a frame inference heuristics. The first step to prove a Hoare triple is to convert it to a proposition on weakest preconditions:

$$[\![\bigwedge F\ s.\ STATE\ (P*F)\ s \implies wp\ c\ (\lambda r\ s'.\ (Q\ r * F)\ s')\ s]\!] \implies \models \{P\}\ c\ \{Q\}$$

where ‹$STATE\ P\ s = P\ (\alpha\ s)$›. In general, the VCG operates on subgoals of the form ‹$STATE\ P\ s \implies wp\ c\ Q\ s$›. It then iteratively performs one of the following steps[4]:

[4] This is a simplified presentation. The actual VCG is an instantiation of a generic VCG framework that can be configured with various solvers, rules, and heuristics.

simplification. Apply a rewrite rule to transform ‹*wp c Q s*› into some equivalent proposition. For example, binding is resolved by the rule:

$$wp\ (\texttt{do}\ \{x{\leftarrow}m;\ f\ x\})\ Q\ s = wp\ m\ (\lambda x.\ wp\ (f\ x)\ Q)\ s$$

rule. If there is a Hoare triple of the form ‹$\models \{P'\}\ c\ \{Q'\}$›, the VCG tries to infer a frame ‹F› such that ‹$P \vdash P'{*}F$›, and replaces the goal by ‹$STATE\ (Q'{*}F)\ s' \Longrightarrow Q\ s'$› for a fresh ‹$s'$›. Here, ‹$P \vdash Q = \forall s.\ P\ s \Longrightarrow Q\ s$› denotes entailment.

final. If the goal has the form ‹$STATE\ P\ s \Longrightarrow Q\ s$› such that ‹$Q$› is not of the form ‹*wp _ _ _*›, a heuristics is used to prove ‹$P \vdash Q$›.

The actual verification conditions are generated during frame inference and the final proof heuristics. For example, the rule for ‹*ll_malloc*› requires to prove that the size operand is not zero. The VCG will try to prove these goals by a default tactic, and leave them to the user if this tactic fails.

▶ **Example 5.** Recall the function ‹*euclid :: 64 word ⇒ 64 word ⇒ 64 word llM*› from Example 4. We prove the following Hoare triple:

$$\models \{uint_{64}\ a\ a_\dagger\ *\ uint_{64}\ b\ b_\dagger\ *\ 0{<}a\ *\ 0{<}b\}\ euclid\ a_\dagger\ b_\dagger\ \{\lambda r_\dagger.\ uint_{64}\ (gcd\ a\ b)\ r_\dagger\}$$

Here, ‹$uint_{64}\ a\ a_\dagger$› states that ‹$a_\dagger$*::64 word*› is an unsigned integer with value ‹*a::int*›, where ‹*int*› is the type of (mathematical) integers in Isabelle, and ‹*gcd*› is Isabelle's greatest common divisor function. After annotating a suitable loop invariant, the VCG generates the following two verification conditions:

$$[\![\ gcd\ x\ y = gcd\ a\ b;\ x \neq y;\ x \leq y;\ \ldots\ ]\!] \Longrightarrow gcd\ x\ (y - x) = gcd\ a\ b$$
$$[\![\ gcd\ x\ y = gcd\ a\ b;\ \neg\ x \leq y;\ \ldots\ ]\!] \Longrightarrow gcd\ (x - y)\ y = gcd\ a\ b$$

These are straightforward to prove in Isabelle, e.g., using sledgehammer [3].

3.5 Data Structures and Basic Refinement

Recall Example 5. The Hoare triple that is proved there first maps the 64 bit word arguments and results to mathematical integers, and then phrases the correctness statement in terms of mathematical integers. This approach is often more feasible than stating correctness on the concrete implementation directly. In our case, we would have to define the concept of greatest common divisor for 64 bit words. In general, an algorithm often computes some function on abstract mathematical concepts like integers or sets, but has to implement these by concrete data structures like 64 bit words or hash-tables. Thus, a concise way to specify the correctness statement is to first map the implementations back to the abstract concepts, and then state the actual correctness abstractly.

In separation logic based reasoning, a data structure provides a *refinement assertion* ‹$A\ x\ x_\dagger$ *:: assn*›, which describes that the abstract value ‹x› is implemented by the concrete value ‹$x_\dagger$›. We define refinement assertions to implement integers and natural numbers by n bit words, and to implement lists by blocks of memory. On top of that, we define more complex data structures like dynamic arrays. Note that new data structures can easily be added. In general, an implementation does not completely implement an abstract mathematical concept. For example, n bit words can only represent the integers ‹$sints\ n = \{-2^{n-1}..<2^{n-1}\}$›, and hash-tables can only represent finite sets. Thus, the rules for the operations generally come with additional preconditions. For example, the rule to implement subtraction on integers by subtraction on n bit words is the following:

$$\models \{sint_n\ a\ a_\dagger * sint_n\ b\ b_\dagger * a-b \in sints\ n\}\ ll\_sub\ a_\dagger\ b_\dagger\ \{\lambda r_\dagger.\ sint_n\ (a-b)\ r_\dagger\}$$
for $a_\dagger\ b_\dagger :: n\ word$ **and** $a\ b :: int$

Here, ‹$sint_n$› implements mathematical integers by n-bit words. Note that the postcondition does not mention the operands ‹a,b› again, though they are still valid after the operation. As ‹$sint_n$› is *pure*, i.e., does not use the memory, our VCG will automatically add the corresponding assertions to the postcondition.

4 Automatic Refinement

Our basic VCG infrastructure can be used to verify simple algorithms like ‹*euclid*› from Example 5. However, many complex algorithms have already been verified using the Isabelle Refinement Framework [33]. It features a non-deterministic programming language with a refinement calculus and a VCG. It allows to express an algorithm using abstract mathematical concepts, and then refine it in multiple steps towards an efficient implementation. The last step of a refinement is typically performed by the Sepref tool [27], which translates a program from the non-deterministic monad of the Refinement Framework into the deterministic heap monad of Imperative HOL [7], replacing abstract data types by concrete implementations. We have modified the Sepref tool to translate to Isabelle-LLVM's monad instead. We only had to modify the translation phase. The preprocessing phases, which only work on the abstract program, remained unchanged.

The translation phase works by symbolically executing the abstract program, thereby synthesizing a structurally similar concrete program. During the symbolic execution, the relation between the abstract and concrete variables is modeled by refinement assertions. The predicate ‹$hnr\ \Gamma\ m_\dagger\ \Gamma'\ R\ m$› means that concrete program ‹$m_\dagger$› implements abstract program ‹m›, where ‹Γ› contains the refinements for the variables before the execution, ‹Γ'› contains the refinements after the execution, and ‹R› is the refinement assertion for the result of ‹m›. For example, a ‹`bind`› is processed by the following rule:

```
1  ⟦ hnr Γ m_† Γ' R_x m;
2    ⋀x x_†. hnr (R_x x x_† * Γ') (f_† x_†) (R'_x x x_† * Γ'') R_y (f x);
3    MK_FREE R'_x free;
4  ⟧ ⟹ hnr Γ (do {x_†←m_†;r_†←f_† x_†; free x_†; return r_†}) Γ'' R_y (do {x←m; f x})
```

To refine ‹$x\leftarrow m;\ f\ x$›, we first execute ‹m›, synthesizing the concrete program ‹$m_\dagger$› (line 1). The state after ‹m› is ‹$R_x\ x\ x_\dagger * \Gamma'$›, where ‹$x$› is the result created by ‹m›. From this state, we execute ‹$f\ x$› (line 2). The new state is ‹$R'_x\ x\ x_\dagger * \Gamma'' * R_y\ y\ y_\dagger$›, where ‹$y$› is the result of ‹$f\ x$›. Now, the variable ‹$x$› goes out of scope, such that it has to be deallocated. The predicate ‹$MK\_FREE\ R'_x\ free = \forall x\ x_\dagger.\ \models \{R'_x\ x\ x_\dagger\}\ free\ x_\dagger\ \{\lambda\_.\ \square\}$› (line 3) states that ‹*free*› is a deallocator for data structures implemented by refinement assertion ‹R'_x›. Note that the refinement for variable ‹x› may change: If ‹$f_\dagger\ x_\dagger$› overwrites ‹$x_\dagger$›, the refinement assertion for ‹x› will be changed to the special assertion ‹*invalid*›. The deallocator for ‹*invalid*› is simply a no-op. Adding support for deallocators was the most substantial change we applied to the Sepref tool. Its original target language, Imperative HOL, is garbage collected, such that there is no need for explicit deallocation.

4.1 Data Structure Library

Once the basic Sepref tool is adapted, we can define data structures. Reusing the basic data structures from the original Sepref tool is not possible, as Imperative HOL uses arbitrary precision integers and algebraic data types, while we have only fixed width words and pairs.

Up to now, we have added the implementation of integers and natural numbers by n bit words and some basic container data structures like dynamic arrays, bit-vectors, and min-heaps. Thereby, we could reuse the existing infrastructure of the Sepref tool: For example, there is support to automatically generate rules that also support refinement of the elements of a data structure, exploiting "free theorems" [45] which stem from parametricity properties of the abstract types.

5 Case Studies

To assess the usability of our approach, we have verified a binary search algorithm and the Knuth-Morris-Pratt string search [24] algorithm. Both algorithms have also been verified with the original Sepref tool, such that we can compare the two approaches.

5.1 Binary Search

Binary search is a simple algorithm to find a value in a sorted array. Despite its simplicity, it has a history of flawed implementations[5], making it a natural example for formal verification.

We start with a high-level specification: For a list ‹*xs*› and a value ‹*x*›, find the index of the first element greater or equal to ‹*x*›. We define the following constant:

```
fi_spec xs x = spec i. i = find_index (λy. x≤y) xs
```

where ‹*find_index P xs*› is a standard list function that returns the index of the first element in ‹*xs*› that satisfies ‹*P*›, or ‹*length xs*› if there is no such element.

Next, we phrase the binary search algorithm in the Isabelle Refinement Framework:

```
bin_search xs x ≡ do {
  (l,h) ← while
    (λ(l,h). l<h)
    (λ(l,h). do {
      assert (l<length xs ∧ h≤length xs ∧ l≤h);
      let m = l + (h−l) div 2;
      if xs!m < x then return (m+1,h) else return (l,m)
    })
    (0,length xs);
  return l }
```

It is a standard exercise to prove that the algorithm adheres to its specification:

```
bs_correct: sorted xs ⟹ bin_search xs x ≤ fi_spec xs x
```

Finally, we invoke our adapted Sepref tool:

```
sepref_definition bs_impl [llvm_code] is bin_search
  :: (larray64 sint64)^k → sint64^k → snat64
  unfolding bin_search_def [...] by sepref
export_llvm bs_impl file bin_search.ll
lemmas bs_impl_correct = bs_impl.refine[FCOMP bs_correct]
```

[5] A buggy implementation in the Java Standard Library has gone undetected for nearly a decade [5].

This produces an Isabelle-LLVM program ‹*bs_impl*›, exports it to actual LLVM text, and proves the refinement theorem ‹*bs_impl_correct*›:

$$(bs\_impl,\ fi\_spec) : [\lambda(xs,\ \_).\ sorted\ xs]\ (larray_{64}\ sint_{64})^k \times sint_{64}^k \to snat_{64}$$

Here, ‹$snat_w$› implements natural numbers by signed w-bit words[6]. Moreover, ‹$larray_w\ A$› refines a list to an array and a w-bit length field, the elements of the list being refined by assertion ‹A›. The notation ‹$[\Phi]\ A_1^{k|d} \times \ldots \times A_n^{k|d} \to R$› specifies a refinement with precondition ‹Φ›, such that the arguments are refined by ‹$A_1\ \ldots\ A_n$› and the result is refined by ‹R›. The $\cdot^{k|d}$ annotations specify whether an argument is overwritten (k for keep, d for destroy). While we use this notation a lot in the Refinement Framework, it is straightforward to prove a standard Hoare triple from it. By unfolding some definitions we get:

$$\begin{array}{l}\models \{larray_{64}\ sint_{64}\ xs\ xs_\dagger\ *\ sint_{64}\ x\ x_\dagger\ *\ sorted\ xs\ \}\\ \quad bs\_impl\ xs_\dagger\ x_\dagger\\ \quad \{\lambda i_\dagger.\ \exists i.\ larray_{64}\ sint_{64}\ xs\ xs_\dagger\ *\ snat_{64}\ i\ i_\dagger\ *\ i{=}find\_index\ (\lambda y.\ x{\leq}y)\ xs\}\end{array}$$

That is, if we start with an array ‹$xs_\dagger$› representing the sorted list ‹xs›, and a 64-bit word ‹$x_\dagger$› representing the integer ‹x›, then the array still represents ‹$xs_\dagger$›, and the result ‹$i_\dagger$› represents a natural number ‹i›, which is equal to the correct index.

The Sepref tool implements mathematical integers by 64-bit words, proving absence of overflows. This is only possible because the assertion in ‹*bin_search*› explicitly states that the indexes are in bounds. Moreover, note the expression ‹$l + (h{-}l)\ div\ 2$› that we used to compute the midpoint index. On mathematical integers, it is equal to ‹$(l{+}h)\ div\ 2$›. However, on fixed-width words, the latter may overflow, while the former does not[7].

5.2 Knuth-Morris-Pratt String Search

Next, we regard the Knuth-Morris-Pratt (KMP) string search algorithm [24], a well-known linear time algorithm to find the index of the first occurrence of a string s in a string t:

$$\begin{array}{l}ss\_spec\ s\ t = \texttt{spec}\\ \quad None \Rightarrow \nexists i.\ sublist\_at\ s\ t\ i\ |\\ \quad Some\ i \Rightarrow sublist\_at\ s\ t\ i \wedge (\forall ii{<}i.\ \neg sublist\_at\ s\ t\ ii))\end{array}$$

where ‹*sublist_at s t i*› specifies that list ‹s› occurs in list ‹t› at index ‹i›:

$$sublist\_at\ s\ t\ i = \exists ps\ ss.\ t = ps@s@ss \wedge i = length\ ps$$

We have recently formalized KMP with the original Sepref tool [19]. The adaption of the existing formalization was straightforward: In the abstract part, we had to explicitly add a few in-bounds assertions. Most of them were already contained implicitly in the original proof. For the synthesis step, we only had to add setup for the fixed-width word types. The result of the automatic synthesis is an Isabelle-LLVM program ‹*kmp_impl*›, and the theorem:

$$\begin{array}{l}(kmp\_impl,\ ss\_spec)\\ \quad : [\lambda s\ t.\ |s|\ +\ |t| < 2^{63}]\ (larray_{64}\ sint_{64})^k \times (larray_{64}\ sint_{64})^k \to snat\_option_{64}\end{array}$$

Here ‹$snat\_option_{64}$› implements the type ‹*nat option*› by signed 64-bit words, mapping ‹*None*› to -1.

---

[6] As LLVM's index operations are on signed words, it's convenient to always implement sizes and indexes by signed types, even if they are natural numbers.

[7] Exactly this overflow caused the infamous bug in the Java Standard Library [5].

Table 1 Time (ms) to search for the values $0, 2, \ldots < 5n$ in an array $[0, 5, \ldots < 5n]$.

$n/10^6$	C	LLVM	SML	SML*
1	121	100	1999	139
2	251	204	4209	289
3	379	304	6516	440
4	513	412	8843	600
5	635	514	11494	756
6	767	617	13646	917
7	908	726	16032	1076
8	1038	854	18421	1250
9	1162	945	20957	1409
10	1293	1045	23409	1564

Table 2 Time (ms) to run the a-l benchmark suite from StringBench [44]. Here a is the alphabet size, and l the pattern size. The sample size is $3 \cdot 2^{20}$ characters. The algorithm stops after finding the first match.

a-l	C++	LLVM	SML	SML*
16-8	499	597	4616	918
16-64	511	598	4621	926
16-512	513	590	4573	909
32-8	453	551	4471	850
32-64	465	552	4523	857
32-512	463	544	4456	840
64-8	418	530	4433	803
64-64	420	531	4514	809
64-512	416	523	4411	799

5.3 Runtime

We have compared our verified LLVM implementations to unverified C/C++ implementations of the same algorithms, as well as to the Standard ML (SML) implementations generated by the original Sepref tool. While we have implemented binary search in C ourselves, we used a publicly available code snippet [34] for KMP[8]. The programs were compiled with MLton-2018 [39] and clang-6.0 [10], and run on a standard laptop machine (2.8GHz Quadcore i7 with 16MiB RAM). Tables 1 and 2 display the results: The verified LLVM implementations are on par with the unverified C/C++ implementations, and an order of magnitude faster than the SML implementations.

Isabelle's code generator uses arbitrary precision integers, which tend to be significantly slower than fixed-width integers. The SML* column shows the results when we manually replace the arbitrary precision integers by 64-bit integers in the generated code. While this is unsound in general, it gives us a lower bound of what would be possible in SML with more elaborate code generator configurations[9]. SML* is significantly faster than the original SML, but still 1.5 times slower than LLVM.

6 Future Work

While our case studies only cover medium complex algorithms, we expect that our approach will scale to more complex algorithms, e.g. model checkers [48, 16] and SAT solvers [16], which have already been formalized with the original refinement framework. While these formalizations use a combination of functional and imperative data structures, the LLVM backend only supports imperative data structures. We expect the necessary changes to be manageable, but non-trivial. In particular, the current Sepref tool only supports pure data structures to be nested in containers. In the Imperative HOL setting, we simply use functional data structures inside containers. For LLVM, nested container data structures currently require ad-hoc proofs on the separation logic level. We leave the lifting of Sepref to support nested imperative data structures to future work.

[8] One easily finds many C implementations of KMP, mainly differing in the loop structure. We tried to choose one that is close to our implementation.

[9] Fleury et al. [16] have successfully experimented with such code generator tuning.

Moreover, the refinement from arbitrary precision integers to fixed size integers was quite straightforward for our case studies, and we expect these refinements to be more complex in general. We leave it to future work to explore this issue more systematically, and to provide semi-automated tools, e.g. along the lines of AutoCorres [17].

Our code generator, as well as most standard code generators in theorem provers, translates from logic to target language code, implicitly identifying logical concepts with programming language concepts. This approach is simple, however, the translation algorithm and its implementation become part of the trusted code base. More recently, code generators that translate into a deeply embedded semantics of the target language have been developed [40, 21]. We leave a translation to a deep embedding of LLVM to future work, and note that a deep embedding will also enable more advanced control flow constructs like exceptions and breaking from loops, without significantly increasing the trusted code base.

Compared to actual LLVM, Isabelle-LLVM makes a few simplifying assumptions: We do not support floating point arithmetic, though this could be added, e.g. based on Lei Yu's floating point formalization [49]. Moreover, we only support two-element structures (pairs). This nicely fits Isabelle HOL's product datatype, and the nested structures resulting from longer tuples should not be a problem for LLVM's optimizer. Also, we do not support concepts that are handy for program optimization, but not required for code generation, like poison values. Isabelle-LLVM assumes an infinite supply of memory, and thus cannot assign a bit-size to pointers. This assumption helps us to retain a deterministic semantics, which is executable inside the theorem prover (cf. Example 1). We plan to use this feature for systematic testing of our code generator against the actual LLVM compiler. A similar assumption is implicitly made for the stack, as our semantics permits arbitrarily deep recursive procedure calls. We remedy this mismatch between semantics and reality by terminating the program in a defined way if it runs out of heap. To protect against stack overflows, LLVM provides mechanisms like stack probing or split stack, which, however, require some effort to enable. We leave that to future work, and note that our generated code allocates no large blocks of memory on the stack. Thus, stack overflows are likely to hit the guard pages inserted by most operating systems, which will cause defined termination of the process.

Currently, we interface our generated LLVM code from C programs compiled by clang. However, the ABIs of C and LLVM only partially match, and some LLVM constructs cannot be expressed in C at all. Currently, it is the user's responsibility to implement a correct header file. We plan to automatically generate header files and adapter functions to make the exported code accessible from C.

7 Related Work

This project would not have been possible without several independent Isabelle developments: We use the Separation Algebra library [23, 22] as basis for our separation logic. We substantially extended this library by a frame inference heuristics, and formalized the extension of separation algebras over functions, products, and options. Moreover, we use Isabelle's machine word library [2] to model the 2's complement arithmetic of LLVM. We slightly extended this library by adding a few lemmas. Finally, the Eisbach language [38] was a great help for prototyping the verification condition generator, although most of the final VCG is now implemented directly in the more low-level Isabelle/ML.

The Vellvm project [50, 51] verifies LLVM program transformations in Coq. To be useful, e.g. as backend for clang, they have to formalize a substantial fragment of LLVM. On the other hand, we can afford to formalize a simplified and abstract semantics that is just powerful enough to cover what Sepref generates.

We drew some of the ideas for our separation logic from the Verifiable C project [1], a Coq formalization of a separation logic on top of the CompCert C semantics [4].

There exists various formalizations of low-level imperative languages, eg [36, 46]. These are focused on specifying the semantics, and we are not aware of any complex algorithm verifications using these formalizations.

The DeepSpec project [14] aims at a completely verified computation environment, down to machine code, including the operating system. This is much more ambitious than the work presented here, which stops at a (simplified) LLVM semantics. For proving correct imperative programs, they have a separation logic based VCG for a fragment of C [1, 9], which they apply to several small C programs, mainly for cryptographic algorithms.

8 Conclusions

We have developed Isabelle-LLVM, a shallowly embedded imperative language designed to be easily translated to actual LLVM text. On top of this, we have built a verification infrastructure, and re-targeted the Sepref tool to connect the Refinement Framework to LLVM. As case studies, we have generated verified LLVM code for a binary search algorithm and the Knuth-Morris-Pratt string search algorithm. Both implementations are an order of magnitude faster than the ones generated with the original Sepref tool, and on par with unverified C implementations. The additional effort required to refine to LLVM instead of Standard ML was quite low.

References

1 Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, New York, NY, USA, 2014.

2 Joel Beeren, Matthew Fernandez, Xin Gao, Gerwin Klein, Rafal Kolanski, Japheth Lim, Corey Lewis, Daniel Matichuk, and Thomas Sewell. Finite Machine Word Library. *Archive of Formal Proofs*, June 2016. , Formal proof development. URL: `http://isa-afp.org/entries/Word_Lib.html`.

3 Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending Sledgehammer with SMT Solvers. *J. Autom. Reasoning*, 51(1):109–128, 2013. `doi:10.1007/s10817-013-9278-5`.

4 Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009. URL: `http://xavierleroy.org/publi/Clight.pdf`.

5 Joshua Bloch. Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken. URL: `http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html`.

6 Julian Brunner and Peter Lammich. Formal Verification of an Executable LTL Model Checker with Partial Order Reduction. *J. Autom. Reasoning*, 60(1):3–21, 2018. `doi:10.1007/s10817-017-9418-4`.

7 Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. Imperative Functional Programming with Isabelle/HOL. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *TPHOLs 2008*, volume 5170 of *LNCS*, pages 134–149. Springer, 2008.

8 C. Calcagno, P.W. O'Hearn, and Hongseok Yang. Local Action and Abstract Separation Logic. In *LICS 2007*, pages 366–378, July 2007.

9 Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *Journal of Automated Reasoning*, 61, February 2018. doi:10.1007/s10817-018-9457-5.

10 Clang: a C language family frontend for LLVM. URL: https://clang.llvm.org/.

11 Luís Cruz-Filipe, Marijn Heule, Warren Hunt, Matt Kaufmann, and Peter Schneider-Kamp. Efficient Certified RAT Verification. In *Proc. of CADE*. Springer, 2017.

12 Luís Cruz-Filipe, Joao Marques-Silva, and Peter Schneider-Kamp. Efficient Certified Resolution Proof Checking. In *Proc. of TACAS*, pages 118–135. Springer, 2017.

13 Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991. doi:10.1145/115372.115320.

14 Deep Spec Project Web Page. URL: https://deepspec.org/.

15 Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A Fully Verified Executable LTL Model Checker. In *CAV*, volume 8044 of *LNCS*, pages 463–478. Springer, 2013.

16 Mathias Fleury, Jasmin Christian Blanchette, and Peter Lammich. A verified SAT solver with watched literals using Imperative HOL. In *Proc. of CPP*, pages 158–171, 2018.

17 David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. Don't sweat the small stuff: formal verification of C code without the pain. In *Proc. of PLDI '14*, pages 429–439, 2014. doi:10.1145/2594291.2594296.

18 Florian Haftmann, Alexander Krauss, Ondřej Kunčar, and Tobias Nipkow. Data Refinement in Isabelle/HOL. In *Proc. of ITP*, pages 100–115. Springer, 2013.

19 Fabian Hellauer and Peter Lammich. The string search algorithm by Knuth, Morris and Pratt. *Archive of Formal Proofs*, December 2017. , Formal proof development. URL: http://isa-afp.org/entries/Knuth_Morris_Pratt.html.

20 Marijn Heule, Warren Hunt, Matt Kaufmann, and Nathan Wetzler. Efficient, Verified Checking of Propositional Proofs. In *Proc. of ITP*. Springer, 2017.

21 Lars Hupel and Tobias Nipkow. A Verified Compiler from Isabelle/HOL to CakeML. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 999–1026, Cham, 2018. Springer International Publishing.

22 Gerwin Klein, Rafal Kolanski, and Andrew Boyton. Mechanised Separation Algebra. In *ITP*, pages 332–337. Springer, August 2012.

23 Gerwin Klein, Rafal Kolanski, and Andrew Boyton. Separation Algebra. *Archive of Formal Proofs*, May 2012. , Formal proof development. URL: http://isa-afp.org/entries/Separation_Algebra.html.

24 Donald E. Knuth, Jr. James H. Morris, and Vaughan R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(2):323–350, 1977. doi:10.1137/0206024.

25 Alexander Krauss. Recursive definitions of monadic functions. In *Proc. of PAR*, volume 43, pages 1–13, 2010.

26 Peter Lammich. Automatic Data Refinement. In *ITP*, volume 7998 of *LNCS*, pages 84–99. Springer, 2013.

27 Peter Lammich. Refinement to Imperative/HOL. In *ITP*, volume 9236 of *LNCS*, pages 253–269. Springer, 2015.

28 Peter Lammich. Refinement based verification of imperative data structures. In Jeremy Avigad and Adam Chlipala, editors, *CPP 2016*, pages 27–36. ACM, 2016.

29 Peter Lammich. Efficient Verified (UN)SAT Certificate Checking. In *Proc. of CADE*. Springer, 2017.

30 Peter Lammich. The GRAT Tool Chain - Efficient (UN)SAT Certificate Checking with Formal Correctness Guarantees. In *SAT*, pages 457–463, 2017.

31 Peter Lammich and S. Reza Sefidgar. Formalizing the Edmonds-Karp Algorithm. In *Proc. of ITP*, pages 219–234, 2016.

32 Peter Lammich and S. Reza Sefidgar. Formalizing Network Flow Algorithms: A Refinement Approach in Isabelle/HOL. *J. Autom. Reasoning*, 62(2):261–280, 2019. doi:10.1007/s10817-017-9442-4.

33 Peter Lammich and Thomas Tuerk. Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm. In Lennart Beringer and Amy P. Felty, editors, *ITP 2012*, volume 7406 of *LNCS*, pages 166–182. Springer, 2012.

34 Yong Li. Knuth-Morris-Pratt code snippet. URL: https://gist.github.com/yongpitt/5704216.

35 LLVM language reference manual. URL: https://llvm.org/docs/LangRef.html.

36 Andreas Lochbihler. Java and the Java Memory Model - A Unified, Machine-Checked Formalisation. In *Proc. of ESOP*, pages 497–517, 2012. doi:10.1007/978-3-642-28869-2_25.

37 George Markowsky. Chain-complete posets and directed sets with applications. *algebra universalis*, 6(1):53–68, December 1976. doi:10.1007/BF02485815.

38 Daniel Matichuk, Toby Murray, and Makarius Wenzel. Eisbach: A Proof Method Language for Isabelle. *Journal of Automated Reasoning*, 56(3):261–282, March 2016. doi:10.1007/s10817-015-9360-2.

39 MLton. URL: http://mlton.org/.

40 Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *J. Funct. Program.*, 24(2-3):284–315, 2014. doi:10.1017/S0956796813000282.

41 Tobias Nipkow and Gerwin Klein. *Concrete Semantics: With Isabelle/HOL*. Springer Publishing Company, Incorporated, 2014.

42 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

43 John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc of. Logic in Computer Science (LICS)*, pages 55–74. IEEE, 2002.

44 StringBench Benchmark Suite. URL: https://github.com/almondtools/stringbench.

45 Philip Wadler. Theorems for free! In *Proc. of FPCA*, pages 347–359. ACM, 1989.

46 Conrad Watt. Mechanising and verifying the WebAssembly specification. In *Proc. of CPP*, pages 53–65, 2018. doi:10.1145/3167082.

47 Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *SAT 2014*, volume 8561 of *LNCS*, pages 422–429. Springer, 2014.

48 Simon Wimmer and Peter Lammich. Verified Model Checking of Timed Automata. In *TACAS 2018*, pages 61–78, 2018.

49 Lei Yu. A Formal Model of IEEE Floating Point Arithmetic. *Archive of Formal Proofs*, July 2013. , Formal proof development. URL: http://isa-afp.org/entries/IEEE_Floating_Point.html.

50 Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proc. of POPL*, pages 427–440. ACM, 2012. doi:10.1145/2103656.2103709.

51 Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formal Verification of SSA-based Optimizations for LLVM. *SIGPLAN Not.*, 48(6):175–186, June 2013. doi:10.1145/2499370.2462164.

Proof Pearl: Purely Functional, Simple and Efficient Priority Search Trees and Applications to Prim and Dijkstra

Peter Lammich
The University of Manchester, UK

Tobias Nipkow
Technical University Munich, Germany
http://www.in.tum.de/~nipkow

Abstract

The starting point of this paper is a new, purely functional, simple and efficient data structure combining a search tree and a priority queue, which we call a *priority search tree*. The salient feature of priority search trees is that they offer a decrease-key operation, something that is missing from other simple, purely functional priority queue implementations. As two applications of this data structure we verify purely functional, simple and efficient implementations of Prim's and Dijkstra's algorithms. This constitutes the first verification of an executable and even efficient version of Prim's algorithm.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Priority queue, Dijkstra's algorithm, Prim's algorithm, verification, Isabelle

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.23

Supplement Material https://www.isa-afp.org/entries/Priority_Search_Trees.html, https://www.isa-afp.org/entries/Prim_Dijkstra_Simple.html

Funding *Peter Lammich*: DFG Grant LA 3292/1 "Verifizierte Model Checker" and VeTSS grant "Formal Verification of Information Flow Security for Relational Databases"
Tobias Nipkow: Supported by DFG Koselleck grant NI 491/16-1.

1 Introduction

The standard implementations (e.g. [6]) of a number of efficient algorithms (e.g. Prim [26] and Dijkstra [7]) require a priority queue with a decrease-key operation. The latter operation is easy to realize efficiently in an imperative setting but harder in a functional one because one cannot use a pointer into the priority queue. The starting point of this paper is an extremely simple and yet efficient functional data structure that supports the usual search tree operations plus the priority queue operations including decrease-key. It can be realized on top of any kind of binary search tree by augmenting it with priority information. We call it a *priority search tree*. Based on this data structure we implement and verify two classic efficient algorithms, Prim and Dijkstra, in a purely functional manner. This is the first formal verification of an executable version of Prim's algorithm and we discuss its details. The work is carried out in the theorem prover Isabelle/HOL [20, 21].

The paper is structured as follows: Section 2 introduces some Isabelle specific notations. Section 3 presents the ADT of priority maps and its efficient realization via priority search trees. Section 4 introduces undirected graphs. Section 5 details the verification of Prim's algorithm. Finally, Section 6 sketches the verification of Dijkstra's algorithm. The discussion of related work is found in each section.

The Isabelle/HOL sources of the formalizations discussed in this paper are available in the Archive of Formal Proofs [16, 17].

2 Notation

Type variables are denoted by $'a$, $'b$, etc. Most type constructors follow postfix syntax, e.g. τ *set* is the type of sets of elements of type τ. Function types are denoted by the infix $\Rightarrow$. Function update is written as $f(x{:=}y)$.

Type *nat* is the type of natural numbers.

On sets, the unary $-$ is complement and the binary $-$ is difference. The image of a set S under a function f is written f ‘ S.

Lists (type τ *list*) are constructed from the empty list [] via the infix cons-operator (#). The infix (@) concatenates two lists. Function *set* converts a list into a set.

The *option* type is also predefined: **datatype** $'a$ *option* = *None* | *Some* $'a$.

3 Priority Maps and Priority Search Trees

3.1 Priority Maps

A *priority map* is a map from keys (type $'a$) to values (type $'b$) where the values (“priorities”) are linearly ordered and a key with minimal value can be extracted. This is the interface:

$$
\begin{aligned}
&\mathit{empty} :: {'m}\\
&\mathit{update} :: {'a} \Rightarrow {'b} \Rightarrow {'m} \Rightarrow {'m}\\
&\mathit{delete} :: {'a} \Rightarrow {'m} \Rightarrow {'m}\\
&\mathit{is\_empty} :: {'m} \Rightarrow \mathit{bool}\\
&\mathit{lookup} :: {'m} \Rightarrow {'a} \Rightarrow {'b}\ \mathit{option}\\
&\mathit{getmin} :: {'m} \Rightarrow {'a} \times {'b}
\end{aligned}
$$

The first five operations are the canonical ones for maps (which is why we omit their specification); function *update* subsumes decrease-key (which should be called decrease-priority). Function *getmin* extracts the key with the minimal value. Its specification is:

$$
\begin{aligned}
&\mathit{getmin}\ m = (k,\ p) \wedge \mathit{invar}\ m \wedge \neg\ \mathit{lookup}\ m = (\lambda x.\ \mathit{None}) \longrightarrow\\
&\mathit{lookup}\ m\ k = \mathit{Some}\ p \wedge (\forall\, p' {\in} \mathit{ran}\ (\mathit{lookup}\ m).\ p \leq p')
\end{aligned}
$$

where *invar* is the representation invariant and $\mathit{ran}\ m = \{b \mid \exists\, a.\ m\ a = \mathit{Some}\ b\}$ is the range of a map.

3.2 Priority Search Trees

The first contribution of this paper is a truly simple implementation of priority maps by means of augmented binary search trees. That is, the basic data structure is some arbitrary binary search tree, e.g. a red-black tree, implementing the map from $'a$ to $'b$ by storing pairs (k,p) in each node. At this point we need to assume that the keys are also linearly ordered. To implement *getmin* efficiently we annotate/augment each node with another pair (k',p'), the intended result of *getmin* when applied to that subtree. The specification of *getmin* tells us that (k',p') must be in that subtree and that p' is the minimal priority in that subtree. Thus the annotation can be computed by passing the (k',p') with the minimal p' up the tree. We will now make this more precise for balanced binary trees in general.

We assume that our trees are either leaves of the form $\langle\rangle$ or nodes of the form $\langle l,\ kp,\ b,\ r\rangle$ where l and r are subtrees, kp is the contents of the node (a key-priority pair) and b is some additional balance information (e.g. colour, height, size, ...). Augmented nodes are of the form $\langle l,\ kp,\ (b,\ kp'),\ r\rangle$.

The implementation of *getmin* is trivial: $getmin\ \langle \_,\ \_,\ (\_,\ kp'),\ \_\rangle = kp'$. It remains to upgrade the existing map operations to work with augmented nodes. Therefore we now show how to transform any function definition on un-augmented trees into one on trees augmented with (k',p') pairs. A defining equation $f\ pats = e$ for the original type of nodes is transformed into an equation $f\ pats' = e'$ on the augmented type of nodes as follows:

- Every pattern $\langle l,\ kp,\ b,\ r\rangle$ in $pats$ and e is replaced by $\langle l,\ kp,\ (b,\ \_),\ r\rangle$ to obtain $pats'$ and e_2.
- To obtain e', every expression $\langle l,\ kp,\ b,\ r\rangle$ in e_2 is replaced by $node\ l\ kp\ b\ r$ where

$$
\begin{aligned}
&node\ l\ a\ c\ r = \langle l,\ a,\ (c,\ min\_kp\ a\ l\ r),\ r\rangle\\
&min\_kp\ kp\ l\ r =\\
&(\mathsf{case}\ (l,\ r)\ \mathsf{of}\ (\langle\rangle,\ \langle\rangle) \Rightarrow kp\\
&\ \mid (\langle\rangle,\ \langle l_2,\ a_2,\ (b_2,\ kp_2),\ r_2\rangle) \Rightarrow min2\ kp\ kp_2\\
&\ \mid (\langle l_1,\ a_1,\ (b_1,\ kp_1),\ r_1\rangle,\ \langle\rangle) \Rightarrow min2\ kp\ kp_1\\
&\ \mid (\langle l_1,\ a_1,\ (b_1,\ kp_1),\ r_1\rangle,\ \langle l_2,\ a_2,\ (b_2,\ kp_2),\ r_2\rangle) \Rightarrow\\
&\quad min2\ kp\ (min2\ kp_1\ kp_2))\\
&min2 = (\lambda(k,\ p)\ (k',\ p').\ \mathsf{if}\ p \le p'\ \mathsf{then}\ (k,\ p)\ \mathsf{else}\ (k',\ p'))
\end{aligned}
$$

Note that this transformation does not affect the asymptotic complexity of f. Therefore the priority search tree operations have the same complexity as the underlying search tree operations, i.e. typically logarithmic (*update*, *delete*, *lookup*) and constant time (*empty*, *is_empty*). For brevity we simply speak of *efficient* in the rest of the paper.

As an example, consider red-black trees where the balancing information b is one of the two colours *Red* or *Black*. In the functional definition of red-black trees due to Okasaki [25] there is a *balance* function that eliminates red-red configurations. We consider a slight variant *baliL* [14] where one of the defining equations is

$$baliL\ (R\ (R\ t_1\ a_1\ t_2)\ a_2\ t_3)\ a_3\ t_4 = R\ (B\ t_1\ a_1\ t_2)\ a_2\ (B\ t_3\ a_3\ t_4)$$

where $R\ l\ a\ r = \langle l,\ a,\ Red,\ r\rangle$ and $B\ l\ a\ r = \langle l,\ a,\ Black,\ r\rangle$. The transformed version of this equation is

$$
\begin{aligned}
&baliL\ (R\ (R\ t_1\ a_1\ \_\ t_2)\ a_2\ \_\ t_3)\ a_3\ t_4 =\\
&node\ (node\ t_1\ a_1\ Black\ t_2)\ a_2\ Red\ (node\ t_3\ a_3\ Black\ t_4)
\end{aligned}
$$

where $R\ l\ a\ kp\ r = \langle l,\ a,\ (Red,\ kp),\ r\rangle$.

We obtained a priority search map based on red-black trees via the above transformations. The correctness proofs could be transformed incrementally, too. The main idea is to augment the data structure invariant to say that the annotations are correct as well. Function *invpst* expresses this property: $invpst\ \langle\rangle = True$ and

$$
\begin{aligned}
&invpst\ \langle l,\ kp,\ (\_,\ kp'),\ r\rangle =\\
&(invpst\ l \wedge invpst\ r \wedge is\_min2\ kp'\ (set\ (inorder\ l\ @\ kp\ \#\ inorder\ r)))
\end{aligned}
$$

where *set* (...) yields all key-priority bindings and *is_min2* asserts that kp' is minimal amongst them: $is\_min2\ kp'\ KP = (kp' \in KP \wedge (\forall kp{\in}KP.\ snd\ kp' \le snd\ kp))$.

It is straightforward to show $invpst\ (node\ l\ a\ c\ r) = (invpst\ l \wedge invpst\ r)$, and this easily discharges the additional proof obligations when transforming the correctness proof.

3.3 Related Work

Our priority map ADT is close to Hinze's [12] *priority search queue* interface, except that he also supports a few further operations that we could easily add but do not need for our applications. However, it is not clear if his implementation technique is the same as our priority search tree because his description employs a plethora of concepts, e.g. *priority search pennants*, *tournament trees*, *semi-heaps*, and multiple *views* of data types that obscure a direct comparison. We claim that at the very least our presentation is new because it is much simpler; we encourage the reader to compare the two.

As already observed by Hinze, McCreight's [19] priority search trees support range queries more efficiently than our trees. However, we can support the same range queries as Hinze efficiently, but that is outside the scope of the paper.

4 Undirected Graphs

There seems to be no single best way of how to represent undirected graphs in Isabelle/HOL. One variant is to represent an undirected graph as a symmetric relation, i.e., an entity of type $('v \times 'v)\ set$. The advantage is that many existing theory of symmetric relations can be reused. However, edges in a symmetric relation are pairs, i.e., they are naturally directed: For $u \neq v$, we have $(u,v) \neq (v,u)$, although, when interpreted as undirected edges, the two should be identified. The same issue transfers to derived concepts like paths in between two nodes.

Another option is to use undirected pairs or doubleton sets to represent an edge. The advantage is that HOL's equality on edges matches the natural equality. However, one cannot re-use the well-established theory of relations then, and has to develop many basic concepts from scratch.

For this formalization, we use a hybrid approach, which tries to combine the advantages of both, and being lightweight at the same time: A graph is represented as a symmetric relation, and only when equality on edges is required, they are converted to doubleton sets on the spot.

We start by defining the type of an undirected graph to be a finite, symmetric, and irreflexive relation together with a set of nodes, which must cover the domain of the relation:

typedef $'v\ ugraph = \{\ (V::'v\ set\ ,\ E).\ E \subseteq V \times V \wedge finite\ V \wedge sym\ E \wedge irrefl\ E\ \}$

Next, we define accessor functions to obtain the nodes and edges of a graph:

$nodes::'v\ ugraph \Rightarrow 'v\ set \qquad edges::'v\ ugraph \Rightarrow ('v \times 'v)\ set$

We also define functions to construct a graph from its nodes and edges, to insert an edge into a graph, and to restrict the edges of a graph:

$graph::'v\ set \Rightarrow ('v \times 'v)\ set \Rightarrow 'v\ ugraph$
$ins\_edge::'v \times 'v \Rightarrow 'v\ ugraph \Rightarrow 'v\ ugraph$
$restrict\_edges::'v\ ugraph \Rightarrow ('v \times 'v)\ set \Rightarrow 'v\ ugraph$

Note that *graph* forms the symmetric closure of the edges, ignores reflexive edges, and adds missing nodes:

$finite\ V \wedge finite\ E \longrightarrow$
$nodes\ (graph\ V\ E) = V \cup fst\ `\ E \cup snd\ `\ E \wedge edges\ (graph\ V\ E) = E \cup E^{-1} - Id$

Similarly, *ins_edge* also inserts the nodes of the edge, and *restrict_edges* removes all edges not in the symmetric closure of the given set.

A *path* is a list of (directed) edges between two nodes:

$$path\ g\ u\ []\ v = (u = v)$$
$$path\ g\ u\ (e\ \#\ ps)\ w = (\exists v.\ e = (u, v) \wedge e \in edges\ g \wedge path\ g\ v\ ps\ w)$$

As the edge relation is symmetric, every path induces a reversed path:

$$path\ g\ u\ (revp\ p)\ v = path\ g\ v\ p\ u \qquad \text{where } revp\ p = rev\ (map\ (\lambda(u, v).\ (v, u))\ p)$$

Obviously, existence of a path between two nodes is equivalent to these nodes being in the reflexive transitive closure of the edge relation:

$$(\exists p.\ path\ g\ u\ p\ v) = ((u, v) \in (edges\ g)^*) \qquad \text{where } \_^* \text{ is reflexive transitive closure}$$

We call a graph *connected*, if there exists path between all its nodes:

$$connected\ g = (nodes\ g \times nodes\ g \subseteq (edges\ g)^*)$$

A *simple path* does not contain any edge twice. Here, we need to consider undirected edges. Thus, we define a function $uedge :: 'a \times 'a \Rightarrow 'a\ set$ to map an edge to a doubleton set.

$$simple\ p = distinct\ (map\ uedge\ p) \qquad \text{where } uedge = (\lambda(a, b).\ \{a, b\})$$

A *cycle* is a simple, non-empty path with the same start and end node. We define a predicate for *cycle-free* graphs:

$$cycle\_free\ g = (\nexists p\ u.\ p \neq [] \wedge u \in nodes\ g \wedge simple\ p \wedge path\ g\ u\ p\ u)$$

A *tree* is a connected and cycle free graph:

$$tree\ g = (connected\ g \wedge cycle\_free\ g)$$

A *spanning tree* of a graph is a tree with the same nodes and a subset of the edges:

$$is\_spanning\_tree\ G\ T = (tree\ T \wedge nodes\ T = nodes\ G \wedge edges\ T \subseteq edges\ G)$$

Every connected graph has a spanning tree:

$$connected\ g \longrightarrow (\exists t.\ is\_spanning\_tree\ g\ t)$$

which is proved by removing edges on cycles until the graph is cycle free.

We model *weighted graphs* as graphs with a function $w :: 'v\ set \Rightarrow nat$ from (doubleton) sets to natural numbers[1]. Note that we do not need to restrict the domain of the weight function to be doubleton sets of valid nodes – the values for invalid nodes or sets will just be ignored. We then define the *weight* of a graph as the sum of the weights of all its edges:

$$weight\ w\ g = sum\ w\ (uedge\ `\ edges\ g)$$

A *minimum spanning tree* (MST) of a graph g is a spanning tree with minimal weight:

$$is\_MST\ w\ g\ t =$$
$$(is\_spanning\_tree\ g\ t \wedge (\forall t'.\ is\_spanning\_tree\ g\ t' \longrightarrow weight\ w\ t \leq weight\ w\ t'))$$

Obviously, each connected graph has a minimum spanning tree:

$$connected\ g \longrightarrow (\exists t.\ is\_MST\ w\ g\ t)$$

[1] For simplicity of presentation, we restrict weights to be natural numbers. Lifting this restriction is straightforward.

4.1 Related Work

There is a plethora of approaches to modelling graphs. Most formalizations focus on *directed* graphs. A typical example is Noschinski [24] (who should be consulted for a more detailed review of related work): he models undirected graphs as symmetric (bidirectional) directed graphs. Abstractly, this is also what Chou [5] does, who was the first to formalize undirected graphs. In both approaches, there is an explicit type of edges. Our approach is at the minimalist end: we avoid a separate edge type but use pairs of nodes. This means that we can use pattern matching on pairs in addition to projection functions but it also means that we cannot have multi-edges, something we don't need in our applications.

4.2 An Interface for Undirected Graphs

We have defined a representation of undirected weighted graphs in Isabelle/HOL. The next step towards an implementation is to specify an interface for the operations on undirected weighted graphs. We fix an implementation type $'g$, and an invariant $invar :: 'g \Rightarrow bool$, as well as two abstraction functions, one for the graph, and one for the weights:

$$\alpha g :: 'g \Rightarrow 'v\ ugraph \qquad \alpha w :: 'g \Rightarrow 'v\ set \Rightarrow nat$$

We specify operations to get the adjacent edges of a node, to create an empty graph, and to add an edge to a graph (implicitly adding the endpoints as nodes).

$$adj :: 'g \Rightarrow 'v \Rightarrow ('v \times nat)\ list \qquad empty :: 'g \qquad add\_edge :: 'v \times 'v \Rightarrow nat \Rightarrow 'g \Rightarrow 'g$$

Note that these are exactly the operations required for our purpose of formalizing Prim's algorithm. More operations can easily be added. The specifications for the operations are:

$$invar\ g \longrightarrow$$
$$set\ (adj\ g\ u) = \{(v,\ d) \mid (u,\ v) \in edges\ (\alpha g\ g) \wedge \alpha w\ g\ \{u,\ v\} = d\}$$

$$invar\ empty \wedge \alpha g\ empty = graph\ \emptyset\ \emptyset \wedge \alpha w\ empty = (\lambda\_.\ 0)$$

$$invar\ g \wedge (u,\ v) \notin edges\ (\alpha g\ g) \wedge u \neq v \longrightarrow$$
$$invar\ (add\_edge\ (u,\ v)\ d\ g) \wedge$$
$$\alpha g\ (add\_edge\ (u,\ v)\ d\ g) = ins\_edge\ (u,\ v)\ (\alpha g\ g) \wedge$$
$$\alpha w\ (add\_edge\ (u,\ v)\ d\ g) = (\alpha w\ g)(\{u,\ v\} := d)$$

That is $adj\ g\ u$ returns a list of pairs of nodes and weights, corresponding to the adjacent edges of node u, *empty* creates an empty graph, and *add_edge* inserts a (new) edge.

Note that designing interfaces often involves a trade-off between usability and implementability. We now motivate some of our design decisions:

- We leave the order of the adjacency list unspecified and allow duplicates. This introduces nondeterminism, and thus makes using the interface more complex. However, an (abstractly) fixed order on the adjacency list can only be implemented when the node type is linearly ordered, and even then, it incurs unnecessary overhead due to sorting.
- The node passed to *adj* needs not be a node of the graph (The returned list is empty for non-nodes). This makes the interface easier to use, as there is one precondition less to prove. Moreover, the implementation is straightforward.
- The weight function of the empty graph is fixed to $\lambda\_.\ 0$. This makes the specification deterministic, and thus simpler to use, and can be easily implemented.

4.3 Parsing Graphs from Lists

Based on the graph interface, we develop an algorithm to create a graph from a list of weighted edges. The elements of the list have the form $((u, v), d)$, describing an edge between u and v with weight d:

$$from\_list\ l = foldr\ (\lambda(e, d).\ add\_edge\ e\ d)\ l\ empty$$

We show that, for a valid list l, a graph implementation gi will be created that satisfies its invariant, and whose abstraction (g,w) contains exactly the nodes, edges, and weights contained in the list:

$$\begin{array}{l} G\_valid\_wgraph\_repr\ l \longrightarrow \\ (\textsf{let}\ gi = from\_list\ l;\ g = \alpha g\ gi;\ w = \alpha w\ gi \\ \quad \textsf{in}\ invar\ gi \wedge nodes\ g = \bigcup \{\{u, v\} \mid \exists d.\ ((u, v), d) \in set\ l\} \wedge \\ \qquad edges\ g = \bigcup \{\{(u, v), (v, u)\} \mid \exists d.\ ((u, v), d) \in set\ l\} \wedge \\ \qquad (\forall ((u, v), d) \in set\ l.\ w\ \{u, v\} = d)) \end{array}$$

Here, a list is valid if no edge is specified twice and there are no reflexive edges:

$$\begin{array}{l} G\_valid\_wgraph\_repr\ l = \\ ((\forall ((u, v), d) \in set\ l.\ u \neq v) \wedge distinct\ (map\ (\lambda((u, v), d).\ \{u, v\})\ l)) \end{array}$$

5 Verifying Prim's Algorithm

Prim's algorithm [26] is a classical algorithm to find a minimum spanning tree of an undirected graph. In this section we describe our formalization of Prim's algorithm, roughly following the presentation of Cormen et al. [6].

Our approach features stepwise refinement. We start by a generic MST algorithm (Section 5.1) that covers both Prim's and Kruskal's algorithms. It maintains a subgraph A of an MST. Initially, A contains no edges and only the root node. In each iteration, the algorithm adds a new edge to A, maintaining the property that A is a subgraph of an MST. In a next refinement step, we only add edges that are adjacent to the current A, thus maintaining the invariant that A is always a tree (Section 5.2). Next, we show how to use a priority queue to efficiently determine a next edge to be added (Section 5.3), and implement the necessary update of the priority queue using a foreach-loop (Section 5.4). Finally we parameterize our algorithm over ADTs for graphs, maps, and priority queues (Section 5.5), instantiate these with actual data structures, and extract executable ML code (Section 5.6).

The advantage of this stepwise refinement approach is that the proof obligations of each step are mostly independent from the other steps. This modularization greatly helps to keep the proof manageable. Moreover, the steps also correspond to a natural split of the ideas behind Prim's algorithm: The same structuring is also done in the presentation of Cormen et al. [6], though not as detailed as ours.

5.1 Generic MST Algorithm

For the rest of this section, $g::'v\ ugraph$ will be an undirected graph, $r \in nodes\ g$ will be the root node identifying the connected component of the graph for which we want to compute the minimum spanning tree, and $w::'v\ set \Rightarrow nat$ will be a weight function.

Once we have fixed a root node, we can define the reachable part of the graph[2]:

$rg = ins\_node\ r\ (restrict\_nodes\ g\ ((edges\ g)^*\ ``\ \{r\}))$

Cormen et al. [6] describe Prim's algorithm as an instance of a more generic algorithm, which maintains a subgraph A of a minimum spanning tree. The graph is grown by repeatedly adding *safe edges*, i.e., edges that preserve the property of A being a subgraph of a minimum spanning tree.

$is\_subset\_MST\ w\ g\ A = (\exists t.\ is\_MST\ w\ g\ t \wedge A \subseteq edges\ t)$

Note that, like Cormen et al., we represent the current subgraph by a set of directed edges $A{::}('v \times 'v)\ set$.

The central idea of the generic algorithm provides a way to find a safe edge: A cut $(C,\ nodes\ g - C)$ is a partitioning of the nodes. A subgraph *respects* a cut, if none of its edges cross the cut:

$respects\_cut\ A\ C = (A \subseteq C \times C \cup (-\ C) \times -\ C)$

An edge (u,v) is *light* w.r.t. a cut $(C,\ nodes\ g - C)$ if it *crosses* C (wlog. $u \in C \wedge v \notin C$), and its weight is minimal among all edges crossing C:

$light\_edge\ C\ u\ v =$
$(u \in C \wedge v \notin C \wedge (u,\ v) \in edges\ rg \wedge$
$(\forall (u',\ v') \in edges\ rg \cap C \times -\ C.\ w\ \{u,\ v\} \leq w\ \{u',\ v'\}))$

Given a cut that is respected by the current subgraph, light edges are safe:

$is\_subset\_MST\ w\ rg\ A \wedge respects\_cut\ A\ C \wedge light\_edge\ C\ u\ v \longrightarrow$
$is\_subset\_MST\ w\ rg\ (\{(v,\ u)\} \cup A)$

5.2 Prim's Algorithm

Prim's algorithm maintains a connected graph, i.e., A forms a tree. It starts with the singleton tree containing no edges and only the root node, and then repeatedly adds light edges connecting a node of the tree with a node not yet in the tree. Note that the nodes of the current tree can be defined from A as $S\ A = \{r\} \cup fst\ `\ A \cup snd\ `\ A$. Obviously, they form a cut respected by A: $respects\_cut\ A\ (S\ A)$.

Figure 1 shows the abstract algorithm in pseudocode: As long as the current nodes $S\ A$ are not closed under the edge relation, we pick a light edge (wrt. the cut $S\ A$), and add it to A. In order to prove this algorithm correct, we have to specify an invariant and a measure function, and show that the invariant holds initially, is preserved by a loop iteration, and implies that the result is a minimum spanning tree when the loop terminates. Moreover, we have to show that the measure decreases in every loop iteration.

As measure, we use the number of nodes that are *not* in $S\ A$:

$T\_measure1\ A = card\ (nodes\ rg - S\ A)$

The invariant states that A is a subgraph of a minimum spanning tree, and that all nodes of A are connected to the root node:

[2] Cormen at al. [6] assume that the graph is connected. Our setting is slightly more general. In particular, it saves us from checking a connectedness precondition.

```
A := {}
while S A not closed under edges
  choose edge (u,v) with u ∉ S A and v ∈ S A such that w {u,v} is minimal
  A := {(u,v)} ∪ A
```

Figure 1 Pseudocode of the abstract version of Prim's Algorithm.

$$prim\_invar1\ A = (is\_subset\_MST\ w\ rg\ A \wedge (\forall (u,\ v) \in A.\ (v,\ r) \in A^*))$$

The following theorems formalize invariant initialization, maintenance, and termination with the correct result for the abstract algorithm:

$$prim\_invar1\ \emptyset$$

$$prim\_invar1\ A \wedge light\_edge\ (S\ A)\ u\ v \longrightarrow prim\_invar1\ (\{(v,\ u)\} \cup A) \wedge T\_measure1\ (\{(v,\ u)\} \cup A) < T\_measure1\ A$$

$$prim\_invar1\ A \wedge edges\ g \cap S\ A \times - S\ A = \emptyset \longrightarrow is\_MST\ w\ rg\ (graph\ \{r\}\ A)$$

Recall that *graph* forms the symmetric closure of the edges and adds missing nodes.

5.3 Using a Priority Queue

To efficiently find a next edge to be added, Prim's algorithm maintains a priority queue $Q::'v \Rightarrow enat$ and a predecessor map $\pi::'v \Rightarrow 'v\ option$, where *enat* is the type of natural numbers with ∞ and $enat :: nat \Rightarrow enat$ is the canonical injection. A node u is *adjacent* to a set of nodes S, iff $u \notin S$ and there is an edge connecting u to some node in S.

For every node u that is adjacent to the current tree, $Q\ u$ stores the minimum weight of all edges connecting u to the tree. Moreover, $\pi\ u$ is the other endpoint of this edge. Additionally, we use π to store the edges of the current subtree itself: If $\pi\ u = Some\ v$, and $Q\ u = \infty$, i.e., the node u is already in the tree, then $(u,\ v)$ is an edge of the current subtree: $A\ Q\ \pi = \{(u,\ v) \mid \pi\ u = Some\ v \wedge Q\ u = \infty\}$.

Note that the implementation of Cormen et al. slightly differs from ours: Our priority queue only stores nodes that are *adjacent* to $S\ A$, while theirs stores *all* nodes not in $S\ A$. In their implementation, the priority queue has to be initialized with all (reachable) nodes of the graph, while we only need to initialize the queue for the root node. This simplifies the implementation as it saves an iteration over the graph's node.

A step of the algorithm extracts a node u from Q with minimum priority, and then updates the priorities for all adjacent nodes. The priority (and predecessor) of a node v' has to be updated if v' is adjacent to u, outside S, and the weight of the edge (u,v') is less than the weight currently stored in $Q\ v'$:

$$upd\_cond\ Q\ \pi\ u\ v' = ((v',\ u) \in edges\ g \wedge v' \notin S\ (A\ Q\ \pi) \wedge enat\ (w\ \{v',\ u\}) < Q\ v')$$

In this case, we update both, $Q\ v'$ and $\pi\ v'$:

$$Qinter\ Q\ \pi\ u\ v' = (\textsf{if}\ upd\_cond\ Q\ \pi\ u\ v'\ \textsf{then}\ enat\ (w\ \{v',\ u\})\ \textsf{else}\ Q\ v')$$
$$Q'\ Q\ \pi\ u = (Qinter\ Q\ \pi\ u)(u := \infty)$$
$$\pi'\ Q\ \pi\ u\ v' = (\textsf{if}\ upd\_cond\ Q\ \pi\ u\ v'\ \textsf{then}\ Some\ u\ \textsf{else}\ \pi\ v')$$

```
Q := (λ_. ∞)(r:=0); π := (λ_. None)
while Q ≠ (λ_. ∞)
  pick u such that Q u is minimal
  Q := Q' Q π u; π := π' Q π u
  Q u := ∞
```

Figure 2 Pseudocode of Prim's Algorithm using a Priority Queue.

Note that we define Q' in two steps: *Qinter* is the priority queue where adjacent nodes have been updated, but the node u has not yet been removed. This definition is motivated by our implementation, which first iterates over the adjacent nodes, and then removes u from Q[3]. The refined algorithm is displayed in Figure 2.

Note that the refined algorithm starts with a priority queue that only contains the root node r. Intuitively, this corresponds to a tree that contains no nodes at all, not even the root node. In particular, it does not correspond to the initial state of the abstract algorithm. Only after the first loop iteration, the priority queue is initialized for the nodes adjacent to r. Thus, the refined state after the first iteration corresponds to the abstract initial state. We accommodate for this discrepancy in the invariant and variant of the refined loop:

$$prim\_invar2\ Q\ \pi = (prim\_invar2\_init\ Q\ \pi \vee prim\_invar2\_ctd\ Q\ \pi)$$
$$T\_measure2\ Q\ \pi = (\textsf{if}\ Q\ r = \infty\ \textsf{then}\ T\_measure1\ (A\ Q\ \pi)\ \textsf{else}\ card\ (nodes\ rg))$$

Here, *prim_invar2_init* states that Q and π are in their initial states, and *prim_invar2_ctd* states that the abstract invariant holds for the abstracted state $A\ Q\ \pi$, and, additionally, some consistency properties on Q and π:

▶ **Definition 1.** *Let (Q, π) be the refined state of the algorithm. Moreover, let A be the corresponding abstract state, S be the nodes of the current subtree, and $cE = edges\ rg \cap (-S) \times S$ be the set of edges crossing S. Then, prim_invar2_ctd $Q\ \pi$ states that*

1. *the abstract invariant holds: prim_invar1 $w\ g\ r\ A$*
2. *the root node has no predecessor and is not in Q: $\pi\ r = None \wedge Q\ r = \infty$*
3. *the outside node of any crossing edge is in Q: $\forall (u, v) \in cE.\ Q\ u \neq \infty$*
4. *π encodes actual edges with target nodes in S:*

   $$\forall u\ v.\ \pi\ u = Some\ v \longrightarrow v \in S \wedge (u, v) \in edges\ rg$$

5. *$Q\ u$ stores the weight of the corresponding edge in π, and this is the minimum weight of all crossing edges from u:*

   $$\forall u\ d.\ Q\ u = enat\ d \longrightarrow (\exists v.\ \pi\ u = Some\ v \wedge d = w\ \{u, v\} \wedge (\forall v'.\ (u, v') \in cE \longrightarrow d \leq w\ \{u, v'\}))$$

Note that the first and second part of the invariant mutually exclude each other:

[3] Although non-standard, we chose this implementation because it slightly simplifies the proofs: When further refining the update, we can simply assume that the loop invariant holds. In the standard implementation, which removes u before the update, we would have to define an assertion that describes the state after the removal.

$prim\_invar2\_init\ Q\ \pi \longrightarrow \neg\ prim\_invar2\_ctd\ Q\ \pi$ Proof: Consider value of $Q\ r$.

Thus, the following lemmas imply correctness of the refined algorithm:

$prim\_invar2\_init\ initQ\ init\pi$

$prim\_invar2\_init\ Q\ \pi \wedge Q\ u = enat\ d \longrightarrow$
$prim\_invar2\_ctd\ (Q'\ Q\ \pi\ u)\ (\pi'\ Q\ \pi\ u) \wedge$
$T\_measure2\ (Q'\ Q\ \pi\ u)\ (\pi'\ Q\ \pi\ u) < T\_measure2\ Q\ \pi$

$prim\_invar2\_ctd\ Q\ \pi \wedge Q\ u = enat\ d \wedge (\forall v.\ enat\ d \leq Q\ v) \longrightarrow$
$prim\_invar2\_ctd\ (Q'\ Q\ \pi\ u)\ (\pi'\ Q\ \pi\ u) \wedge$
$T\_measure2\ (Q'\ Q\ \pi\ u)\ (\pi'\ Q\ \pi\ u) < T\_measure2\ Q\ \pi$

$prim\_invar2\_ctd\ Q\ \pi \wedge Q = (\lambda\_.\ \infty) \longrightarrow$
$is\_MST\ w\ rg\ (graph\ \{r\}\ \{(u, v) \mid \pi\ u = Some\ v\})$

5.4 Inner Foreach Loop

As a next step towards an efficiently executable implementation, we implement Q' and π' by iterating over the nodes adjacent to u. We assume that $adjs::('v \times nat)\ list$ is the adjacency list of node u, and define:

$foreach\ u\ adjs\ (Q, \pi) =$
$foldr$
$(\lambda(v, d)\ (Q, \pi).$
$\quad$ *if* $v \neq r \wedge (\pi\ v = None \vee Q\ v \neq \infty) \wedge enat\ d < Q\ v$
$\quad$ *then* $(Q(v := enat\ d), \pi(v \mapsto u))$ *else* $(Q, \pi))$
$adjs\ (Q, \pi)$

where $f(x{\mapsto}y)$ is short for $f(x{:=}Some\ y)$. This updates Q and π only for adjacent nodes, with a smaller associated weight than that currently stored in Q. We show that this implementation computes the correct result:

$set\ adjs = \{(v, d) \mid (u, v) \in edges\ g \wedge w\ \{u, v\} = d\} \longrightarrow$
$foreach\ u\ adjs\ (Q, \pi) = (Qinter\ Q\ \pi\ u, \pi'\ Q\ \pi\ u)$

In order to express Q and π after some but not all adjacent nodes have been processed, we need to generalize the statement accordingly. We define

$Qigen\ Q\ \pi\ u\ adjs\ v = (\textit{if}\ v \notin fst\ `\ set\ adjs\ \textit{then}\ Q\ v\ \textit{else}\ Qinter\ Q\ \pi\ u\ v)$
$\pi'gen\ Q\ \pi\ u\ adjs\ v = (\textit{if}\ v \notin fst\ `\ set\ adjs\ \textit{then}\ \pi\ v\ \textit{else}\ \pi'\ Q\ \pi\ u\ v)$

and prove

$set\ adjs \subseteq \{(v, d) \mid (u, v) \in edges\ g \wedge w\ \{u, v\} = d\} \longrightarrow$
$foreach\ u\ adjs\ (Q, \pi) = (Qigen\ Q\ \pi\ u\ adjs, \pi'gen\ Q\ \pi\ u\ adjs)$

by induction on the adjacency list $adjs$.

5.5 Data Structures

The next step towards an executable algorithm is to implement the graph, priority queue, and predecessor map by actual data structures. We do this in a two-step approach: First, we implement the algorithm parameterized over the interfaces of graphs, maps, and priority maps, and, in a second step, we instantiate these interfaces to actual data structures. This approach has two advantages: First, it is easy to exchange the used data structures by simply exchanging the instantiation. Second, not knowing the actual data structures when proving the implementation correct prevents accidental breaking of the interface and looking into data structure details. Note that this actually happens in practice, e.g., due to "forgotten" simplifier setup, or automated tools like sledgehammer, which do not know about interfaces.

For Prim's algorithm, we fix the interfaces of an undirected weighted graph (cf. Section 4.2), a map, and a priority map (cf. Section 3). The interface functions are prefixed with G, M, and Q, respectively, and the implementation types are $'g$, $'m$, and $'q$:

$G\_\alpha w :: 'g \Rightarrow 'v\ set \Rightarrow nat$ $\quad$ $G\_\alpha g :: 'g \Rightarrow 'v\ ugraph$ $\quad$ $G\_invar :: 'g \Rightarrow bool$
$G\_adj :: 'g \Rightarrow 'v \Rightarrow ('v \times nat)\ list$ $\quad$ $G\_empty :: 'g$
$G\_add\_edge :: 'v \times 'v \Rightarrow nat \Rightarrow 'g \Rightarrow 'g$

$M\_lookup :: 'm \Rightarrow 'v \Rightarrow 'v\ option$ $\quad$ $M\_invar :: 'm \Rightarrow bool$ $\quad$ $M\_empty :: 'm$
$M\_update :: 'v \Rightarrow 'v \Rightarrow 'm \Rightarrow 'm$ $\quad$ $M\_delete :: 'v \Rightarrow 'm \Rightarrow 'm$

$Q\_lookup :: 'q \Rightarrow 'v \Rightarrow nat\ option$ $\quad$ $Q\_invar :: 'q \Rightarrow bool$ $\quad$ $Q\_empty :: 'q$
$Q\_update :: 'v \Rightarrow nat \Rightarrow 'q \Rightarrow 'q$ $\quad$ $Q\_delete :: 'v \Rightarrow 'q \Rightarrow 'q$ $\quad$ $Q\_is\_empty :: 'q \Rightarrow bool$
$Q\_getmin :: 'q \Rightarrow 'v \times nat$

For the rest of this section, we also fix a graph $g :: 'g$ and root node $r :: 'v$.

At this point of the formalization, we can actually define Prim's algorithm as a functional program. On the previous abstraction levels, this was not possible because functional programs in Isabelle/HOL must be deterministic. However, when, e.g., extracting a minimum element from a priority queue, we cannot define any tie-breaking in terms of the abstract representation $Q :: 'v \Rightarrow enat$, as the actual tie-breaking will depend on the data structure that is used. The same holds for the order in which the foreach loop iterates over the list of adjacent nodes. Figure 3 shows our implementation of the algorithm. It uses the *while* combinator, which obeys the following recursion equation:

$while\ b\ c\ s = (\mathsf{if}\ b\ s\ \mathsf{then}\ while\ b\ c\ (c\ s)\ \mathsf{else}\ s)$

In HOL, tail-recursive functions can always be defined, regardless of termination [2].

Like for the other abstraction levels, we define an invariant and a measure function:

$prim\_invar\_impl\ Qi\ \pi i =$
$(Q\_invar\ Qi \wedge M\_invar\ \pi i \wedge prim\_invar2\ (Q\_\alpha\ Qi)\ (M\_lookup\ \pi i))$
$T\_measure\_impl = (\lambda(Qi, \pi i).\ T\_measure2\ (Q\_\alpha\ Qi)\ (M\_lookup\ \pi i))$

where Q_α abstracts the priority map to the type $'v \Rightarrow enat$, mapping *None* to ∞, and *M_lookup* abstracts the predecessor map to the type $'v \Rightarrow 'v\ option$.

Again, we show invariant initialization, maintenance, and termination with the correct result:

$prim\_invar\_impl\ (Q\_update\ r\ 0\ Q\_empty)\ M\_empty$

$prim\_invar\_impl\ Qi\ \pi i \wedge \neg\ Q\_is\_empty\ Qi \wedge Q\_getmin\ Qi = (u,\ d) \wedge$

$$\begin{array}{l}
\mathit{foreach_impl}\ \mathit{Qi}\ \pi i\ u\ (\mathit{G_adj}\ g\ u) = (\mathit{Qi}', \pi i') \longrightarrow \\
\mathit{prim_invar_impl}\ (\mathit{Q_delete}\ u\ \mathit{Qi}')\ \pi i' \wedge \\
\mathit{T_measure_impl}\ (\mathit{Q_delete}\ u\ \mathit{Qi}', \pi i') < \mathit{T_measure_impl}\ (\mathit{Qi}, \pi i)
\end{array}$$

$$\begin{array}{l}
\mathit{Q_is_empty}\ Q \wedge \mathit{prim_invar_impl}\ Q\ \pi \longrightarrow \\
\mathit{M_invar}\ \pi \wedge \mathit{is_MST}\ (\mathit{G_}\alpha w\ g)\ \mathit{rg}\ (\mathit{graph}\ \{r\}\ \{(u, v) \mid \mathit{M_lookup}\ \pi\ u = \mathit{Some}\ v\})
\end{array}$$

Here, *foreach_impl* stands for the inner foreach loop:

$$\mathit{foreach\_impl}\ \mathit{Qi}\ \pi i\ u\ \mathit{adjs} = \mathit{foldr}\ (\mathit{foreach\_impl\_body}\ u)\ \mathit{adjs}\ (\mathit{Qi}, \pi i)$$

We show its correctness separately:

$$\begin{array}{l}
\mathit{foreach_impl}\ \mathit{Qi}\ \pi i\ u\ (\mathit{G_adj}\ g\ u) = (\mathit{Qi}', \pi i') \wedge \mathit{prim_invar_impl}\ \mathit{Qi}\ \pi i \longrightarrow \\
\mathit{Q_invar}\ \mathit{Qi}' \wedge \mathit{M_invar}\ \pi i' \wedge \mathit{Q_}\alpha\ \mathit{Qi}' = \mathit{Qinter}\ (\mathit{Q_}\alpha\ \mathit{Qi})\ (\mathit{M_lookup}\ \pi i)\ u \wedge \\
\mathit{M_lookup}\ \pi i' = \pi'\ (\mathit{Q_}\alpha\ \mathit{Qi})\ (\mathit{M_lookup}\ \pi i)\ u
\end{array}$$

This is proved by first showing that the abstract foreach loop *foreach* can simulate the concrete one, and then using the already proved correctness of the abstract loop.

Finally, we show correctness of the whole algorithm, i.e., that the returned predecessor map satisfies its invariant, and encodes a minimum spanning tree of the reachable part of the graph:

$$\begin{array}{l}
\mathit{invar_MST}\ \mathit{prim_impl} \wedge \\
\mathit{is_MST}\ (\mathit{G_}\alpha w\ g)\ (\mathit{component_of}\ (\mathit{G_}\alpha g\ g)\ r) \\
\ (\mathit{graph}\ \{r\}\ \{(u, v) \mid \mathit{M_lookup}\ \mathit{prim_impl}\ u = \mathit{Some}\ v\})
\end{array}$$

The proof is straightforward, using the standard invariant proof rule for while loops:

$$\begin{array}{l}
[\![P\ s; \bigwedge s.\ P\ s \wedge b\ s \longrightarrow P\ (c\ s); \bigwedge s.\ P\ s \wedge \neg\ b\ s \longrightarrow Q\ s; \mathit{wf}\ r; \\
\ \bigwedge s.\ P\ s \wedge b\ s \longrightarrow (c\ s, s) \in r]\!] \\
\Longrightarrow Q\ (\mathit{while}\ b\ c\ s)
\end{array}$$

5.6 Executable Code

Using Isabelle's locale mechanism, it is straightforward to instantiate the algorithm *prim_impl* to actual data structures implementing the interfaces. We do so by using red-black trees for both, the priority map and predecessor map. The graph is implemented by red-black trees, mapping nodes to their adjacency lists.

Finally, we combine the list parser *from_list* with *prim_impl*.

$$\begin{array}{l}
\mathit{prim_list_impl}\ l\ r = \\
(\mathit{if}\ \mathit{G_valid_wgraph_repr}\ l\ \mathit{then}\ \mathit{Some}\ (\mathit{prim_impl}\ (\mathit{G_from_list}\ l)\ r)\ \mathit{else}\ \mathit{None})
\end{array}$$

We return *None* if the input list is not valid, otherwise we return a minimum spanning tree:

$$\begin{array}{l}
\mathit{case}\ \mathit{prim_list_impl}\ l\ r\ \mathit{of}\ \mathit{None} \Rightarrow \neg\ \mathit{G_valid_wgraph_repr}\ l \\
\mid \mathit{Some}\ \pi i \Rightarrow \\
\quad \mathit{let}\ g = \alpha g\ (\mathit{from_list}\ l);\ w = \alpha w\ (\mathit{from_list}\ l);\ \mathit{rg} = \mathit{component_of}\ g\ r; \\
\qquad t = \mathit{graph}\ \{r\}\ \{(u, v) \mid \mathit{lookup}\ \pi i\ u = \mathit{Some}\ v\} \\
\quad \mathit{in}\ \mathit{G_valid_wgraph_repr}\ l \wedge \mathit{invar}\ \pi i \wedge \mathit{is_MST}\ w\ \mathit{rg}\ t
\end{array}$$

Isabelle's code generator [11] can generate a functional program in various different target languages (SML, OCaml, Haskell, Scala) from *prim_list_impl*.

```
prim_impl = (let
  — Initialization
  (Q,π) = (Q_update r 0 Q_empty, M_empty);
  — Outer loop: Iterate until Q is empty
  (Q, π) =
  while (λ(Q, π). ¬ Q_is_empty Q) (λ(Q, π). let
    (u, _) = Q_getmin Q;
    — Inner loop: Update for adjacent nodes
    (Q, π) =
    foldr ((λ(v, d) (Q, π). let
        qv = Q_lookup Q v;
        πv = M_lookup π v
      in
        if v≠r ∧ (qv≠None ∨ πv=None) ∧ enat d < enat_of_option qv
        then (Q_update v d Q, M_update v u π) else (Q, π))
    ) (G_adj g u) (Q, π);
    Q = Q_delete u Q
    in (Q, π)) (Q, π)
  in π)
```

Figure 3 Implementation of Prim's algorithm, parameterized over a graph, map, and priority map interface.

5.7 Discussion and Related Work

We have used a stepwise refinement approach, from an abstract generic MST algorithm, over Prim's algorithm, to its implementation with a priority queue, and finally the realization of the priority queue with a concrete data structure and the extraction of executable code.

The abstract versions of the algorithm are inherently nondeterministic, which prevents their straightforward formalization in Isabelle/HOL. Instead, we manually came up with verification conditions (invariant maintenance) for the abstract level, and refined them towards the concrete level until we could use them to prove correct the concrete implementation. We expect that our approach of manual verification condition generation against informal algorithm sketches will not scale to more complex algorithms. To this end, the Isabelle Refinement Framework [18] provides a more scalable, though less lightweight, approach to stepwise refinement in Isabelle/HOL.

We are aware of two previous formal verifications of Prim's algorithm, but both of them ignore our focus, efficient data structures, and stop short of executable code. Abrial *et al.* [1] perform a stepwise refinement using the B event-based method. Guttmann [10] uses Isabelle/HOL to verify a version of Prim's algorithm in an extension of relation algebra.

6 Dijkstra's Algorithm

Dijkstra's algorithm [7] is a classical algorithm to determine the shortest paths from a root node to all other nodes in a weighted directed graph. Although it solves a different problem, and works on a different type of graphs, its structure is very similar to Prim's algorithm.

In particular, like Prim's algorithm, it has a simple loop structure and can be efficiently implemented by a priority queue. This makes Dijkstra's algorithm another good example to illustrate the main points of this proof pearl: Functional implementations of algorithms that use priority queues with a decrease-key operation.

A directed graph is represented by a weight function $w::'v \times 'v \Rightarrow enat$. The edge relation is $edges = \{(u, v) \mid w\ (u, v) \neq \infty\}$.

Note that this formalization differs from our formalization of undirected graphs, in that we do not model an explicit node set, nor do we encode finiteness into the type. The modeling of an explicit node set has proved useful when formalizing the concept of trees[4], which is not required for Dijkstra's algorithm. As finiteness is the only additional property that we require, we traded the overhead of defining a new type for the overhead of maintaining finiteness as an explicit assumption.

6.1 Abstract Algorithm

Again, our formalization of Dijkstra's algorithm follows the presentation of Cormen et al. [6]. However, for the sake of simplicity, our algorithm does not compute actual shortest paths, but only their weights.

For the rest of this section, we fix a weighted directed graph w and a source node s. We define $\delta\ u\ v$ to be the minimum distance from node u to node v.

Abstractly, Dijkstra's algorithm keeps track of a set of finished nodes $S::'v\ set$, and an estimate of the shortest path weights $D::'v \Rightarrow enat$. The invariant $D\_invar\ D\ S$ states that

1. $D\ u$ is an upper bound of the minimum distance between s and u:

   $$\delta\ s\ u \leq D\ u$$

2. $D\ u$ is precise if u is finished:

   $$u \in S \longrightarrow D\ u = \delta\ s\ u$$

3. $D\ u$ is consistent with the distances induced by paths that end with an edge from a node $v{\in}S$:

   $$v \in S \longrightarrow D\ u \leq \delta\ s\ v + w\ (v, u)$$

4. The start node is finished, unless in the initial state

   $$s \in S \vee D = (\lambda\_.\ \infty)(s := 0) \wedge S = \emptyset$$

The main idea of the algorithm is that the least estimate $D\ u$ among all unfinished nodes $u{\notin}S$ is already precise:

$$u \notin S \wedge (\forall v.\ v \notin S \longrightarrow D\ u \leq D\ v) \longrightarrow D\ u = \delta\ s\ u$$

Thus, adding an unfinished node $u{\notin}S$ with minimal $D\ u$ to the finished set S, and updating the estimates of all successor nodes to accommodate for paths over u, will preserve the invariant. We iterate this until all nodes are finished, and thus all estimates are precise.

[4] For example, connecting two trees on disjoint nodes by a single edge yields a tree again. Without an explicit node set, the formulation of this lemma requires tedious special cases for singleton trees.

6.2 Refined Algorithm

Like for Prim's algorithm, a priority map $Q::'v \Rightarrow nat\ option$ from unfinished nodes to estimates is used to efficiently obtain a node with minimum estimate. Moreover, we use another map $V::'v \Rightarrow nat\ option$ to map finished nodes to their minimum distances from the source. The relation between a refined state (Q,V) and an abstract state (D,S) is defined as:

$$\begin{array}{l} coupling\ Q\ V\ D\ S = \\ (D = enat\_of\_option \circ V\ ++\ Q \wedge S = dom\ V \wedge dom\ V \cap dom\ Q = \emptyset) \end{array}$$

where $(++)$ joins two maps, and *enat_of_option* maps *None* to ∞. The refined loop invariant states that the refined state is related to an abstract state that satisfies its invariant:

$$D\_invar'\ Q\ V = (\exists D\ S.\ coupling\ Q\ V\ D\ S \wedge D\_invar\ D\ S)$$

The rest of the formalization proceeds analogously to the formalization of Prim's algorithm: We implement the update of the successor nodes by iteration over a successor list, refine the algorithm to use the interfaces of directed graphs, priority maps and maps, and finally instantiate it with concrete, red-black-tree based implementations. Combining the implementation with a *from_list* function for directed graphs, we get the executable function $dijkstra\_list::('v \times 'v \times nat)\ list \Rightarrow 'v \Rightarrow ('v \times nat, color)\ tree\ option$ and the theorem

$$\begin{array}{l} \textsf{case}\ dijkstra\_list\ l\ s\ \textsf{of}\ None \Rightarrow \neg\ valid\_graph\_rep\ l \\ |\ Some\ D \Rightarrow \\ \quad valid\_graph\_rep\ l \wedge invar\ D \wedge \\ \quad (\forall u\ d.\ (lookup\ D\ u = Some\ d) = (\delta\ (wgraph\_of\_list\ l)\ s\ u = enat\ d)) \end{array}$$

Note that the distance δ is also parameterized over the graph here.

6.3 Related Work

Dijkstra's algorithm seems to be a standard benchmark for formal verification tools. One of the authors [23] has already verified Dijkstra's algorithm (including computation of shortest paths) using a functional priority queue based on Finger Trees [13], and later [15] amended the formalization to use an imperative heap data structure. Filliâtre [8] provides a verification in Why3 [9], Böhme *et al.* [3] provide one in Boogie, and Charguéraud [4] uses characteristic formulae to verify a Caml implementation. However, all of these do not verify priority queue data structures and only compute the distances, instead of actual shortest paths.

The treatment of priority queues differs in the above formalizations. Nordhoff and Lammich [23] use finger trees which support decrease-key. Charguéraud [4], however, writes:

> This implementation uses a priority queue that does not support the decrease-key operation. Using such a queue makes the proofs slightly more involved, because the invariants need to account for the fact that the queue may contain superseded values.

It appears that he uses the following trick that works for Dijkstra and Prim. Instead of decreasing the priority of some key k to p one adds the new pair (k,p) to the priority queue. If one also maintains a set of keys that have been extracted from the priority queue (which Dijkstra and Prim do anyway), one can simply ignore pairs (k,p) returned by *getmin*, if k has been extracted before. This trick requires that the same key is not inserted again after it has been extracted.

7 Conclusion

We presented priority search trees, a simple, purely functional and efficient data structure that combines search trees and priority queues, including an operation for modifying the priority associated with a key (aka "decrease-key"). We are only aware of considerably more complicated purely functional data structures with decrease-key, and the only one that has been formally verified is based on finger trees [13, 27, 22].

Based on priority search trees we gave the first verified executable implementation of Prim's algorithm. In particular we included the level of efficient data structures which had been ignored before. Therefore we show the details of the stepwise refinement and verification. We have also verified Dijkstra's algorithm in the same manner, but because of the ubiquity of this algorithm as a verification benchmark we merely sketched the proof.

References

1 Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. Formal Derivation of Spanning Trees Algorithms. In Didier Bert, Jonathan P. Bowen, Steve King, and Marina Waldén, editors, *ZB 2003: Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings*, volume 2651 of *Lecture Notes in Computer Science*, pages 457–476. Springer, 2003. doi:10.1007/3-540-44880-2_27.

2 Stefan Berghofer and Tobias Nipkow. Executing Higher Order Logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs (TYPES 2000)*, volume 2277 of *LNCS*, pages 24–40. Springer, 2002.

3 Sascha Böhme, K. Rustan M. Leino, and Burkhart Wolff. HOL-Boogie — An Interactive Prover for the Boogie Program-Verifier. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 150–166, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

4 Arthur Charguéraud. Characteristic Formulae for the Verification of Imperative Programs. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 418–430, New York, NY, USA, 2011. ACM. doi:10.1145/2034773.2034828.

5 Ching-Tsun Chou. A Formal Theory of Undirected Graphs in Higher-Order Logic. In Thomas F. Melham and Juanito Camilleri, editors, *Higher Order Logic Theorem Proving and Its Applications, 7th International Workshop, Valletta, Malta, September 19-22, 1994, Proceedings*, volume 859 of *Lecture Notes in Computer Science*, pages 144–157. Springer, 1994. doi:10.1007/3-540-58450-1_40.

6 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2009.

7 E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959. doi:10.1007/BF01386390.

8 Jean-Christophe Filliâtre. Dijkstra's shortest path algorithm. From the Toccata gallery, http://toccata.lri.fr/gallery/dijkstra.en.html.

9 Jean-Christophe Filliâtre and Andrei Paskevich. Why3 – Where Programs Meet Provers. In *ESOP*, volume 7792. Springer, March 2013. URL: https://hal.inria.fr/hal-00789533.

10 Walter Guttmann. Relation-Algebraic Verification of Prim's Minimum Spanning Tree Algorithm. In Augusto Sampaio and Farn Wang, editors, *Theoretical Aspects of Computing - ICTAC 2016 - 13th International Colloquium, Taipei, Taiwan, ROC, October 24-31, 2016, Proceedings*, volume 9965 of *Lecture Notes in Computer Science*, pages 51–68, 2016. doi:10.1007/978-3-319-46750-4_4.

11 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.

12 Ralf Hinze. A Simple Implementation Technique for Priority Search Queues. In Benjamin C. Pierce, editor, *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001.*, pages 110–121. ACM, 2001. doi:10.1145/507635.507650.

13 Ralf Hinze and Ross Paterson. Finger Trees: A Simple General-purpose Data Structure. *Journal of Functional Programming*, 16(2):197–217, 2006.

14 http://isabelle.in.tum.de/library/HOL/HOL-Data_Structures/RBT.html.

15 Peter Lammich. Refinement based verification of imperative data structures. In Jeremy Avigad and Adam Chlipala, editors, *CPP 2016*, pages 27–36. ACM, 2016.

16 Peter Lammich and Tobias Nipkow. Priority Search Trees. *Archive of Formal Proofs*, 2019. Formal proof development. URL: http://isa-afp.org/entries/Priority_Search_Trees.html.

17 Peter Lammich and Tobias Nipkow. Purely Functional, Simple, and Efficient Implementation of Prim and Dijkstra. *Archive of Formal Proofs*, 2019. , Formal proof development. URL: http://isa-afp.org/entries/Prim_Dijkstra_Simple.html.

18 Peter Lammich and Thomas Tuerk. Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm. In *Proc. of ITP*, volume 7406 of *LNCS*, pages 166–182. Springer, 2012.

19 Edward M. McCreight. Priority Search Trees. *SIAM J. Comput.*, 14(2):257–276, 1985. doi:10.1137/0214021.

20 Tobias Nipkow and Gerwin Klein. *Concrete Semantics with Isabelle/HOL*. Springer, 2014. URL: http://concrete-semantics.org.

21 Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

22 Benedikt Nordhoff, Stefan Körner, and Peter Lammich. Finger Trees. In Gerwin Klein, Tobias Nipkow, and Lawrence Paulson, editors, *Archive of Formal Proofs*. http://afp.sf.net/entries/Tree-Automata.shtml, October 2010. Formal proof development.

23 Benedikt Nordhoff and Peter Lammich. Dijkstra's Shortest Path Algorithm. *Archive of Formal Proofs*, January 2012. , Formal proof development. URL: http://isa-afp.org/entries/Dijkstra_Shortest_Path.html.

24 Lars Noschinski. A Graph Library for Isabelle. *Mathematics in Computer Science*, 9(1):23–39, 2015. doi:10.1007/s11786-014-0183-z.

25 Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

26 R. C. Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, November 1957. doi:10.1002/j.1538-7305.1957.tb01515.x.

27 Matthieu Sozeau. Program-ing Finger Trees in Coq. *SIGPLAN Not.*, 42(9):13–24, October 2007. doi:10.1145/1291220.1291156.

A Verified LL(1) Parser Generator

Sam Lasser
Tufts University, Medford, MA, USA
samuel.lasser@tufts.edu

Chris Casinghino
Draper, Cambridge, MA, USA
ccasinghino@draper.com

Kathleen Fisher
Tufts University, Medford, MA, USA
kfisher@cs.tufts.edu

Cody Roux
Draper, Cambridge, MA, USA
croux@draper.com

Abstract

An LL(1) parser is a recursive descent algorithm that uses a single token of lookahead to build a grammatical derivation for an input sequence. We present an LL(1) parser generator that, when applied to grammar $\mathcal{G}$, produces an LL(1) parser for $\mathcal{G}$ if such a parser exists. We use the Coq Proof Assistant to verify that the generator and the parsers that it produces are sound and complete, and that they terminate on all inputs without using fuel parameters. As a case study, we extract the tool's source code and use it to generate a JSON parser. The generated parser runs in linear time; it is two to four times slower than an unverified parser for the same grammar.

2012 ACM Subject Classification Theory of computation → Grammars and context-free languages; Software and its engineering → Parsers; Software and its engineering → Formal software verification

Keywords and phrases interactive theorem proving, top-down parsing

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.24

Supplement Material `https://github.com/slasser/vermillion`

Funding *Sam Lasser*: Draper Fellowship

Acknowledgements We thank our anonymous reviewers for their helpful feedback.

1 Introduction

Many software systems employ parsing techniques to map sequential input to structured output. Often, a parser is the system component that consumes data from an untrusted source—for example, many applications parse input in a standard format such as XML or JSON as the first step in a data-processing pipeline. Because parsers mediate between the outside world and application internals, they are good targets for formal verification; parsers that come with strong correctness guarantees are likely to increase the overall security of applications that rely on them.

Several recent high-profile software vulnerabilities demonstrate the consequences of using unsafe parsing tools. Attackers exploited a faulty parser in a web application framework, obtaining the sensitive data of as many as 143 million consumers [5, 14]. An HTML parser vulnerability led to private user data being leaked from several popular online services [6]. And a flaw in an XML parser enabled remote code execution on a network security device—a flaw that received a Common Vulnerability Score System (CVSS) score of 10/10 due to its severity [13]. These and other examples highlight the need for secure parsing technologies.

Parsing is a widely studied topic, and it encompasses a range of techniques with different advantages and drawbacks [7]. One family of parsing algorithms, the top-down or LL-style algorithms, shares several strengths relative to other strategies. LL parsers typically produce clear error messages, and they can easily be extended with semantic actions that produce user-defined data structures; in addition, generated LL parser code is often human-readable and similar to hand-written code [15].

The common ancestor of the LL family is LL(1), a recursive descent algorithm that avoids backtracking by looking ahead at a single input token when it reaches decision points. Its descendants, including LL(k), LL(*), and ALL(*), share an algorithmic skeleton. Each of these approaches comes with different tradeoffs with respect to expressiveness vs. efficiency. For example, LL(1) operates on a restricted class of grammars and offers linear-time execution, while ALL(*) accepts a larger class of grammars and runs in $O(n^4)$ time [16]. Different algorithms are therefore suited to different applications; it is often advantageous to choose the most efficient algorithm compatible with the language being parsed.

In this paper, we present Vermillion, a formally verified LL(1) parser generator. This tool is part of a planned suite of verified LL-style parsing technologies that are suitable for a wide range of data formats. We implemented and verified the parser generator using the Coq Proof Assistant [19], a popular interactive theorem prover. The tool has two main components. The first is a *parse table generator* that, when applied to a context-free grammar, produces an LL(1) parse table—an encoding of the grammar's lookahead properties—if such a table exists for the grammar. The second component is an LL(1) algorithm implementation that is parameterized by a parse table. By converting a grammar to a table and then partially applying the parser to the table, the user obtains a parser that is specialized to the original grammar. The paper's main contributions are as follows:

1. **End-to-End Correctness Proofs** – We prove that both the parse table generator and the parser are sound and complete. The generator produces a correct LL(1) parse table for any grammar if such a table exists. The parser produces a semantic value for its input that is correct with respect to the grammar used to generate the parser. Although prior work has verified some of the steps involved in LL(1) parse table generation [2], to the best of our knowledge, our LL(1) parse table generator and parser are the first formally verified versions of these algorithms.
2. **Total Algorithm Implementations** – We prove that the parse table generator and parser terminate on both valid and invalid inputs without the use of fuel-like parameters. To the best of our knowledge, we are the first to prove this property about a parser generator based on the context-free grammar formalism. Some existing verified parsers are only guaranteed to terminate on valid inputs; others ensure termination by means of a fuel parameter, which can produce "out of fuel" return values that do not clearly indicate success or failure. A guarantee of termination on all inputs is useful for ruling out denial-of-service attacks against the parser.
3. **Efficient Extractable Code** – We used Coq's Extraction mechanism [10] to convert Vermillion to OCaml source code and generated a parser for a JSON grammar. We then used Menhir [17], a popular OCaml parser generator, to produce an unverified parser for the same grammar and compared the two parsers' performance on a JSON data set. The verified parser was two to four times slower than the unverified and optimized one, which is similar to the reported results for other certified parsers [8, 9]. Our implementation empirically lives up to the LL(1) algorithm's theoretical linear-time guarantees.

Along the way, we deal with several interesting verification challenges. The parse table generator performs dataflow analyses with non-obvious termination metrics over context-free grammars. To implement and verify these analyses, we make ample use of Coq's tools for

defining recursive functions with well-founded measures, and we prove a large collection of domain-neutral lemmas about finite sets and maps that may be useful in other developments. The parser also uses well-founded recursion on a non-syntactic measure, and our initial implementation must perform an expensive runtime computation to terminate provably; in the final version, we make judicious use of dependent types to avoid this penalty while still proving termination. Our parser completeness proof relies on a lemma stating that if a correct LL(1) parse table exists for some grammar, then the grammar contains no left recursion. Our proof of this lemma is quite intricate, and we were unable to find a rigorous proof of this seemingly intuitive fact in the literature.

Our formalization consists of roughly 8,000 lines of Coq definitions and proofs. The development is available at the URL listed as Supplement Material above.

This paper is organized as follows: in §2, we review background material on context-free grammars and LL(1) parsing. In §3, we describe the high-level structure of our parse table generator and its correctness proofs. In §4, we present the LL(1) parsing algorithm and its correctness properties. In §5, we present the results of evaluating our tool's performance on a JSON benchmark. We discuss related work in §6 and our plans for future work in §7.

2 Grammars and Parse Tables

2.1 Grammars

Our grammars are composed of terminal symbols drawn from a set $\mathcal{T}$ and nonterminal symbols drawn from a set $\mathcal{N}$. Throughout this work, we use the letters $\{a,\ b,\ c\}$ as terminal names, $\{X,\ Y,\ Z\}$ as nonterminal names, $\{s,\ s',\ \ldots\}$ as names for arbitrary symbols (terminals or nonterminals), and $\{\alpha,\ \beta,\ \gamma\}$ as names for sentential forms (finite sequences of symbols).

A grammar consists of a start symbol $S \in \mathcal{N}$ and a finite sequence of productions $\mathcal{P}$ (described in detail below). In addition, we require the grammar writer to provide a mapping from each grammar symbol s to a type $[\![s]\!]$ in the host language (i.e., a Coq type). We borrow this mapping from a certified LR(1) parser development [8]; it enables us to specify the behavior of a parser that maps a valid input to a *semantic value* with a user-defined type, rather than simply recognizing the input as valid or building a generic parse tree for it. The symbols-to-types mapping supports the construction of flexible semantic values as follows:

- The parser consumes a list of tokens, where each token is a dependent pair $(a,\ v)$ of a terminal symbol a and a semantic value v of type $[\![a]\!]$. When the parser successfully consumes a token $(a,\ v)$, it produces the value v.
- A production $X \to \gamma\ \{f\}$ consists of a left-hand nonterminal X, a right-hand sentential form γ, and a semantic action f of type $[\![\gamma]\!] \to [\![X]\!]$. The notation $[\![\gamma]\!]$ refers to the tuple type built from the symbols in γ—for example, $[\![aY]\!] = [\![a]\!] \times [\![Y]\!]$. After the parser uses a production's right-hand side to construct a tuple of type $[\![\gamma]\!]$, it applies f to this tuple to produce a final semantic value of type $[\![X]\!]$. The user provides semantic actions at grammar definition time; these actions are dependently typed Coq functions. Throughout this work, we use the notation $X \to \gamma$ to refer to a production when its semantic action is clear from context or irrelevant to the discussion.

2.2 LL(1) Derivations

We define a derivation relation over a grammar symbol s, a word or token sequence w that s derives, and a semantic value v that s produces for w. Because it is useful for a parser to produce a semantic value for a prefix of its input sequence and return the remainder of the

sequence along with the value, the derivation relation also includes the *remainder*, or the unparsed suffix of the input. The relation has the judgment form $s \xrightarrow{v} w \mid r$, which is read, "$s$ derives w, producing v and leaving r unparsed."

The derivation relation appears in Figure 1. It is mutually inductive with an analogous relation (also in Figure 1) over a list of symbols γ, a word w, a tuple of semantic values vs, and a remainder r. This second relation has the judgment form $\gamma \overset{vs}{\Rightarrow} w \mid r$ ("γ derives w, producing vs and leaving r unparsed").

$$\textsc{DerT}\ \frac{}{a \xrightarrow{v} (a, v) \mid r} \qquad \textsc{DerNT}\ \frac{\begin{array}{c} X \to \gamma\ \{f\} \in \mathcal{P} \\ \text{peek}(w +\!\!+ r) \in \text{LOOKAHEAD}(X \to \gamma) \\ \gamma \overset{vs}{\Rightarrow} w \mid r \end{array}}{X \xrightarrow{f\ vs} w \mid r}$$

$$\textsc{DerNil}\ \frac{}{[\,] \overset{()}{\Rightarrow} \epsilon \mid r} \qquad \textsc{DerCons}\ \frac{s \xrightarrow{v} w \mid w' +\!\!+ r \qquad \gamma \overset{vs}{\Rightarrow} w' \mid r}{s :: \gamma \overset{(v,\,vs)}{\Longrightarrow} w +\!\!+ w' \mid r}$$

Figure 1 Derivation relations for symbols and lists of symbols.

The DerNT rule is the only LL(1)-specific rule in the relation. The peek function returns a value $l \in \mathcal{T} \cup \{\text{EOF}\}$ that is either the first token of the input sequence $w +\!\!+ r$, or EOF if the entire sequence is empty. The rule itself states that production $X \to \gamma\ \{f\}$ applies when $\text{peek}(w +\!\!+ r)$ and $X \to \gamma$ are in the LOOKAHEAD relation (Figure 5)—i.e., when the first input token "predicts" that production. To make this lookahead concept precise, we introduce the definitions of several predicates that are commonly used in parsing theory to relate a grammar's structure to its semantics.

2.3 NULLABLE, FIRST, and FOLLOW

A nullable grammar symbol is a symbol that can derive the empty word ϵ. The NULLABLE relation (Figure 2) captures the syntactic pattern that makes a symbol nullable. A nonterminal is nullable if it appears on the left-hand side of a production and every symbol on the right-hand side is also nullable (note that an empty right-hand side makes the left-hand nonterminal trivially nullable). A sentential form γ is nullable if it consists entirely of nullable symbols. We overload our notation for nullable symbols, writing $\text{NULLABLE}(\gamma)$ to represent the fact that γ is a nullable symbol sequence.

$$\textsc{NuSym}\ \frac{X \to \gamma\ \{f\} \in \mathcal{P} \qquad \text{NULLABLE}(\gamma)}{\text{NULLABLE}(X)} \qquad \textsc{NuGamma}\ \frac{\forall i \in \{1 \ldots n\},\ \text{NULLABLE}(s_i)}{\text{NULLABLE}(s_1 \ldots s_n)}$$

Figure 2 NULLABLE relation.

The FIRST relation (Figure 3) for a symbol s describes the set of terminals that can begin a word derived from s. If s derives a word beginning with terminal a, then $a \in \text{FIRST}(s)$. Once again, we extend this concept to sentential forms, writing $a \in \text{FIRST}(\gamma)$ if γ derives a word that begins with a.

$$\textsc{FirstT}\ \frac{}{a \in \text{FIRST}(a)} \qquad \textsc{FirstNT}\ \frac{X \to \gamma\ \{f\} \in \mathcal{P} \qquad a \in \text{FIRST}(\gamma)}{a \in \text{FIRST}(X)}$$

$$\textsc{FirstGamma}\ \frac{\text{NULLABLE}(\alpha) \qquad a \in \text{FIRST}(s)}{a \in \text{FIRST}(\alpha s \beta)}$$

Figure 3 FIRST relation.

The FOLLOW relation (Figure 4) for a symbol s describes the set of terminals that can appear immediately after a word derived from s. There is a standard practice among parser implementers of placing the EOF symbol in FOLLOW(S), where S is the start symbol, so that the parser can consume the entire input sequence. We follow this practice by adding the FollowStart rule to the relation.

$$\textsc{FollowStart}\ \frac{S \text{ is the start symbol}}{\text{EOF} \in \text{FOLLOW}(S)} \qquad \textsc{FollowRight}\ \frac{X \to \alpha Y \beta\ \{f\} \in \mathcal{P} \qquad a \in \text{FIRST}(\beta)}{a \in \text{FOLLOW}(Y)}$$

$$\textsc{FollowLeft}\ \frac{X \to \alpha Y \beta\ \{f\} \in \mathcal{P} \qquad \text{NULLABLE}(\beta) \qquad l \in \text{FOLLOW}(X)}{l \in \text{FOLLOW}(Y)}$$

Figure 4 FOLLOW relation.

With these definitions in hand, we can give a precise definition for the judgment form $l \in \text{LOOKAHEAD}(X \to \gamma)$ ("l is a lookahead token for production $X \to \gamma$") in Figure 5. Intuitively, l is a token that, when it begins a sequence ts, "predicts" that the production can derive a prefix of ts. As a special case, if the production derives $ts = \epsilon$, then EOF $\in$ LOOKAHEAD($X \to \gamma$). When an LL(1) parser builds a derivation from nonterminal X for a prefix of ts, it "looks ahead" at ts and applies a production $X \to \gamma$ such that peek(ts) $\in$ LOOKAHEAD($X \to \gamma$).

$$\textsc{FirstLk}\ \frac{l \in \text{FIRST}(\gamma)}{l \in \text{LOOKAHEAD}(X \to \gamma)} \qquad \textsc{FollowLk}\ \frac{\text{NULLABLE}(\gamma) \qquad l \in \text{FOLLOW}(X)}{l \in \text{LOOKAHEAD}(X \to \gamma)}$$

Figure 5 LOOKAHEAD relation.

2.4 Parse Tables

An LL(1) parse table is a data structure that encodes a grammar's lookahead information. An LL(1) parser uses a parse table as an oracle; it consults the table to choose which productions to apply as it builds a derivation for a token sequence.

A parse table's rows are labeled with nonterminals and its columns are labeled with lookahead symbols. Its cells contain production right-hand sides. A cell at row X and column l that contains γ, written $(X, l) \mapsto \gamma$, represents the fact $l \in \text{LOOKAHEAD}(X \to \gamma)$.

Figure 6 contains a grammar and its LL(1) parse table. Cell (**X**, **b**), for instance, contains Zc (the right-hand side of production 2) because of the fact $b \in \text{FIRST}(Zc)$. Cell (**Z**, **c**) contains Y (the right-hand side of production 5) because of the facts $\text{NULLABLE}(Y)$ and $c \in \text{FOLLOW}(Z)$.

(X is the start symbol)

1. $X \to aY$
2. $X \to Zc$
3. $Y \to \epsilon$
4. $Z \to b$
5. $Z \to Y$

	a	**b**	**c**	**EOF**
X	aY	Zc	Zc	
Y			ϵ	ϵ
Z		b	Y	

Figure 6 Example grammar and its LL(1) parse table.

A correct LL(1) parse table for grammar $\mathcal{G}$ contains all and only the lookahead facts about $\mathcal{G}$—i.e., $(X, l) \mapsto \gamma \iff l \in \text{LOOKAHEAD}(X \to \gamma)$. Not every grammar has a correct LL(1) parse table. If $l \in \text{LOOKAHEAD}(X \to \gamma)$ and $l \in \text{LOOKAHEAD}(X \to \gamma')$, where $\gamma \neq \gamma'$, then no correct table exists for $\mathcal{G}$—a parser would be unable to choose whether to apply γ or γ' upon encountering nonterminal X and token l. A grammar that has a correct LL(1) parse table is called an LL(1) grammar.

3 Parse Table Generator Correctness Properties and Verification

We now describe the process of developing and verifying an LL(1) parse table generator. Our first goal is to define the Coq function `parseTableOf : grammar -> sum error_message parse_table`. (A value of type `sum A B` is either `inl A` or `inr B`.) We then wish to prove that the function is both both *sound* (every table that it produces is the correct LL(1) parse table for its input grammar) and *complete* (it produces the correct LL(1) parse table for the grammar if such a table exists).

3.1 Structure of Parse Table Generator

Many standard compiler references describe variations on an algorithm for constructing an LL(1) parse table from a grammar. The algorithm typically involves computing the grammar's NULLABLE, FIRST, and FOLLOW sets, and then constructing the table from these sets (or returning an error value if a table cell contains multiple entries, in which case no correct parse table exists for the grammar). Appel's *Modern Compiler Implementation in ML* [1], for example, contains pseudocode for performing the first of these two steps. The algorithm presents several interesting challenges from a verification standpoint:

1. It uses an "iterate until convergence" strategy to perform a dataflow analysis over the grammar. Such an algorithm is difficult to implement in a total language because it has no obvious (i.e., syntactic) termination metric.
2. NULLABLE, FIRST, and FOLLOW are all computed simultaneously, so a proof of the function's correctness must simultaneously deal with the correctness of all three sets.

It is also possible to perform the NULLABLE, FIRST, and FOLLOW dataflow analyses sequentially (in that order) because each analysis depends only on the previous ones. This sequential approach is preferable from a proof engineering perspective, because we can clearly

state the correctness criteria for each step and verify the implementation independently of the other steps. It is also preferable from a code reuse perspective, because some individual steps may be useful in the context of other developments (for example, many species of parser generators need to compute the set of nullable nonterminals). Therefore, we structure our parse table generator as a pipeline of small functions that perform the following steps:

(1) Compute the set of nullable nonterminals.
(2) For each nonterminal X, compute FIRST(X) (using NULLABLE).
(3) For each nonterminal X, compute FOLLOW(X) (using NULLABLE and FIRST).
(4) Using NULLABLE, FIRST, and FOLLOW, compute the set of parse table entries.
(5) Build a table from the set of entries, or return an error if the set contains a conflict.

Several steps involve similar reasoning and require the same proof techniques. In the next section, we examine step (1) and its correctness proof in detail to illustrate these techniques.

3.2 Implementation of NULLABLE Dataflow Analysis

The first step in the parse table generation process is to compute the set of nullable nonterminals. Our goal is to define the function `mkNullableSet : grammar -> NtSet.t` (where `NtSet.t` is the type of finite sets of nonterminals) and then prove that when this function is applied to grammar `g`, the resulting set contains all and only the nullable nonterminals from `g`. We formalize this correctness property and theorem statement in Coq as follows (`nullable_sym` is the mechanized version of the NULLABLE relation in Figure 2):

```
Definition nullable_set_correct (nu : NtSet.t) (g : grammar) :=
  forall (x : nonterminal), NtSet.In x nu <-> nullable_sym g (NT x).

Theorem mkNullableSet_correct :
  forall (g : grammar), nullable_set_correct (mkNullableSet g) g.
```

Portions of the `mkNullableSet` implementation appear in Figure 7. We represent a grammar as a record with fields `start : nonterminal` and `prods : list production`. The expression `g.(prods)` projects the `prods` field from a grammar. The auxiliary function `mkNullableSet'` takes a (possibly incomplete) NULLABLE set `nu` as an argument and performs a single pass of the NULLABLE dataflow analysis over the grammar's productions, which produces a (possibly updated) set `nu'`. If `nu` has converged—i.e., if it is a fixed point of the dataflow analysis—then it is returned. Otherwise, the algorithm performs another iteration of the analysis, using `nu'` as the starting point.

Because of this algorithm's "iterate until convergence" structure, we need to do some extra work to prove that it terminates. To accomplish this task, we use Coq's Program extension [18], which provides support for defining functions using well-founded recursion. The `Program Fixpoint` command enables the user to define a non-structurally recursive function by providing a measure—a mapping from one or more function arguments to a value in some well-founded relation $\mathcal{R}$—and then showing that the measure of recursive call arguments is less than that of the original arguments in $\mathcal{R}$.

In the case of `mkNullableSet'`, the measure (called `countNullCands` in Figure 7) is the cardinality of `nu`'s complement with respect to the universe $\mathcal{U}$ of grammar nonterminals. We then prove that if the NULLABLE set is different before and after a single iteration of the analysis, then the more recent version contains a nonterminal that was not present in the previous version, and therefore that the set's complement with respect to $\mathcal{U}$ has decreased (this fact is captured in the lemma `nullablePass_neq_candidates_lt`).

```
Lemma nullablePass_neq_candidates_lt :
  forall (ps : list production) (nu : NtSet.t),
    ~ NtSet.Equal nu (nullablePass ps nu)
    -> countNullCands ps (nullablePass ps nu) < countNullCands ps nu.

Program Fixpoint mkNullableSet' (ps : list production) (nu : NtSet.t)
       { measure (countNullCands ps nu) } : NtSet.t :=
  let nu' := nullablePass ps nu in
  if NtSet.eq_dec nu nu' then nu else mkNullableSet' ps nu'.
Next Obligation.
  apply nullablePass_neq_candidates_lt; auto.
Defined.

Definition mkNullableSet (g : grammar) : NtSet.t :=
  mkNullableSet' g.(prods) NtSet.empty.
```

Figure 7 Selected portions of the `mkNullableSet` implementation.

Now that we have a suitable definition of `mkNullableSet` and a proof that it terminates, we turn to the proofs of its main correctness properties.

3.3 Soundness of NULLABLE Analysis

One property of `mkNullableSet` that we wish to verify is that the function is *sound*—i.e., every nonterminal in the set that it returns really is nullable in `g`:

```
Definition nullable_set_sound (nu : nullable_set) (g : grammar) :=
  forall (x : nonterminal), NtSet.In x nu -> nullable_sym g (NT x).

Theorem mkNullableSet_sound :
  forall (g : grammar), nullable_set_sound (mkNullableSet g) g.
```

The soundness proof's structure arises from the intuition that soundness holds not only of `mkNullableSet`'s final return value, but of the intermediate sets that the function computes along the way—in other words, soundness is an invariant of the function. We prove this invariant with the following two lemmas:

(1) The initial set passed to `mkNullableSet'` is sound

(2) If `nu` is sound, then `mkNullableSet'` applied to `nu` is also sound

(1) is simple to prove, because the initial `nu` argument passed to `mkNullableSet'` is the empty set, which is trivially sound. Our earlier reasoning about the termination properties of `mkNullableSet'` pays dividends in the proof of (2), because we can proceed by well-founded induction on the function's measure. The main lemma involved in this proof states that a single iteration of the dataflow analysis (called `nullablePass` in Figure 7) preserves soundness of the NULLABLE set.

3.4 Completeness of NULLABLE Analysis

In addition to being sound, `mkNullableSet` should be complete—that is, every nullable nonterminal from `g` should appear in the set that the function returns:

```
Definition nullable_set_complete (nu : NtSet.t) (g : grammar) :=
  forall (x : nonterminal), nullable_sym g (NT x) -> NtSet.In x nu.

Theorem mkNullableSet_complete :
  forall (g : grammar), nullable_set_complete (mkNullableSet g) g.
```

Once again, the proof is based on well-founded induction on the `mkNullableSet'` measure. In the interesting case, we must prove `nu` complete given the fact that `nu` and `(nullablePass g.(prods) nu)` are equal. In other words, we need to show that any fixed point of the dataflow analysis is complete. We isolate this fact in the lemma `nullablePass_equal_complete`:

```
Lemma nullablePass_equal_complete :
  forall (g : grammar) (nu : NtSet.t),
    NtSet.Equal nu (nullablePass g.(prods) nu)
    -> nullable_set_complete nu g.
```

After some simplification, we are left with this goal:

$$\frac{\texttt{nullable\_sym g x} \qquad \texttt{nu = nullablePass g.(prods) nu}}{\texttt{NtSet.In x nu}}$$

The proof proceeds by induction on the `nullable_sym` judgment. Because this relation is mutually inductive with `nullable_gamma`, we use Coq's `Scheme` command to generate a suitably powerful mutual induction principle for the two relations. Using this principle requires some extra work because the programmer must manually specify the two properties that the induction is intended to prove—one for symbols, and one for lists of symbols.

It can be difficult to come up with the right instantiations for mutual induction principles such as this one. For several of the proofs in this development, such a choice was the most difficult step. In some cases, we were able to avoid this problem by finding mutual induction-free variants of relations whose pencil-and-paper definitions seem to call for mutuality.

3.5 Correctness of Parse Table Generator

Computing the NULLABLE set is the first of several dataflow analyses involved in generating an LL(1) parse table. The correctness proofs for the remaining steps are similar in structure to the NULLABLE proofs. For example, the FIRST and FOLLOW analyses each have a soundness proof based on the fact that soundness is an invariant of the analysis, and a completeness proof based on the fact that a fixed point of the analysis must be complete.

After proving each step correct given the correctness of previous steps, we can verify `parseTableOf`—the function that implements the entire sequence—simply by chaining together the proofs for the individual steps. The `parseTableOf` soundness and completeness theorem statements appear below:

```
Theorem parseTableOf_sound :
  forall (g : grammar) (tbl : parse_table),
    parseTableOf g = inr tbl
    -> parse_table_correct tbl g.
```

```
Theorem parseTableOf_complete :
  forall (g : grammar) (tbl : parse_table),
    unique_productions g
    -> parse_table_correct tbl g
    -> exists (tbl' : parse_table),
         ParseTable.Equal tbl tbl'
         /\ parseTableOf g = inr tbl'.
```

In both theorems, the proposition `parse_table_correct tbl g` says that `tbl` contains all and only the lookahead facts about `g`. It is the mechanized notion of LL(1) parse table correctness from Section 2.4; the only difference is that in the development, we store an entire production and its semantic action in each table cell, rather than just the right-hand side.

In the completeness theorem, the `unique_productions` condition says that the grammar contains no duplicate productions. Productions are considered duplicates if they are equal up to their semantic actions—i.e., the `unique_productions` definition ignores actions. Duplicate productions always indicate user error; to understand why, consider a grammar with two productions, $X \to \gamma\ \{f\}$ and $X \to \gamma\ \{g\}$. If f and g are the same function, then the productions are redundant, and one of them can be removed without affecting the grammar's semantics. If f and g are different, then the grammar is ambiguous; the parser performs a single semantic action upon reducing a production, and it is unclear whether that action should be f or g. Coq functions cannot be compared for equality, so `parseTableOf` cannot determine whether duplicate productions are redundant or ambiguous. The `unique_productions` property is decidable, however, so the function checks its input grammar for this property and alerts the user when the check fails. The user can then correct the error in the grammar.

The completeness theorem's conclusion may seem odd; why don't we use this version?

```
Theorem unprovable_parseTableOf_complete :
  forall (g : grammar) (tbl : parse_table),
    unique_productions g
    -> parse_table_correct tbl g
    -> parseTableOf g = inr tbl.
```

In the development, a parse table is simply a finite map in which keys are row/column pairs and values are cell contents. We use FMaps, a Coq finite map library, to obtain a map representation and many useful lemmas about map operations. Two maps defined with this library that contain identical entries are not definitionally equal in Coq because they might have different internal representations. Thus, if `tbl` is a correct LL(1) parse table for `g`, we cannot prove that `parseTableOf` returns `tbl` itself—only that it returns a table `tbl'` containing exactly the same entries as `tbl`, which should be sufficient for any application.

To summarize our progress so far, we have proved that the parse table generator terminates on all inputs, and that it produces a correct LL(1) parse table for its input grammar whenever such a table exists.

4 Parser Correctness and Verification

We now turn to the task of defining and verifying the LL(1) parsing algorithm. Our first goal is to define a function `parse` that uses an LL(1) parse table `tbl` and a symbol `s` to build a semantic value for a prefix of the token sequence `ts`:

```
Definition parse (tbl : parse_table) (s : symbol) (ts : list token) :
  sum parse_failure (symbol_semty s * list token).
```

(The type `symbol_semty s` is the type of semantic values for symbol `s`.) We then wish to verify that as long as the function's LL(1) parse table argument is correct for some grammar, its return value is correct with respect to the grammar's derivation relation. Below are the three main parser correctness properties that we prove:

1. (*Soundness*) – If the parser consumes a token sequence, returning a semantic value v for prefix w and an unparsed suffix r, then $s \xrightarrow{v} w \mid r$ holds.
2. (*Error-Free Termination*) – The parser never reaches an error state when applied to a correct LL(1) parse table.
3. (*Completeness*) – If $s \xrightarrow{v} w \mid r$ holds, then the parser returns v and r when applied to symbol s and token sequence $w +\!\!+ r$.

4.1 Parser Structure

Because our parser's correctness specification is the LL(1) derivation relation, it is natural to structure the parser in a way that mirrors the relation's structure. An intuitive way of doing so is to define two mutually recursive functions, `parseSymbol` and `parseGamma`, that respectively consume a symbol and a list of symbols and return a semantic value and a tuple of semantic values. However, a naïve attempt at defining these two functions leads to a violation of Coq's syntactic guardedness condition, which requires all recursive function calls to have a structurally decreasing argument. The termination checker is not being overly conservative—a naïvely defined LL(1) parser might actually fail to terminate on certain inputs! The reason is that our parse tables are simply finite maps, and it is possible to create a map that would cause the functions to diverge. For example, consider the singleton map containing the binding $(X, a) \mapsto X$. Applying the parser to this map and a token sequence beginning with a would cause it to loop infinitely.

The problem with this table is that it includes a *left-recursive* entry—an entry that leads the parser from nonterminal X back to X without consuming any input. Our parser detects left recursion dynamically by maintaining a set of visited nonterminals that is reset to $\emptyset$ when the parser consumes a token. If the parser reaches a nonterminal that is already present in the visited set, it halts and returns an error value. In our proof of error-free termination, we show that the parser never actually returns this "left recursion detected" value as long as it is applied to a correct LL(1) parse table for some grammar, because a grammar that has such a table contains no left recursion.

Of course, left recursion is not the only failure case—the parser could also determine that no input prefix is in the language that it recognizes. In this case, it should provide some information about why it rejected the input. Therefore, our parser returns one of the following values:

- `inr (v, r)`, where `v` is a semantic value for a prefix of the input tokens and remainder `r` is the unparsed suffix, indicating a successful parse.
- `inl (Reject m r)`, where `m` is an error message and remainder `r` is the suffix that the parser was unable to consume.
- `inl (Error m x r)`, where `m` is an error message, `x` is the nonterminal found to be left-recursive, and `r` is the unparsed suffix.

After adding left recursion detection, we still have to convince Coq that `parseSymbol` and `parseGamma` terminate, because their termination metric depends on multiple function parameters. The token sequence decreases structurally in some recursive calls, while in others, the visited set grows larger (and therefore, its complement relative to the universe of grammar nonterminals grow smaller). Coq's `Function` and `Program` commands can often ease the burden of defining functions with subtle termination conditions; both commands enable the user to write a function and then provide its termination proof after the fact. Unfortunately, `Function` and `Program` do not support mutually recursive functions that are defined with a well-founded measure. Therefore, we implement well-founded recursion "by hand," mimicking the process that these commands perform automatically. The process involves the following steps:

1. Define a measure `meas` that maps arguments of `parseSymbol` and `parseGamma` to the following triple of natural numbers:
   - *(First projection)* The length of the token sequence.
   - *(Second projection)* The cardinality of the visited set's complement relative to the set of all grammar nonterminals.
   - *(Third projection)* The size of the function's "symbolic" argument, which is a symbol in the case of `parseSymbol` and a list of symbols in the case of `parseGamma`. We define the size of a symbol to be 0 and the size of a list of symbols `gamma` to be `1 + length gamma`. This choice allows `parseGamma` to call `parseSymbol` with an unchanged token sequence and visited set, and it allows `parseGamma` to call itself under the same conditions as long as `length gamma` decreases.
2. Define a lexicographic ordering `triple_lt` on triples of natural numbers.
3. Add a proof of the measure value's *accessibility* in the `triple_lt` relation (i.e., a proof that there are no infinite descending chains from the value in `triple_lt`) as an extra function argument.
4. Prove lemmas showing that the size of this accessibility proof decreases on recursive calls.
5. Prove that `triple_lt` is well-founded so that the parser can be called with any initial set of arguments.

This process yields functions with the following signatures:

```
Fixpoint parseSymbol  (tbl : parse_table) (s : symbol)
                      (ts : list token) (vis : NtSet.t)
                      (a : Acc triple_lt (meas tbl ts vis (Sym_arg s)))
  : sum parse_failure
        (symbol_semty s * {ts' & length_lt_eq _ ts' ts}) ...
with parseGamma (tbl : parse_table) (gamma : list symbol)
                (ts : list token) (vis : NtSet.t)
                (a : Acc triple_lt (meas tbl ts vis (Gamma_arg gamma)))
  : sum parse_failure
        (rhs_semty gamma * {ts' & length_lt_eq _ ts' ts}) ...
```

In each return type, `{ts' & length_lt_eq _ ts' ts}` is the dependent type of a token sequence `ts'` that is either shorter than the `ts` argument or definitionally equal to `ts`. By including this information in the functions' dependent return types, we avoid computing the length of the remaining token sequence at runtime, which would hamper performance.

Finally, we define `parse`, a top-level interface to the parser that invokes `parseSymbol` with an empty visited set and an appropriate accessibility proof term, and that strips out the return value's dependent component:

```
Definition parse (tbl : parse_table) (s : symbol) (ts : list token) :
  sum parse_failure (symbol_semty s * list token) :=
  match parseSymbol tbl s ts NtSet.empty (triple_lt_wf _) with
  | inl failure => inl failure
  | inr (v, existT _ ts' _) => inr (v, ts')
  end.
```

4.2 Parser Soundness

The first parser correctness property that we prove is soundness with respect to the LL(1) derivation relation. We show that whenever the parser returns a semantic value for a prefix of its input, the relation `sym_derives_prefix` (the mechanized version of the Figure 1 symbol derivation relation) produces the same value for the same prefix:

```
Theorem parse_sound :
  forall (g : grammar) (tbl : parse_table) (s : symbol)
         (w r : list token) (v : symbol_semty s),
    parse_table_correct tbl g
    -> parse tbl s (w ++ r) = inr (v, r)
    -> sym_derives_prefix g s w v r.
```

We prove this theorem via a slightly different statement that implies the previous one:

```
Lemma parseSymbol_sound :
  forall g tbl s ts vis Hacc v r Hle,
    parse_table_correct tbl g
    -> parseSymbol tbl s ts vis Hacc = inr (v, existT _ r Hle)
    -> exists w, w ++ r = ts /\ sym_derives_prefix g s w v r.
```

The main difference between these two properties is that `parse_sound` uses the append function (`++`) to specify exactly how the function divides its input sequence into a parsed prefix and an unparsed suffix. It is difficult to reason directly about this statement because there are multiple ways of dividing the input into a prefix and suffix.

The `parseSymbol_sound` proof relies on yet another lemma that generalizes over both `parseSymbol` and `parseGamma`. The proof of this latter lemma proceeds by nested induction on the lexicographic components of the functions' measure. The proof is straightforward by design; we were careful to define `parseSymbol` and `parseGamma` so that the "success" path through the functions' recursive calls mirrors the structure of the derivation relation.

4.3 Parser Error-Free Termination

Our next task is to prove that the parser never returns an error value as long as its table argument is a correct LL(1) parse table for some grammar:

```
Theorem parse_terminates_without_error :
  forall (g : grammar) (tbl : parse_table)
         (s : symbol) (ts ts' : list token)
         (m : string) (x : nonterminal),
    parse_table_correct tbl g
    -> ~ parse tbl s ts = inl (Error m x ts').
```

However, it is certainly possible for `parseSymbol` and `parseGamma` to return an error value! For example, they will produce an error when applied to nonterminal X and a visited set that already contains X. To prove the top-level function `parse` safe, we need to specify the conditions that cause the underlying functions to produce an error, and then prove that these conditions do not apply to the top-level call.

One error condition is when the parser is applied to symbol s and its visited set already contains a nonterminal that is reachable from s without any input being consumed. We formalize this notion of "null-reachability" in the inductive predicate `nullable_path`:

```
Inductive nullable_path (g : grammar) (la : lookahead) :
  symbol -> symbol -> Prop :=
| DirectPath : forall x z gamma f pre suf,
    In (existT _ (x, gamma) f) g.(prods)
    -> gamma = pre ++ NT z :: suf
    -> nullable_gamma g pre
    -> lookahead_for la x gamma g
    -> nullable_path g la (NT x) (NT z)
| IndirectPath : forall x y z gamma f pre suf,
    In (existT _ (x, gamma) f) g.(prods)
    -> gamma = pre ++ NT y :: suf
    -> nullable_gamma g pre
    -> lookahead_for la x gamma g
    -> nullable_path g la (NT y) (NT z)
    -> nullable_path g la (NT x) (NT z).
```

When this predicate holds of two symbols s and s', there exists a sequence of steps through the grammar from s to s' in which all symbols visited along the way are nullable.

The second error condition is when the grammar contains a left-recursive nonterminal, which is just a special case of null-reachability:

```
(* symbol s is left-recursive in grammar g on lookahead token la *)
Definition left_recursive (g : grammar) (s : symbol) (la : lookahead) :=
  nullable_path g la s s.
```

We prove a lemma stating that when `parseSymbol` or `parseGamma` returns an error value, one or both of these error conditions holds. The first condition does not apply to `parse` because the top-level function calls `parseSymbol` with an empty visited set. To prove that the second condition does not apply, we show that a grammar with a correct LL(1) parse table contains no left recursion. Although standard references mention this property in passing, we could not find a rigorous proof in the literature. Our proof involves a fair amount of machinery; it consists of the following steps:

(1) We define *sized* versions of the `nullable_sym` (Figure 2) and `first_sym` (Figure 3) relations. These versions include a natural number representing the proof term's size.
(2) We prove that these sizes are deterministic for an LL(1) grammar—any two proofs of the same `nullable_sym` or `first_sym` fact have the same size.
(3) We show that if grammar `g` contains a left-recursive nonterminal, then there are two proofs of the same `nullable_sym` or `first_sym` fact about `g` with *different* sizes.

These steps enable us to prove the lemma `LL1_parse_table_impl_no_left_recursion` by obtaining a contradiction from (2) and (3):

```
Lemma LL1_parse_table_impl_no_left_recursion :
  forall (g : grammar) (tbl : parse_table)
         (x : nonterminal) (la : lookahead),
    parse_table_correct tbl g
    -> ~ left_recursive g (NT x) la.
```

4.4 Parser Completeness

Finally, we prove that our parser is *complete*—if a grammar symbol derives a semantic value for a prefix of a token sequence, then the parser produces the same value for the same prefix:

```
Theorem parse_complete :
  forall (g : grammar) (tbl : parse_table)
         (s : symbol) (w r : list token)
         (v : symbol_semty s),
    parse_table_correct tbl g
    -> sym_derives_prefix g s w v r
    -> parse tbl s (w ++ r) = inr (v, r).
```

Our error-free termination result simplifies the task of proving completeness. We begin by proving a more general lemma stating that when a grammar derivation exists, the parser either returns an error or produces the semantic value from the derivation:

```
Theorem parseSymbol_error_or_complete :
  forall g tbl s w r v vis a,
    parse_table_correct tbl g
    -> sym_derives_prefix g s w v r
    -> (exists m x ts',
          parseSymbol tbl s (w ++ r) vis a = inl (Error m x ts'))
       \/ (exists Hle,
             parseSymbol tbl s (w ++ r) vis a = inr (v, existT _ r Hle)).
```

We prove this lemma by induction on the derivation relation, use the error-free termination theorem to rule out the left disjunct, and use the right disjunct to prove the completeness theorem itself.

5 Evaluation

To evaluate the efficiency of our generated parsers, we extracted Vermillion to OCaml source code and generated an LL(1) parser for the JSON data format. We also used Menhir, a popular OCaml LR(1) parser generator, to produce an unverified parser for the same grammar and compared the two parsers' performance on a JSON data set.

We based our Menhir lexer[1] and grammar on the ones described in the *Real World OCaml* textbook's tutorial on JSON parsing [12]. We then replicated the grammar in Vermillion's input format. Because our tool consumes a list of tokens, we used Menhir to generate a second parser that acts as a preprocessor for Vermillion—it simply tokenizes an entire JSON string. In our evaluation, we count this tokenizer's execution time as part of the LL(1) parser's total execution time.

[1] The lexer does not support Unicode escape sequences, but nothing prevents Vermillion or Menhir from handling Unicode tokens in principle.

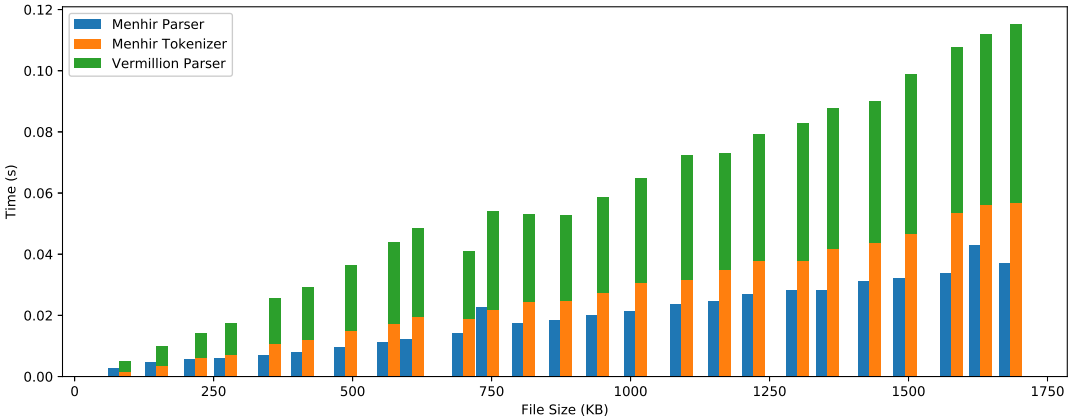


Figure 8 Average execution times of Menhir and Vermillion JSON parsers.

We ran both JSON parsers on a small data set, averaging the execution times of ten trials for each data point. The results appear in Figure 8. The Vermillion parser is between two and four times slower than the unverified Menhir parser on each data point. This comparison is not entirely scientific, because Menhir and Vermillion use two different parsing algorithms—LR(1) and LL(1), respectively. Nevertheless, it suggests that Vermillion's performance is reasonable, given that it was designed with ease of verification (rather than optimal performance) in mind. Other certified parsers obtain similar performance results; a validated LR(1) parser [8] runs about five times slower than its unvalidated counterpart, and a verified PEG interpreter [9] is two to three times slower than an unverified version.

As an interesting side note, when we first extracted Vermillion to OCaml, we discovered that its performance was superlinear! This earlier version of the parser periodically computed the length of the remaining input to determine whether a previous recursive call had consumed any tokens, and thus whether it was safe to empty the set of visited nonterminals. With some refactoring, we were able to lift this reasoning about input length into the proof component of the parser's dependent return type, ensuring that it is erased at extraction time.

6 Related Work

Barthwal and Norrish [2] use the HOL4 proof assistant to prove the soundness and completeness of generated SLR parsers. Like us, they structure their tool as a generator and a parse function parameterized by the generator's output. The parsers are not proved to terminate on invalid inputs. The work does not include performance results, but the parsers are not designed to be performant; they compute DFA states during execution rather than statically.

Jourdan et al. [8] present a validator that determines whether a generated LR(1) parser is sound and complete. *A posteriori* validation is a flexible and lightweight alternative to full verification; the validator is compatible with untrusted generators, and its formalization is small. The validator does not guarantee that a parser terminates on invalid inputs. While LR(1) parsers are compatible with a larger class of grammars than LL(1) parsers, they often produce less intuitive error messages.

Parsing Expression Grammars (PEGs) [4] are a language representation that is sometimes used in place of context-free grammars to specify parsers. Koprowski and Binsztok [9] verify the soundness and completeness of a PEG parser interpreter. They also ensure that the

interpreter terminates on both valid and invalid inputs by rejecting grammars that fail a syntactic check for left recursion. Wisnesky et al. [20] verify an optimized PEG parser using the Ynot framework. Ynot is a library for proving the partial correctness of imperative programs, so the parser is not guaranteed to terminate. One drawback of using PEG parsers is that they make greedy choices at decision points—e.g., the rule $S \to a \mid ab$ applied to string ab parses a instead of ab—which can produce difficult-to-debug behavior.

7 Conclusions

We have verified that our parser generator produces a sound and complete LL(1) parser for its input grammar whenever such a parser exists, and that the generated parsers terminate on valid and invalid inputs without using fuel. Below, we discuss two possible extensions of this work: ruling out parser errors *a priori* and generating parser source code.

Our parser includes branches that represent error states. These branches survive the extraction process and slow down the resulting code, even though we prove that the algorithm never reaches them when applied to a correct LL(1) parse table. An anonymous reviewer made a useful analogy between the parser and an interpreter that checks for type errors dynamically, even when a static type system ensures that a valid input program never triggers these errors—i.e., that "well-typed programs cannot 'go wrong'" [11]. The reviewer also noted that it might be possible to remove these branches from the parser by making it a function over correct LL(1) parse tables instead of simply-typed tables, just as one can remove dynamic type-checking from an interpreter by parameterizing it with typing derivations instead of raw terms. We chose to rule out errors *a posteriori* because it is often simpler to separate the concerns of programming and proving, but the *a priori* approach would be more elegant to some observers and certainly more efficient. We hope to explore the idea in future extensions to this work.

Our parsers represent tables as finite maps and perform map lookups at decision points, which is a likely source of inefficiency. Many production-grade parser generators produce source code that is specialized to their input grammar. These parsers represent table lookups with source-level constructs (e.g., `match` expressions) instead of data structure operations. Generated parser code is likely to be more efficient than a table-based interpreter; for example, Menhir enables the user to choose between these two representations, and an informal benchmark finds that code generation produces parsers that are two to five times faster than their table-based counterparts [17]. We could develop a version of our tool that generates abstract syntax for a language with mechanized semantics, such as Clight [3], and verify that the abstract syntax representation of a parser is extensionally equivalent to a table-based parser for the same grammar.

References

1 Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

2 Aditi Barthwal and Michael Norrish. Verified, executable parsing. In *European Symposium on Programming*, pages 160–174. Springer, 2009.

3 Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

4 Bryan Ford. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 111–122, New York, NY, USA, 2004. ACM. doi:10.1145/964001.964011.

5 Dan Goodin. Failure to patch two-month-old bug led to massive Equifax breach. https://arstechnica.com/information-technology/2017/09/massive-equifax-breach-caused-by-failure-to-patch-two-month-old-bug/, 2017.

6 cloudflare: Cloudflare Reverse Proxies are Dumping Uninitialized Memory. https://bugs.chromium.org/p/project-zero/issues/detail?id=1139, 2017.

7 Dick Grune and Ceriel JH Jacobs. *Parsing Techniques (Monographs in Computer Science)*. Springer-Verlag, 2006.

8 Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. Validating LR(1) parsers. In *European Symposium on Programming*, pages 397–416. Springer, 2012.

9 Adam Koprowski and Henri Binsztok. TRX: A formally verified parser interpreter. In *European Symposium on Programming*, pages 345–365. Springer, 2010.

10 Pierre Letouzey. Extraction in Coq: An overview. In *Conference on Computability in Europe*, pages 359–369. Springer, 2008.

11 Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

12 Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml: Functional programming for the masses*. O'Reilly Media, Inc., 2013.

13 CVE-2016-0101. National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2016-0101, 2016.

14 CVE-2017-5638. National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2017-5638, 2017.

15 Terence Parr and Kathleen Fisher. LL(*): The foundation of the ANTLR parser generator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 425–436, June 2011.

16 Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, volume 49, pages 579–598, October 2014. doi:10.1145/2714064.2660202.

17 François Pottier and Yann Régis-Gianas. Menhir reference manual. *Inria*, August 2016.

18 Matthieu Sozeau. PROGRAM-ing finger trees in Coq. In *ACM SIGPLAN International Conference on Functional Programming. Association for Computing Machinery*, 2007.

19 The Coq Proof Assistant, version 8.9.0, January 2019. doi:10.5281/zenodo.2554024.

20 Ryan Wisnesky, Gregory Michael Malecha, and John Gregory Morrisett. Certified web services in Ynot, 2010.

Binary-Compatible Verification of Filesystems with ACL2

Mihir Parang Mehta
University of Texas at Austin, USA
http://www.cs.utexas.edu/users/mihir
mihir@cs.utexas.edu

William R. Cook
University of Texas at Austin, USA
http://www.cs.utexas.edu/users/wcook
wcook@cs.utexas.edu

Abstract

Filesystems are an essential component of most computer systems. Work on the verification of filesystem functionality has been focused on constructing new filesystems in a manner which simplifies the process of verifying them against specifications. This leaves open the question of whether filesystems already in use are correct at the binary level.

This paper introduces LoFAT, a model of the FAT32 filesystem which efficiently implements a subset of the POSIX filesystem operations, and HiFAT, a more abstract model of FAT32 which is simpler to reason about. LoFAT is proved to be correct in terms of refinement of HiFAT, and made executable by enabling the state of the model to be written to and read from FAT32 disk images. EqFAT, an equivalence relation for disk images, considers whether two disk images contain the same directory tree modulo reordering of files and implementation-level details regarding cluster allocation. A suite of co-simulation tests uses EqFAT to compare the operation of existing FAT32 implementations to LoFAT and check the correctness of existing implementations of FAT32 such as the mtools suite of programs and the Linux FAT32 implementation. All models and proofs are formalized and mechanically verified in ACL2.

2012 ACM Subject Classification Theory of computation → Program verification

Keywords and phrases interactive theorem proving, filesystems

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.25

Related Version A preprint version was published at https://easychair.org/publications/preprint/dMh7.

Supplement Material The proof development described in this paper has been incorporated into the ACL2 Community books; these are part of the ACL2 distribution on GitHub (http://www.github.com/acl2/acl2). The source code for the models and proofs, with instructions for certifying the models, is available (https://github.com/acl2/acl2/tree/master/books/projects/filesystems).

Funding *Mihir Parang Mehta*: This material is based upon work supported by the National Science Foundation under Grant No. CNS-1525472.

Acknowledgements Thanks to Warren A. Hunt Jr. and Matt Kaufmann for their guidance.

1 Introduction

Filesystems offer a critical part of the functionality of modern operating systems, going beyond the basic functionality of persistent storage to offer crash consistency, concurrent data access, and distributed operation. Within the formal methods community, filesystem verification is becoming a mature discipline with the development of high-performance filesystems accompanied by proofs of increasingly expansive notions of correctness. However,

it is often necessary to verify the operation of an existing filesystem which is known to be suitable in a particular context, in terms of properties such as CPU usage, memory usage, or fragmentation behavior. This remains a challenge.

This paper shows the construction of an executable model of the `FAT32` filesystem, using the interactive theorem prover ACL2 [23], which is useful for reasoning about programs that interact with the filesystem. The aim for this effort is *binary compatibility*, i.e. byte-level correspondence between the model and existing, mature implementations of `FAT32`. This is achieved through a careful examination of the specification of `FAT32` and the behavior of its implementations. Binary compatibility enables reasoning at a low level of abstraction about the precise sequences of bytes accepted and returned by `POSIX` system calls, as well as their return values and the `errno` [24] values set by them. By building this model, LoFAT, incrementally in the refinement style, we are able to address these low-level details while adhering to a more abstract model, HiFAT, which is easier to reason about. The refinement relation between LoFAT and HiFAT is proved.

LoFAT is executable; it includes functionality to read the filesystem state from and write the filesystem state to `FAT32` disk images. The disk image is a convenient abstraction to represent the state of the filesystem, and by interacting directly with disk images the verified implementation needs to trust only a small number of ACL2 functions for writing and reading. Optimization of these procedures for faster I/O enables the efficient execution of the model and co-simulation with existing implementations of `FAT32` over various types of file operations, which helps find bugs.

We begin by providing some necessary details about the reasoning and execution properties of the ACL2 theorem proving system (Section 2). Touching on the `FAT32` filesystem's on-disk format, we proceed to introduce LoFAT and HiFAT[1] (Section 3), detailing the refinement relation between these models and the proof thereof (Section 4). We examine some performance considerations involved in making executions of LoFAT efficient enough for co-simulation tests, and describe the co-simulation tests developed (Section 5). We briefly review the related work (Section 6) and outline some plans for concurrency and crash consistency-related future extensions of this work (Section 7).

2 Background on ACL2

The ACL2 theorem proving system consists of a language, which is a pure functional subset of Common Lisp, and a prover which discharges proof obligations expressed in this language.[2] ACL2, employing an untyped first-order logic, incorporates many automated strategies for discharging first-order goals while also allowing user control of the proof process at different levels of abstraction. As in mathematics, the proof of a conjecture in ACL2 usually relies on the proof of simpler lemmas (*rules*). Most often, these lemmas are *rewrite rules* for rewriting a certain type of term under certain hypotheses; however, other types of rules exist, such as linear rules for arithmetic reasoning.

2.1 Guard verification

A function can optionally have a *guard*, an arbitrary propositional formula in terms of its arguments which is checked to be true at runtime. ACL2 generates a proof obligation stating that the guards of all functions called within the function body are satisfied when the guard

[1] These names respectively refer to Low Level of Abstraction and High Level of Abstraction.
[2] In the literature, the term ACL2 is sometimes used to refer to the language, and sometimes used to refer to the prover.

itself is satisfied. The proof of this obligation is optional; when a function is not guard-verified, guards for function calls within the body are instead checked at runtime. Guard-verified functions, however, avoid these runtime checks, and in general execute faster because guards often include constraints on the type of the function's arguments which allow space to be efficiently allocated for fixed-width integers, strings, and the like. Guard verification helps correct programming errors early and often leads to the formulation of lemmas which can be reused in later proofs.

The guard mechanism also supports `mbe` (*must be equal*) [4], an ACL2 construct which allows the user to locally decouple logical meaning and operational behavior. In a function body, a sub-expression of the form `(mbe :logic term1 :exec term2)` will be treated as meaning `term1` during reasoning but will behave as `term2` at runtime; this enables optimization by a choice of `term2` which is efficient during execution. This is sound because `mbe` extends the function's guard obligation to include the statement that `term1` and `term2` are equal in their local context when the function's guard is satisfied.

2.2 Single-threaded objects

In applicative settings, updates to data structures result in the creation of a new copy of the data structure, which can prove expensive in terms of time and memory. In ACL2, this kind of performance penalty is avoided by the use of immutable data structures called *single-threaded objects*, or stobjs [9]. Stobjs are aggregate structures with scalar and array fields, equipped with the usual applicative semantics, but restricted syntactically to ensure that only one copy of the stobj can be referenced at a given time. With just one immutable copy of the stobj, accesses and updates to scalar and array fields can be implemented in constant time.

As with all aggregate data structures, proving invariants of algorithms involving stobjs necessitates lemmas about the invariance of stobj fields while updating other fields (akin to *frame axioms* [40], although these lemmas are not axioms of the theory). ACL2 macros are used to reduce the effort required to generate these lemmas.

2.3 Equivalence and rewriting

In ACL2, binary predicates can be proved to be equivalence relations. Such an equivalence is treated like first-order equality, in that rules can be formulated to rewrite terms in the context of the given equivalence.

We use a few standard techniques for defining and establishing equivalences in ACL2's untyped logic.

- When a subset relation can be defined on objects which are to be assigned to equivalence classes, equivalent objects can be defined to be subsets of each other. Then, the proofs of reflexivity, symmetry and transitivity arise from the proofs of reflexivity, anti-symmetry and transitivity for the subset relation.
- When a transformation exists between two types, objects of the first type can be defined to be equivalent when they transform to the same object (modulo a previously defined equivalence) of the second type.
- Sometimes an equivalence relation needs to be defined on some notion of well-formed objects (such as objects which can be transformed to objects of a different type). However, guards notwithstanding, all functions in ACL2 must be total including equivalence predicates. In such a case, the predicate can be made a total function by assigning all

ill-formed objects to the same equivalence class and assigning no well-formed objects to this class. This renders the claim of reflexivity, symmetry and transitivity trivial in the ill-formed case.

2.4 Logical story of I/O

Theorem proving systems generally have interfaces with the operating system which are unverified, because operating-system activity is unpredictable and may not return a consistent result on two calls to the same function with the same arguments. This is also the case with ACL2, which provides I/O functionality at various levels of abstraction for programmer convenience. However, a *logical story of I/O* [13] is adhered to, consisting of formal specifications for these I/O functions in terms of their input/output behavior and errors passed on from the operating system. These formal specifications exist in the ACL2 logic and support proofs about sequences of I/O operations and optimizations thereof.

3 `FAT32` – specification and modeling

`FAT32` was previously the default filesystem for the Windows operating system, and continues to see widespread use in embedded systems and in removable media.

Having detailed the data organization of a `FAT32` disk image in our earlier work [34], we limit ourselves here to a brief summary of the on-disk data structures, i.e. the *reserved area*, the *file allocation table*, and the *data region*. Unless otherwise specified, we refer to both regular files (which contain sequences of bytes) and directory files (which contain sequences of directory entries pointing to other files with names, access times and other metadata for each) as *files*.

- The contents of all files are split into fixed-size *clusters* (or extents); these clusters are stored in the data region.
- Linked lists, called *clusterchains*, yield the sequences of clusters belonging to a given file; these clusterchains are stored in the file allocation table. Multiple copies of the file allocation table are allowed in order to protect against data loss in the event of corruption; however only the first one is considered authoritative, and a `FAT32` implementation may update the redundant copies infrequently (or not at all).
- The reserved area is a collection of scalar and array fields which specify such volume-wide metadata for the filesystem as the location of the root directory, the size of a cluster, and the number of clusters.

Microsoft provides an authoritative `FAT32` specification [35], which includes a number of constraints on the various scalar and array fields, specifying such things as the maximum and minimum number of clusters, the maximum sizes of regular and directory files, and the allowable sizes of clusters. It is necessary to incorporate these constraints into our formal development in order to reason about upper bounds on the sizes of the data structures we allocate and avoid impossible corner cases while proving other useful properties.

Thus, to define our model LoFAT, we first define a single-threaded object type recognized by the predicate `fat32-in-memoryp`. Augmenting this predicate with clauses for the various `FAT32` constraints, we obtain the predicate `lofat-fs-p` (Listing 1), which recognizes valid instances of LoFAT. These constraints are a subset of the constraints actually stipulated for `FAT32`, chosen to be as small as possible while meeting our proof needs. This helps us avoid unduly restricting the possible co-simulations we can undertake with `FAT32` implementations which may not strictly adhere to the specification.

Listing 1 `lofat-fs-p`.

```
(defun lofat-fs-p (fat32-in-memory)
  (and
   (fat32-in-memoryp fat32-in-memory)
   ;; There must be at least 512 bytes per sector.
   (>= (bpb_bytspersec fat32-in-memory) *ms-min-bytes-per-sector*)
   ;; Each cluster must contain a positive integer number of sectors.
   (>= (bpb_secperclus fat32-in-memory) 1)
   ;; There is a lower bound and an upper bound to the number of
   ;; clusters.
   (>= (count-of-clusters fat32-in-memory) *ms-min-count-of-clusters*)
   (<= (+ *ms-first-data-cluster* (count-of-clusters fat32-in-memory))
       *ms-bad-cluster*)
   ;; The reserved area must span a positive integer number of
   ;; sectors.
   (>= (bpb_rsvdseccnt fat32-in-memory) 1)
   ;; Zero or more redundant copies of the FAT are allowed.
   (>= (bpb_numfats fat32-in-memory) 1)
   ;; The FAT must span a positive integer number of sectors.
   (>= (bpb_fatsz32 fat32-in-memory) 1)
   ;; The root cluster must exist in the addressable part of the file
   ;; allocation table.
   (>= (fat32-entry-mask (bpb_rootclus fat32-in-memory))
       *ms-first-data-cluster*)
   (< (fat32-entry-mask (bpb_rootclus fat32-in-memory))
      (+ *ms-first-data-cluster* (count-of-clusters fat32-in-memory)))
   (<= (+ (count-of-clusters fat32-in-memory) *ms-first-data-cluster*)
       (fat-entry-count fat32-in-memory))
   ;; The cluster size must be a multiple of the size of a directory
   ;; entry.
   (equal (mod (cluster-size fat32-in-memory) *ms-dir-ent-length*) 0)
   (equal (mod *ms-max-dir-size* (cluster-size fat32-in-memory)) 0)
   ;; The data region must be an array of clusters of the appropriate
   ;; length.
   (stobj-cluster-listp-helper fat32-in-memory
                               (data-region-length fat32-in-memory))
   (equal (data-region-length fat32-in-memory)
          (count-of-clusters fat32-in-memory))
   ;; The file allocation table must contain the appropriate number
   ;; of 4-byte-wide entries.
   (equal (* 4 (fat-length fat32-in-memory))
          (* (bpb_fatsz32 fat32-in-memory)
             (bpb_bytspersec fat32-in-memory)))))
```

It has been argued [32] that the axiomatic verification methodology, wherein specific properties of a system are enumerated and proved, is inadequate for systems of any significant complexity, which can only be verified through refinement. Much of the related work [6, 41, 11] opts to prove the correctness of an implemented filesystem through refinement of a specification, demonstrating a *de facto* consensus on this point. Thus, we choose to develop the abstract model HiFAT, and prove that it is *refined without stuttering* [1] by LoFAT. HiFAT instances are directory trees in which the leaf nodes are regular files and the non-leaf nodes are directories. Each node in a directory tree contains the `FAT32` directory entry for the corresponding file, and the full contents of each regular file are stored as strings within the tree. Further, these trees are subject to `FAT32`'s constraints – a regular file may be up to $2^{32} - 1$ bytes long; a directory may contain up to $2^{16} - 1$ directory entries; and a directory may not contain duplicate directory entries.

As a result of this refinement relationship, we are able to reason about file operations in terms of operations on directory trees, while implementing them efficiently in a data format that is very close to the actual structure of a `FAT32` disk image.

4 Properties proved

Much of the proof effort for this work concerns the correctness of the transformations between the different `FAT32` representations used; these transformations are obliged to terminate in a bounded amount of time, be invertible in terms of appropriate equivalence relations, and return the proper error codes. These proofs lead up to the refinement proof showing the correctness of the `POSIX` system calls implemented for `FAT32` (Section 5.2).

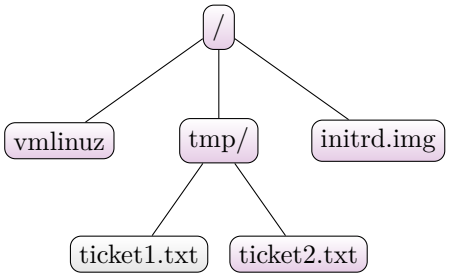
4.1 Termination

ACL2 requires each recursive function definition to be accompanied by a proof that it will terminate in a bounded amount of time. Such a proof is accomplished by defining a function-specific *measure* (in many cases, determined automatically by ACL2) and proving that the measure strictly decreases for each recursive call within the function body. However, termination proofs pose a challenge in many applications where pointer chasing is involved [19, 18]. In the context of `FAT32`, when transforming a LoFAT instance into an HiFAT instance, pointer chasing is necessary both for regular files and directory files, neccesitating some care towards the avoidance of non-terminating computation in both cases without incurring the overhead of general-purpose cycle detection algorithms.

Each file's directory entry contains the index of its first cluster, and its contents are determined by following its clusterchain in the file allocation table and concatenating together the corresponding clusters. This is subject to potential cycles in the clusterchain. These can be mitigated because of the `FAT32` stipulation of maximum lengths for regular files and directory files; the measure for the recursion becomes the remaining length of the file, which decreases with each cluster visited in the file allocation table.

For directory files the problem is more involved; since the transformation of a directory on disk to a directory tree involves the recursive transformation of all sub-directories, it is possible for a sub-directory cycle to arise. Consider an ill-formed disk image where the top-level directory `etc` contains an entry for the sub-directory `apt` and `apt` in turn contains a directory entry for `etc`; in this scenario, it is possible for the algorithm to spin over the fictitious sub-directories `/etc/apt/etc`, `/etc/apt/etc/apt`, ... A loop-stopping criterion is required which accepts all disk images which are free of cycles and returns an error for all disk images with sub-directory cycles. `POSIX` defines the constant `PATH_MAX` to bound the length

(a) A directory tree with a deleted file.



(b) An equivalent rearranged directory tree with the deleted file removed.

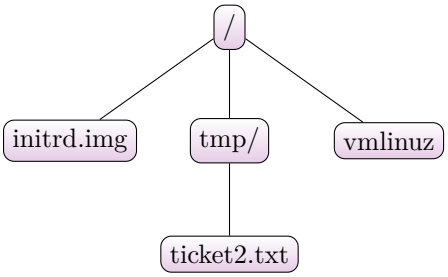


Figure 1 Two equivalent directory trees.

of a pathname, but it is inconsistently used by implementations [30]; thus a naive solution based on a maximum directory nesting depth is likely to reject valid disk images. A better way is to examine the filesystem at the granularity of directory entries, noting that these cannot exceed the total space available in the data region. Thus, an argument `entry-count` is added to `lofat-to-hifat-helper` (a recursive helper function for the transformation) and designated as the measure, with decrementation for each entry counted when making recursive calls. `entry-count` is instantiated to the maximum number of entries possible in the data region in `lofat-to-hifat` (the top-level wrapper function); this ensures that all valid filesystem instances are accepted without an error and demonstrates the existence of a cycle in each case where the total possible number of directory entries is exceeded.

4.2 Equivalence

Several useful filesystem correctness properties depend, for their proofs, on a notion of equivalence between two filesystem instances. While defining such a notion of equivalence, it is desirable to leave room for different implementation choices for cluster allocation, garbage collection, and other such details. Some constraints which characterize such an equivalence relation follow, and are illustrated in Figure 1.

- Modulo rearrangement, each directory in two equivalent filesystem instances should contain the same regular files and, recursively, the same sub-directories. This ensures that looking up the same pathname in both yields the same results.
- Directory entries for the current directory (`.`) and the parent directory (`..`) should be disregarded, since they do not refer to new unique files. The same is true for deleted files' directory entries.
- Re-allocation of clusters for the contents of a given file without changing the contents should be disregarded.
- Changes to the redundant copies of the file allocation table should be disregarded.
- Changes to volume-level metadata, such the size of a cluster or the total number of clusters in the filesystem, should be taken into account only if they result in the deletion of file data.

Also, to simplify the verification task, creation times, access times, write times, and long names for files are set aside, even though this limits the reasoning which can be carried out about programs which rely on these for their correct operation, such as the incremental compilation system Make [43].

Listing 2 `hifat-equiv`.

```
(defun hifat-equiv (m1-file-alist1 m1-file-alist2)
  (b* ((m1-file-alist1 (hifat-file-alist-fix m1-file-alist1))
       (m1-file-alist2 (hifat-file-alist-fix m1-file-alist2)))
    (and (hifat-subsetp m1-file-alist1 m1-file-alist2)
         (hifat-subsetp m1-file-alist2 m1-file-alist1))))
```

Listing 3 `lofat-equiv`.

```
(defund-nx lofat-equiv (fat32-in-memory1 fat32-in-memory2)
  (b* (((mv fs1 error-code1) (lofat-to-hifat fat32-in-memory1))
       (good1 (and (lofat-fs-p fat32-in-memory1)
                   (equal error-code1 0)))
       ((mv fs2 error-code2) (lofat-to-hifat fat32-in-memory2))
       (good2 (and (lofat-fs-p fat32-in-memory2)
                   (equal error-code2 0)))
       ((unless (and good1 good2)) (and (not good1) (not good2))))
    (hifat-equiv fs1 fs2)))
```

At HiFAT, the most abstract level, we meet the above requirements by first defining a subset relation `hifat-subsetp`; and then defining the equivalence relation `hifat-equiv` (Listing 2) in terms of subsets as discussed in Section 2.3.

At LoFAT, the next lower level of abstraction, we define the equivalence relation `lofat-equiv` in terms of the transformation between LoFAT and HiFAT, once again grouping ill-formed LoFAT instances (that is, instances which return a non-zero error code when transformed to HiFAT) into the same equivalence class (Listing 3). Finally, we define an equivalence relation for disk images. These are strings, each representing the entire contents of the image. This equivalence relation, EqFAT, groups all ill-formed disk images which cannot be transformed to a valid LoFAT instance into the same equivalence class (Listing 4).[3]

[3] Both these functions are considered *non-executable* in ACL2, because they reference two instances of the stobj `fat32-in-memory` at the same time. They are thus introduced with `defund-nx` [3] instead of the usual `defun` and can only be used for reasoning. These functions also use `b*` [2], an ACL2 extension of the Common Lisp `let*` with a more flexible syntax for let-bindings.

Listing 4 EqFAT.

```
(defund-nx eqfat (str1 str2)
  (b* (((mv fat32-in-memory1 error-code1)
        (string-to-lofat (create-fat32-in-memory) str1))
       (good1 (and (stringp str1) (equal error-code1 0)))
       ((mv fat32-in-memory2 error-code2)
        (string-to-lofat (create-fat32-in-memory) str2))
       (good2 (and (stringp str2) (equal error-code2 0)))
       ((unless (and good1 good2)) (and (not good1) (not good2))))
    (lofat-equiv fat32-in-memory1 fat32-in-memory2)))
```

4.3 Invertibility and error codes

HiFAT instances are directory trees, defined recursively; thus, proofs about HiFAT generally require induction. Many theorems about recursive functions can be automatically proved in ACL2 through inference of induction schemes; however, an induction scheme can also be explicitly designated in order to control the inductive formulation of a theorem. In such an induction scheme, the induction hypothesis can be strengthened or weakened as needed.

Between HiFAT and LoFAT, transformations `hifat-to-lofat` and `lofat-to-hifat` are defined, and must be proved to be inverses of each other under the appropriate equivalence relations. This is a claim in two parts: transforming m_1 to m_2 and back should result in an m_1' related to m_1 by `hifat-equiv`; and transforming m_2 to m_1 and back should result in an m_2' related to m_1 by `lofat-equiv`.

The proof of the first part of this claim (as illustrated in Figure 2a) turns out to also involve error codes; no claims can be made about the invertibility of a transformation if it returns an error. Thus, the proof requires a strengthened induction hypothesis to show in tandem that the error code returned by `lofat-to-hifat` while transforming m_2 back to m_1' is 0 (signifying no error.) This induction is the most complex proof undertaken in this work, since it requires an induction scheme to be defined on functions which interpret binary file formats.

The second part of this claim is true by the definition of `lofat-equiv` and an instantiation of the first claim (Figure 2b).

It is also necessary to prove the correctness of the transformations between instances of LoFAT and `FAT32` disk images (strings). These transformations, `lofat-to-string` and `string-to-lofat`, are proved to be mutual inverses under the equivalence relations `equal` (first-order equality) and EqFAT, respectively. As before, one direction of the claim is proved and then instantiated to prove the other by the definition of EqFAT (Figure 2c).

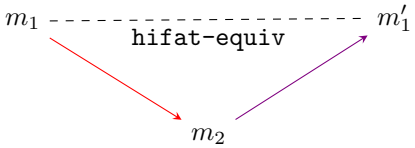
Equality is known to refine all equivalence relations; thus, `equal` refines `lofat-equiv`, and the correctness of the transformations between disk images and HiFAT instances through the intermediate level LoFAT can finally be certified by composing these proofs (Figure 2d).

4.4 Correctness of the specification

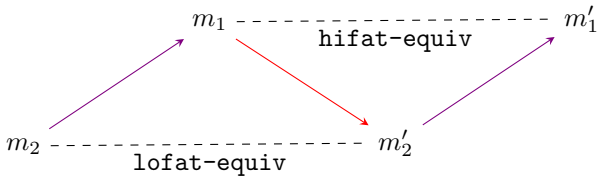
Prior filesystem verification work [6] has shown the proof process to uncover subtle bugs in the specification of a filesystem which would otherwise have remained hidden; this has matched our experience modeling and verifying HiFAT and LoFAT. We note some examples of bugs we found in our models in this manner.

- In `FAT32`, the first two entries in the file allocation table are reserved for volume-level metadata; thus, the size of the file allocation table must exceed the number of available clusters by at least two. Additionally, there may be a number of unused entries at the end of the file allocation table, since it must span an integer number of sectors. These differences led us to place incorrect upper bounds on the root cluster of the filesystem, the first cluster of an arbitrary file, and the length of the file allocation table. These errors were discovered and rectified during the proofs of correctness of our transformations.
- An off-by-one bug caused the directory bit of a directory entry to be wrongly set; this was also identified and rectified in the process of proving the transformations correct.

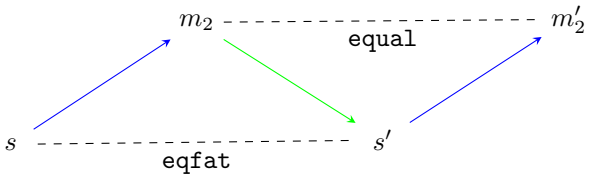
In addition, some bugs in parts of the code which were not immediately verified were found outside of the theorem proving process, by means of co-simulation. One example was the case of a `FAT32` volume in which the root had no directory entries, which is possible in



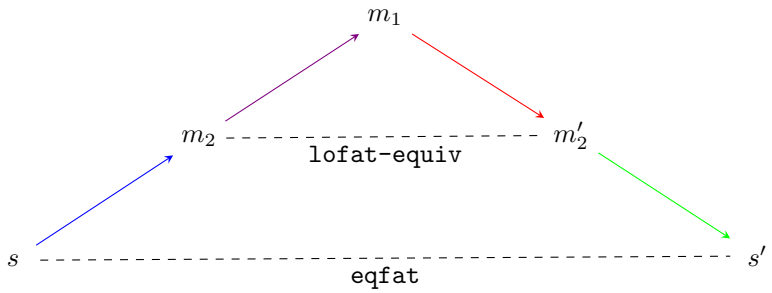
(a) `hifat-to-lofat-inversion` is derived as a corollary of an induction proof (Section 4.3).



(b) `hifat-to-lofat-inversion` is instantiated in order to derive `lofat-to-hifat-inversion`.



(c) Similarly, `lofat-to-string-inversion` (not shown) is instantiated in order to derive `string-to-lofat-inversion`. Here, m_2 and m_2' are LoFAT instances and s and s' are disk image strings.



(d) `string-to-hifat-inversion` is a corollary of `lofat-to-hifat-inversion`.

Figure 2 Equivalences.

`FAT32` because only directories other than the root are required to have `.` and `..` entries. `FAT32` constrains each directory file to have at least one cluster, and this constraint had been omitted from the specification. Over a number of co-simulation tests with the Linux `FAT32` implementation, this bug was discovered and fixed.

5 Evaluation

5.1 Co-simulation

Co-simulation is a necessary component of formal verification efforts when binary compatibility is the aim, in order to validate the correspondence of the verified model with the software/hardware system in question [17]. A challenge, from the perspective of co-simulation as well as from the perspective of reducing the risk of bugs in unverified code, is the choice of an interface to the operating system. We develop our co-simulation tests as reads and writes

on disk images; thus, the potential for bugs outside the verified part of the implementation is confined to specification and implementation errors in ACL2's built-in I/O operations (and indeed, one such bug was found during this development [22]).

Among existing FAT32 implementations, we have chosen to co-simulate with the Linux kernel implementation of FAT32 (as mediated by the GNU Coreutils) and the mtools [31]. The mtools perform various operations such as copying and deletion of files on a given FAT32 disk image or block device, which makes co-simulation relatively straightforward. Co-simulation with the Coreutils involves more steps since they are agnostic towards the underlying filesystem; each test proceeds by mounting a disk image, running the program in question, and unmounting. This co-simulation setup checks the correctness of file operations, without changing filesystem state, in the two following scenarios (which are not mutually exclusive).

1. File operations which retrieve data from the filesystem, such as pread [26], result in output which must be compared to that of the canonical FAT32 implementation. The program diff [15] effects this comparison.
2. File operations which modify the state of the filesystem, such as pwrite [27], result in a modification to the disk image. The modified disk image must then be compared to a disk image modified by the canonical FAT32 implementation; this is done by an ACL2 program which checks whether EqFAT holds for the two images.

5.2 POSIX interface and tests

Table 1 summarizes the subset of the POSIX system calls which have been implemented. The Linux convention is for system calls to return an error code, which is zero if and only if no error occurred, and set the global variable errno; together, these allow an application program making a system call to include error-handling code based on whether an error arose and why. In the ACL2 setting, where there are no global variables, FAT32 system calls maintain the convention by including the "return value" and errno value in the values they return. This matches the Linux implementation of FAT32; thus, for example, when rmdir is called on a non-empty directory, the filesystem instance is returned unmodified along with a non-zero "return value" and an errno value of EEXIST, as specified in the POSIX manual page for rmdir [28]. File descriptors, for operations such as pread and pwrite, are provided through a straightforward implementation of a file table and a file descriptor table, similar to Synergy's [8] implementation; however, the interaction of multiple processes with the filesystem is not yet supported.

This subset suffices for writing and testing ACL2 programs which co-simulate a number of programs from the Coreutils suite (Figure 3a) and from the mtools (Figure 3b). The co-simulation test suite also includes a basic sanity check which compares the output of the program mkfs.fat -v, which creates a FAT32 disk image and prints a textual summary of volume-level metadata [20], with the output of an ACL2 program which reports the same metadata.

For each system call except statfs [29], a version applicable to HiFAT is first developed, and then the LoFAT version is implemented by first transforming the filesystem instance to an HiFAT instance, and then performing the HiFAT version of the system call. If the system call results in a change to the filesystem state, the HiFAT instance is then transformed back to an LoFAT instance at the end. This approach is correct by construction, by the definition of lofat-equiv.

Table 1 `POSIX` syscalls implemented.

Syscall	LoFAT implementation	LoFAT implementation through HiFAT transformation
`close`	✓	✓
`lstat`	✓	✓
`mkdir`		✓
`mknod`		✓
`open`	✓	✓
`pread`	✓	✓
`pwrite`		✓
`rename`		✓
`rmdir`		✓
`statfs`	✓	
`truncate`		✓
`unlink`		✓

Program
`cp`
`ls`
`mkdir`
`mv`
`rm`
`rmdir`
`stat`
`truncate`
`unlink`
`wc`

(a) Coreutils programs co-simulated.

Program
`mcopy`
`mdel`
`mdeltree`
`mmd`
`mmove`
`mrd`
`mren`

(b) `mtools` programs co-simulated.

Figure 3 Syscalls and co-simulation tests.

Co-simulation tests almost always require more than one system call on a given disk image. When this happens, contiguous sequences of operations on the HiFAT instance are carried out while eliding back and forth transformations between HiFAT and LoFAT until the moment of writing back to disk. This elision is sound, as shown by the theorem `lofat-to-hifat-inversion` (Figure 2b), and places HiFAT in a role similar to that of a cache.

`statfs` [29] is an exception and must be implemented at the LoFAT level, since it reports volume-level metadata, such as the total space and free space in the filesystem, which is abstracted away in HiFAT. This also limits the extent to which `statfs`, and programs which use it such as `stat` (more precisely, `stat -f`/`stat --file-system`), can be incorporated into co-simulation tests, because volume-level metadata can differ between filesystems which are identical in terms of the files contained. For instance, the directory tree in Figure 1a contains the same files and directories as the tree in Figure 1b but may still occupy more space on disk, because the directory entry for the deleted file `/tmp/ticket1.txt` still exists and may cause the contents of the directory `/tmp` to occupy an additional cluster.

Since HiFAT is a sparse format for representing the filesystem state, the overheads for these transformations are small enough for co-simulation testing to be feasible; a further improvement in efficiency comes from verifying the guards of all the system calls. However,

considering there to be room for improvement in terms of removing these overheads, we also construct provably equivalent implementations of `open`, `pread` and `lstat` for LoFAT which skip the transformation from LoFAT to HiFAT. We are working on doing this for the remaining system calls.

5.3 Performance

This implementation of `FAT32` loads up ACL2 in order to execute the model, which necessarily imposes a lower bound on the time taken for a co-simulation test with a program. However, reasonably quick co-simulation is essential to achieving breadth as well as depth in the co-simulation coverage; thus, optimizations become an important part of the modeling effort. The following two design choices are significant.

1. LoFAT is implemented as a stobj, even though this complicates syntax and reasoning, in order to avoid the performance penalties associated with creating and destroying large immutable data structures each time a single element in the in-memory `FAT32` representation is modified. Transformations to and from HiFAT would have been prohibitive in co-simulation tests without a guard-verified stobj implementation of LoFAT.
2. String representations of data are chosen over byte-list or character-list representations wherever possible. While lists are simpler to reason about, it makes a difference to be able to use the efficient implementations of the built-in string operations `concatenate` (string concatenation), `subseq` (substring extraction) and so on while extracting and reconstructing file contents and working with disk images. In addition, `read-file-into-string` [5], a recent addition to ACL2, provides a fast `mmap`-based [25] alternative to ACL2's character-oriented I/O operations for the use case of reading information from a `FAT32` disk image and populating the fields of the LoFAT instance. Thus, by choosing to work with a string representation for disk images, and by choosing to represent the contents of the data region as an array of cluster-sized strings (Section 3) to take full advantage of the atomicity of clusters in `FAT32`, performance penalties associated with conversions between strings and lists are avoided.

Within the parameters of this design, two optimizations are made possible by ACL2's logical story for I/O operations. Both of these avoid the construction of intermediate string representations while transforming between disk images and LoFAT instances, in order to reduce the associated overheads, while retaining the abstraction of the disk image as a string. Specifically, while writing back to disk, the explicit construction of a data region string would involve an expensive concatenation of all the clusters; this is omitted by instead writing back all the clusters in sequential order. Similarly, while reading a disk image, the population of the data region after having read the disk image would involve multiple `subseq` operations for extracting the clusters, with significant memory allocation overhead; this is avoided by instead calling `read-file-into-string` multiple times with the appropriate offsets to read the pertinent clusters from the disk image directly into the data region of the LoFAT instance. For both these optimizations, `mbe` is used (Section 2.1) to show that the optimized ACL2 code has the same effect. This is in keeping with the refinement style of proof used throughout this work: when a conceptually simple sequence of I/O operations is replaced with a more complex sequence, the simpler sequence is, in a sense, a specification which is refined. This is also how the model development remains tractable as it evolves: while replacing an earlier implementation, in which disk image strings were explicitly handled, with the optimized one, the co-simulation test suite showed the absence of regressions but the proof that both implementations work the same way enabled much greater confidence.

Table 2 Timing disk image I/O.

Disk image size	Read time	Write time
128 MB	2.48 s	4.14 s
256 MB	3.58 s	7.91 s
512 MB	7.52 s	15.46 s
1024 MB	15.92 s	24.87 s

Table 3 Code summary.

Lines of code (models and proofs)	24,905
Lines of code (co-simulation)	619
Co-simulation tests	31

As a result of these design choices and optimizations, co-simulations involving relatively large disk images become possible. For comparison, the in-memory sparse filesystem `tmpfs` [42] usually mounts volumes of size 1 GB to 10 GB on a standard consumer laptop; we have been able to run tests involving disk images of size 1 GB in the same environment. Further, since HiFAT is by nature a sparse format, allocating memory only for file contents, there is little overhead associated with most file operations which affect only the intermediate HiFAT instance. Table 2 lists some timing results for such tests in terms of reading and writing `FAT32` disk images, and Table 3 summarizes statistics pertaining to the magnitude of the modeling effort.[4]

6 Related work

Much of the existing filesystem verification work has taken on the task of synthesizing a new filesystem, developed in a way that simplifies the proofs of filesystem properties of interest.

An early effort was Synergy [8], in which a filesystem was developed and verified according to a specification in ACL2. However, binary compatibility was not a goal for this work, and the design choice of maintaining a mapping from filenames to file contents did not take into account the complexity of path resolution. The `POSIX`-like formulation chosen by Synergy and by other efforts on the more abstract end of the filesystem verification spectrum [16] was an inspiration for an earlier phase of the present work [34], in which `FAT32` models were developed in an incremental fashion from a series of abstract filesystem models, adding more realistic filesystem features in each of the models.

FSCQ [11], developed with Coq [7], is a high-performance FUSE-based [39] filesystem with formally verified crash consistency properties. However, FSCQ exports its executable code to Haskell, and Haskell's FUSE interface to the operating system is, by necessity, unverified. A bug in this interface was discovered through the use of Bounded Black-Box testing, a methodology for automatically testing the data persistence behavior of filesystems [36] and later fixed. FSCQ has been followed by DFSCQ [10], which formally specifies the `fsync` and `fdatasync` file operations, and SFSCQ [21] which proves two-safety confidentiality properties in terms of data noninterference.

COGENT [6], developed with the help of Isabelle/HOL [38], takes a different tack by providing a verified compiler for turning a domain-specific filesystem specification language into verified C code implementing a filesystem.

[4] These statistics were generated using David A. Wheeler's "SLOCCount".

Z3 [14], a non-interactive theorem prover, has also been used for filesystem verification through SMT solving. Hyperkernel [37] attempts a verification of the xv6 [12] microkernel by simplifying it to make the problem tractable through SMT solving. This simplification replaced all kernel data structures with fixed-length implementations, leading all kernel operations (including file operations) to become constant-time. A more filesystem-focused effort is Yggdrasil [41], which verifies a number of filesystem calls by providing a refinement proof showing that its concrete filesystem implementation adheres to a formal specification. This is similar to what FSCQ does, but Yggdrasil's Z3-based verifier achieves this automatically by means of symbolic execution.

7 Future work

While a number of properties have been proved about the `FAT32` models and extensive co-simulation tests have been carried out, there remain research questions to be answered in terms of concurrency and generalization to other filesystems.

Within the `FAT32` context, a straightforward extension of this work would be to provide an interface closer to that of `POSIX`, for instance through FUSE [39]. This would allow programs written in C and other languages, which use file descriptors, to interface with our implementation. This, in turn, would help mature the model by facilitating the use of automated testing methodologies such as Bounded Black-Box testing [36] in order to discover more bugs. This would also offer greater opportunities to seek performance gains, including by skipping the transformations to the intermediate HiFAT representation and back in favor of direct manipulation of LoFAT instances or disk images for file operations (as already demonstrated with `open`) and by using a demand paging-like algorithm to turn LoFAT into a sparse format only storing clusters allocated to files.

We have taken some steps to generalize this work, including the development of macros (Section 2.2), and we are interested in applying this filesystem verification methodology to a binary-compatible verification effort for more complex filesystems with features such as hard linking and crash consistency. The ext4 filesystem [33], which provides crash consistency by means of journaling, is an example.

We are also interested in extending this work to incorporate a model of concurrency, along the same lines as prior work on formalization of microprocessor architectures in theorem proving environments. This would allow the filesystem to serve as a precise specification for correct filesystem behavior in a multiprogramming environment. Making use of such a specification, it would become possible to prove the correctness of programs which concurrently interact with the filesystem and make use of the functionality provided by the operating system to avoid race conditions.

8 Conclusion

A byte-level examination of the specification and existing implementations of a filesystem is a necessary part of a verification effort for it to enable reasoning about the behavior of programs which interact with the filesystem. The recursive definition of the directory tree is central to the study of filesystems; thus, induction is also central to the analysis. Defining and using notions of equivalence between directory trees which disregard implementation details is essential for demonstrating that our `FAT32` model and existing `FAT32` implementations operate the same way. The logical decoupling enabled by `mbe` helps keep formal developments involving binary file formats tractable as they evolve through various optimizations, including optimizations based on the logical story of I/O.

This paper's contribution is the general-purpose methodology for binary compatible filesystem verification which makes use of the above techniques and is illustrated through LoFAT and HiFAT. This makes reasonably good performance possible for a disk-image manipulation methodology of verified filesystem implementation, which is sufficient for validating existing filesystem implementations by means of extensive co-simulation testing.

References

1 Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991. doi:10.1016/0304-3975(91)90224-P.

2 ACL2 Community. ACL2 documentation for B*. See URL http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____B_A2.

3 ACL2 Community. ACL2 documentation for DEFUND-NX. See URL http://www.cs.utexas.edu/users/moore/acl2/current/manual/index.html?topic=ACL2____DEFUND-NX.

4 ACL2 Community. ACL2 documentation for MBE. See URL http://www.cs.utexas.edu/users/moore/acl2/current/manual/index.html?topic=ACL2____MBE.

5 ACL2 Community. ACL2 documentation for READ-FILE-INTO-STRING. See URL http://www.cs.utexas.edu/users/moore/acl2/current/manual/index.html?topic=ACL2____READ-FILE-INTO-STRING.

6 Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, et al. Cogent: Verifying high-assurance file system implementations. *ACM SIGPLAN Notices*, 51(4):175–188, 2016. doi:10.1145/2872362.2872404.

7 Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013. doi:10.1007/978-3-662-07964-5.

8 William R. Bevier and Richard M. Cohen. An executable model of the Synergy file system. Technical report, Technical Report 121, Computational Logic, Inc, 1996.

9 Robert S. Boyer and J Strother Moore. Single-threaded objects in ACL2. In *International Symposium on Practical Aspects of Declarative Languages*, pages 9–27. Springer, 2002. doi:10.1007/3-540-45587-6_3.

10 Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 270–286. ACM, 2017.

11 Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for Certifying the FSCQ File System. In *USENIX Annual Technical Conference*, 2016.

12 Russ Cox, M. Frans Kaashoek, and Robert T. Morris. Xv6, a simple Unix-like teaching operating system, 2016. URL: http://pdos.csail.mit.edu/6.828/2014/xv6.html.

13 Jared Davis. Reasoning about ACL2 file input. In *Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 117–126. ACM, 2006.

14 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

15 Paul Eggert, Mike Haertel, David Hayes, Richard Stallman, and Len Tower. diff (1)-Linux manual page, accessed: 07 Sep 2018.

16 Philippa Gardner, Gian Ntzik, and Adam Wright. Local reasoning for the POSIX file system. In *European Symposium on Programming Languages and Systems*, pages 169–188. Springer, 2014. doi:10.1007/978-3-642-54833-8_10.

17 Shilpi Goel, Warren A. Hunt Jr., Matt Kaufmann, and Soumava Ghosh. Simulation and formal verification of x86 machine-code programs that make system calls. In *Formal Methods*

in Computer-Aided Design (FMCAD), 2014, pages 91–98. IEEE, 2014. doi:10.1109/FMCAD.2014.6987600.

18 David Greve. Address enumeration and reasoning over linear address spaces. In *5th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2004), Austin, TX*, 2004.

19 David Greve and Matt Wilding. Dynamic datastructures in ACL2: A challenge, 2002. URL: http://hokiepokie.org/docs/festival02.txt.

20 Dave Hudson, Peter Anvin, and Roman Hodek. mkfs.fat (8)-Linux manual page, accessed: 09 Jul 2018.

21 Atalay Ileri, Tej Chajed, Adam Chlipala, Frans Kaashoek, and Nickolai Zeldovich. Proving confidentiality in a file system using DISKSEC. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 323–338, 2018.

22 Matt Kaufmann. Fixed read-file-into-string bug. (commit message), July 2018. URL: https://github.com/acl2/acl2/commit/8388ac10289d5cab791953238057294604af6d60.

23 Matt Kaufmann and J Strother Moore. ACL2: An industrial strength version of Nqthm. In *Proceedings of 11th Annual Conference on Computer Assurance (COMPASS'96)*, pages 23–34. IEEE, 1996.

24 Michael Kerrisk. errno (3)-Linux manual page, accessed: 07 Sep 2018.

25 Michael Kerrisk. mmap (2)-Linux manual page, accessed: 09 Dec 2018.

26 Michael Kerrisk. pread (2)-Linux manual page, accessed: 09 Jul 2018.

27 Michael Kerrisk. pwrite (2)-Linux manual page, accessed: 09 Jul 2018.

28 Michael Kerrisk. rmdir (2)-Linux manual page, accessed: 17 Mar 2019.

29 Michael Kerrisk. statfs (2)-Linux manual page, accessed: 09 Dec 2018.

30 Evan Klitzke. PATH_MAX is tricky, April 2017. URL: https://eklitzke.org/path-max-is-tricky.

31 Alain Knaff. mtools (1)-Linux manual page, accessed: 09 Dec 2018.

32 Leslie Lamport. Verification and specification of concurrent programs. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*, pages 347–374. Springer, 1993. doi:10.1007/3-540-58043-3_23.

33 Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33, 2007.

34 Mihir Parang Mehta. Formalising Filesystems in the ACL2 Theorem Prover: an Application to FAT32. *Proceedings of the 15th International Workshop on the ACL2 Theorem Prover and Its Applications*, page 18, 2018.

35 Microsoft. Microsoft Extensible Firmware Initiative FAT32 File System Specification, December 2000. URL: https://download.microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/fatgen103.doc.

36 Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. *arXiv preprint*, 2018. arXiv:1810.02904.

37 Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP'17, pages 252–269, New York, NY, USA, 2017. ACM. doi:10.1145/3132747.3132748.

38 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002. doi:10.1007/3-540-45949-9.

39 N. Rath and M. Szeredi. The reference implementation of the Linux FUSE (Filesystem in Userspace) interface, 2018.

40 Murray Shanahan. The Frame Problem. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, 2016.

41 Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-Button Verification of File Systems via Crash Refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, 2016.

42 Peter Snyder. tmpfs: A virtual memory file system. In *In Proceedings of the Autumn 1990 European UNIX Users' Group Conference*, pages 241–248, 1990.

43 Richard M Stallman. GNU Make, 1988.

Ornaments for Proof Reuse in Coq

Talia Ringer
University of Washington, USA
tringer@cs.washington.edu

Nathaniel Yazdani
University of Washington, USA
nyazdani@cs.washington.edu

John Leo
Halfaya Research, USA
leo@halfaya.org

Dan Grossman
University of Washington, USA
djg@cs.washington.edu

Abstract

Ornaments express relations between inductive types with the same inductive structure. We implement fully automatic proof reuse for a particular class of ornaments in a Coq plugin, and show how such a tool can give programmers the rewards of using indexed inductive types while automating away many of the costs. The plugin works directly on Coq code; it is the first ornamentation tool for a non-embedded dependently typed language. It is also the first tool to automatically identify ornaments: To lift a function or proof, the user must provide only the source type, the destination type, and the source function or proof. In taking advantage of the mathematical properties of ornaments, our approach produces faster functions and smaller terms than a more general approach to proof reuse in Coq.

2012 ACM Subject Classification Software and its engineering → Formal software verification

Keywords and phrases ornaments, proof reuse, proof automation

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.26

Supplement Material The Coq plugin, examples, and case study code for this paper can be found at `http://github.com/uwplse/ornamental-search/tree/itp+equiv`.

Acknowledgements We thank Jasper Hugunin, James Wilcox, Jason Gross, Pavel Panchekha, and Marisa Kirisame for ideas that helped inform tool design. We thank Thomas Williams, Josh Ko, Matthieu Sozeau, Cyril Cohen, Nicolas Tabareau, and Enrico Tassi for help navigating related work. We thank Emilio J. Gallego Arias, Gaëtan Gilbert, Pierre-Marie Pédrot, and Yves Bertot for help understanding Coq plugin APIs. We thank Shachar Itzhaky and Tej Chajed for ideas for future directions. We thank the UW and UCSD programming languages labs for feedback. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-1256082. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

1 Introduction

Indexed inductive types make it possible to internalize data into the type level, eliminating the need for certain functions and proofs. Consider, for example, a theorem from the Coq standard library [17] which states that mapping a function over lists preserves length:

```
map_length T1 T2 (f : T1 → T2) : ∀ (l : list T), length (List.map f l) = length l.
```

```
list (T : Type) : Type :=
| nil : list T
| cons :
    T → list T → list T.
```

```
vector (T : Type) : nat → Type :=
| nilV : vector T 0
| consV :
    ∀ (n :  nat), T → vector T n → vector T (S n).
```

Figure 1 A `vector` (right) is a `list` (left) indexed by its length (highlighted in orange).

One way to eliminate the need for this theorem is to internalize the length of a list into its type, creating a dependently typed vector (Figure 1). The map function for vectors in Coq's standard library, for example, carries a proof that it preserves length:

```
Vector.map {T1} {T2} (f : T1 → T2) : ∀ (n : nat) (v : vector T1 n), vector T2 n.
```

so that a theorem like `map_length` is no longer necessary.

Unfortunately, for all of the benefits they bring, indexed inductive types are notoriously difficult to use. Dependently typed vectors, for example, impose proof obligations about their lengths on the user; these can quickly spiral out of control. In recent coq-club threads asking for advice on how to use dependently typed vectors, experts called them "not suitable for extended use" [7] and noted that "almost no one should be using [them] for anything" [8].

We show how *proof reuse* – reusing existing proofs to derive new proofs – can tackle many of the challenges posed by indexed inductive types, allowing the user to move between unindexed and indexed versions of a type (for example, lists and vectors) and reap the benefits of indexed types without many of the costs. We focus in particular on the benefits of this approach in deriving functions and proofs for fully-determined indexed types, when the index is a fold over the unindexed version (such as the length of a list). In our approach, the user writes functions and proofs over the unindexed version, and a tool then automatically *lifts* those functions and proofs to the indexed version. The user can then switch back to working with the unindexed version by running the tool in the opposite direction. In that way, the user can use lists when lists are convenient, and vectors when vectors are convenient.

Our approach uses *ornaments* [23], which express relations between types that preserve inductive structure, and which enable lifting of functions and proofs along those relations. Recent work introduced ornaments to a subset of ML and was heavily focused on automatically lifting functions [33]; until now, such an approach was not available in a dependently typed language. Existing implementations of ornaments in dependently typed languages work only in embedded languages, and have little to no automation [20, 23, 11].

Our main contribution is a Coq plugin for automatic function and proof reuse using ornaments. Our plugin Devoid (Dependent Equivalences Via Ornamenting Inductive Definitions) works directly on Coq code, rather than on an embedded language. Devoid automates lifting functions and proofs along *algebraic ornaments* [23], a particular class of ornaments that represent fully-determined indexed types like lists and vectors. Devoid implements an algorithm to search for ornaments between these types – to the best of our knowledge, the first search algorithm for ornaments – and an algorithm to lift functions and proofs along the ornaments it discovers.

We motivate (Section 2), specify (Section 3), and formalize (Section 4) the search and lifting algorithms that Devoid implements (Section 5). A comparison to a more general proof reuse approach (Section 6) demonstrates the benefits of using ornaments: Devoid imposes less of a proof burden on the user, and produces smaller terms and faster functions.

2 Motivating Example: Porting a Library

Devoid is a plugin for Coq 8.8; it can be found in the repository linked to as **Supplement Material** under the abstract of this paper. To see how it works, consider an example using the types from Figure 1, the code for which is in `Example.v`. In this example, we lift two list zip functions and a proof of a theorem relating them from the Haskell CoreSpec library [29]:

```
zip {T₁ T₂}: list T₁ → list T₂ → list (T₁ * T₂).
zip_with {T₁ T₂ T₃} (f : T₁ → T₂ → T₃): list T₁ → list T₂ → list T₃.
zip_with_is_zip {T₁ T₂}: ∀(l₁:list T₁)(l₂:list T₂), zip_with pair l₁ l₂ = zip l₁ l₂.
```

Devoid runs a preprocessing step before lifting, which we describe in Section 5; we assume this step has already run. We use the cyan background color to denote tool-produced terms and the names that refer to them. We run Devoid to lift functions and proofs from lists to vectors, but it can also lift in the opposite direction.

Step 1: Search. We first use Devoid's `Find ornament` command to search for the relation between lists and vectors:

```
Find ornament list vector.
```

This produces functions which together form an equivalence (denoted $\simeq$):

```
list T ≃ Σ (n : nat).vector T n
```

Step 2: Lift. We then lift our functions and proofs along that equivalence using Devoid's `Lift` command. For example, to lift `zip`, we run the command:

```
Lift list vector in zip as zipV_p.
```

This produces a function with this type:

```
zipV_p {T₁ T₂} : Σ n.vector T₁ n → Σ n.vector T₂ n → Σ n.vector (T₁ * T₂) n.
```

that behaves like `zip`, but whose body no longer refers to lists. We lift our proof similarly:

```
Lift list vector in zip_with_is_zip as zip_with_is_zipV_p.
```

This produces a proof of the analogous result (denoting projections by π_l and π_r):

```
zip_with_is_zipV_p {T₁ T₂} : ∀ (v₁ : Σ n.vector T₁ n) (v₂ : Σ n.vector T₂ n),
  zip_withV_p pair (∃ (πl v₁) (πr v₁)) (∃ (πl v₂) (πr v₂)) =
  zipV_p (∃ (πl v₁) (πr v₁)) (∃ (πl v₂) (πr v₂)).
```

that no longer refers to lists, `zip`, or `zip_with` in any way.

Step 3: Unpack. The lifted terms operate over vectors whose lengths are *packed* inside of a sigma type. While this lets `Lift` provide strong theoretical guarantees, it can make it difficult to interface with the lifted code. We can recover *unpacked* terms using Devoid's `Unpack` command. For example, to unpack `zipV_p`, we run the command:

```
Unpack zipV_p as zipV.
```

This produces functions and proofs that operate directly over vectors, like `zipV`:

```
zipV {T₁ T₂} {n₁} (v₁ : vector T₁ n₁) {n₂} (v₂ : vector T₂ n₂) :
  vector (T₁ * T₂) (πl (zipV_p (∃ n₁ v₁) (∃ n₂ v₂))).
```

and `zip_with_is_zipV`:

```
zip_with_is_zipV : ∀ {T1 T2} {n1} (v1 : vector T1 n1) {n2} (v2 : vector T2 n2),
  eq_dep _ _ _ (zip_withV pair v1 v2) _ (zipV v1 v2).
```

Step 4: Interface. For any two inputs of the same length, `zipV` and `zipV_with` contain proofs that the output has the same length as the inputs. However, the types obscure this information. `Example.v` explains how to recover more user-friendly types, like that of `zipV_uf`:

```
zipV_uf {T1 T2} {n} : vector T1 n → vector T2 n → vector (T1 * T2) n.
```

and that of `zip_withV_uf`:

```
zip_withV_uf {T1 T2 T3} (f : T1 → T2 → T3) {n} :
  vector T1 n → vector T2 n → vector T3 n.
```

which both restrict input lengths. We can then use our lifted functions and proofs in client code. For example, we can write a different version of Coq's `BVand` function for bitvectors:

```
BVand {n} (v1 : vector bool n) (v2 : vector bool n) : vector bool n :=
  zip_withV_uf andb v1 v2.
```

By working over lists, we are able to reason about only the interesting pieces, thinking about indices only when relevant; in contrast, when writing proofs over vectors, even simple theorems can generate tricky proof obligations. With Devoid, the programmer can use the lifted functions and proofs to interface with code that uses vectors, then switch back to lists when vectors are unmanageable. In essence, ornaments form the glue between these types.

3 Specification

This section specifies the two commands that Devoid implements:

1. `Find ornament` searches for ornaments (specified in Section 3.1, described in Section 4.1).
2. `Lift` lifts along those ornaments (specified in Section 3.2, described in Section 4.2).

Algebraic Ornaments. Devoid searches for and lifts along *algebraic ornaments* in particular. An algebraic ornament relates an inductive type A to an indexed version of that type B with a new index of type I_B, where the new index is fully determined by a unique fold over A. For example, `vector` is exactly `list` with a new index of type `nat`, where the new index is fully determined by the `length` function. Consequentially, there are two functions:

```
ltv : list T                → Σ(n : nat).vector T n.
vtl : Σ(n : nat).vector T n → list T.
```

that are mutual inverses:

```
∀ (l : list T),                vtl (ltv l) = l.
∀ (v : Σ(n : nat).vector T n), ltv (vtl v) = v.
```

and therefore form the type equivalence from Section 2. Moreover, since the new index is fully determined by `length`, we can relate `length` to `ltv`:

```
∀ (l : list T), length l = π1 (ltv l).
```

In general, we can view an algebraic ornament as a type equivalence:

$$A\ \vec{i}\ \simeq\ \Sigma(n : I_B\ \vec{i}).B\ (\texttt{index}\ n\ \vec{i})$$

where $\vec{i}$ are the indices of A, I_B is a function over those indices, and the `index` operation inserts the new index n at the right offset. Such a type equivalence consists of two functions [32]:

$$\begin{array}{lll}\texttt{promote} & : A\ \vec{i} & \rightarrow\ \Sigma(n : I_B\ \vec{i}).B\ (\texttt{index}\ n\ \vec{i}).\\ \texttt{forget} & : \Sigma(n : I_B\ \vec{i}).B\ (\texttt{index}\ n\ \vec{i}) & \rightarrow\ A\ \vec{i}.\end{array}$$

that are mutual inverses:[1]

$$\begin{array}{lll}\texttt{section} & : \forall\ (a : A\ \vec{i}), & \texttt{forget}\ (\texttt{promote}\ a)\ = a.\\ \texttt{retraction} & : \forall\ (b_\Sigma : \Sigma(n : I_B\ \vec{i}).B\ (\texttt{index}\ n\ \vec{i})), & \texttt{promote}\ (\texttt{forget}\ b_\Sigma) = b_\Sigma.\end{array}$$

An algebraic ornament is additionally equipped with an indexer, which is a unique fold:

$$\texttt{indexer} : A\ \vec{i}\ \rightarrow\ I_B\ \vec{i}.$$

which projects the promoted index:

$$\texttt{coherence} : \forall (a : A\ \vec{i}),\ \texttt{indexer}\ a = \pi_l\ (\texttt{promote}\ a).$$

Following existing work [20], we call this equivalence the *ornamental promotion isomorphism*; when it holds and the indexer exists, we say that B is an algebraic ornament of A.

`Find ornament` searches for algebraic ornaments between types and is, to the best of our knowledge, the first search algorithm for ornaments. `Lift` then lifts functions and proofs along those ornaments, removing all references to the old type. Both commands make some additional assumptions for simplicity; detailed explanations for these are in `Assumptions.v`.

3.1 Find ornament

In their original form, ornaments are a programming mechanism: Given a type A, an ornament determines some new type B. We invert this process for algebraic ornaments: Given types A and B, Devoid searches for an ornament between them. This is possible for algebraic ornaments precisely because the indexer is extensionally unique. For example, all possible indexers for `list` and `vector` must compute the length of a list; if we were to try doubling the length instead, we would not be able to satisfy the equivalence.

`Find ornament` takes two inductive types and searches for the components of the ornamental promotion isomorphism between them:

- **Inputs**: Inductive types A and B, assuming:
  - B is an algebraic ornament of A,
  - B has the same number of constructors in the same order as A,
  - A and B do not contain recursive references to themselves under products, and
  - for every recursive reference to A in A, there is exactly one new hypothesis in B, which is exactly the new index of the corresponding recursive reference in B.
- **Outputs**: Functions `promote`, `forget`, and `indexer`, guaranteeing:
  - the outputs form the ornamental promotion isomorphism between the inputs.

`Find ornament` includes an option to generate a proof that the outputs form the ornamental promotion isomorphism; by default, this option is false, since `Lift` does not need this proof.

[1] The adjunction condition follows from section and retraction.

3.2 Lift

`Lift` lifts a term along the ornamental promotion isomorphism between A and B. That is, it lifts types to corresponding types and terms of those types to corresponding terms:

```
Lift list vector in list as vector_p. (* vector_p T := Σ (n : nat).vector T n *)
Lift list vector in (cons 5 nil) as v_p. (* v_p := ∃ 1 (consV 0 5 nilV) *)
```

Furthermore, it recursively preserves this equivalence, lifting non-dependent functions like `zip` so that they map equivalent inputs to equivalent outputs:

```
∀ {T₁ T₂} l₁ l₂, promote (zip l₁ l₂) = zipV_p (promote l₁) (promote l₂).
```

This intuition breaks down with dependent types. With equivalence alone, we can't state the relationship between `zip_with_is_zip` and `zip_with_is_zipV_p`, since the unlifted conclusion:

```
zip_with pair l₁ l₂ = zip l₁ l₂.
```

does not have the same type as the conclusion of the lifted version applied to promoted arguments; any relation between these terms must be heterogenous.

In particular, `Lift` preserves the *univalent parametric relation* [30], a heterogenous parametric relation that strengthens an existing parametric relation for dependent types [2] to make it possible to state preservation of an equivalence: Two terms t and t' are related by the univalent parametric relation $[[\Gamma]]_u \vdash [t]_u : [[T]]_u\ t\ t'$ at type T in environment Γ if they are equivalent up to transport. The details of this relation can be found in the cited work.

`Lift` preserves this relation using the components that `Find ornament` discovers, and additionally guarantees that the lifted term does not refer to the old type in any way:

- **Inputs**: The inputs to and outputs from `Find ornament`, along with a term t, assuming:
  - the assumptions and guarantees from `Find ornament` hold,
  - I_B is not A,
  - t is well-typed and fully η-expanded,
  - t does not apply `promote` or `forget`, and
  - t does not reference B.
- **Outputs**: A term t', guaranteeing:
  - if t is $A\ \vec{i}$, then t' is $\Sigma(n : I_B\ \vec{i}\ ).B\ (\texttt{index}\ n\ \vec{i}\ )$,
  - t' does not reference A, and
  - if in the current environment $\Gamma \vdash t : T$, then $[[\Gamma]]_u \vdash [t]_u : [[T]]_u\ t\ t'$.

`Lift` does not require a proof that the input components form the ornamental promotion isomorphism, but they must for the guarantees to hold. It can operate in either direction, promoting from A to packed B or forgetting in the opposite direction; the specification for the forgetful direction is similar, with extra restrictions on how B is used within t.

4 Algorithms

This section describes the algorithms that implement the specifications from Section 3.

Presentation. We present both algorithms relationally, using a set of judgments; to turn these relations into algorithms, prioritize the rules by running the derivations in order, falling back to the original term when no rules match. The default rule for a list of terms is to run the derivation on each element of the list individually.

$\langle i \rangle \in \mathbb{N}$, $\langle v \rangle \in$ Vars, $\langle s \rangle \in$ { Prop, Set, Type$\langle i \rangle$ }

$$
\begin{array}{l}
\langle t \rangle ::= \langle v \rangle \mid \langle s \rangle \mid \Pi\ (\langle v \rangle : \langle t \rangle)\ .\ \langle t \rangle \mid \\
\quad \lambda\ (\langle v \rangle : \langle t \rangle)\ .\ \langle t \rangle \mid \langle t \rangle\ \langle t \rangle \mid \\
\quad \mathrm{Ind}\ (\langle v \rangle : \langle t \rangle)\{\langle t \rangle, \ldots, \langle t \rangle\} \mid \mathrm{Constr}\ (\langle i \rangle, \langle t \rangle) \mid \\
\quad \mathrm{Elim}(\langle t \rangle, \langle t \rangle)\{\langle t \rangle, \ldots, \langle t \rangle\}
\end{array}
$$

$\Gamma \vdash t : T$ // type checking
$\Gamma \vdash t_1 \equiv_{\beta\delta\iota} t_2$ // definitional equality
t_β // beta-reduction
$t_{\beta\delta\iota}$ // normalization
$t\ [y\ /\ x]$ // substitution
$\xi\ (I,\ Q,\ c,\ C)$ // type of eliminator

Figure 2 CIC_ω syntax (left, from existing work [31]) and judgments and operations (right).

$$
\begin{array}{ll}
A\ :=\ \mathrm{Ind}(Ty_A : \Pi(\vec{i_A} : \vec{\mathrm{X}_A}).\mathrm{s}_A)\{\mathrm{C}_{A_1}, \ldots, \mathrm{C}_{A_n}\} & \mathrm{P}_A\ :=\ \Pi(\vec{i_A} : \vec{\mathrm{X}_A})(a : A\ \vec{i_A}).\mathrm{s}_A \\
B\ :=\ \mathrm{Ind}(Ty_B : \Pi(\vec{i_B} : \vec{\mathrm{X}_B}).\mathrm{s}_B)\{\mathrm{C}_{B_1}, \ldots, \mathrm{C}_{B_n}\} & \mathrm{P}_B\ :=\ \Pi(\vec{i_B} : \vec{\mathrm{X}_B})(b : B\ \vec{i_B}).\mathrm{s}_B \\
\forall 1 \leq i \leq n, & \\
\quad \mathrm{E}_{A_i}\ (p_A : \mathrm{P}_A)\ :=\ \xi(A,\ p_A,\ \mathrm{Constr}(i,\ A),\ C_{A_i}) & \mathrm{index}\ :=\ \mathrm{insert}\ (\mathrm{off}\ A\ B) \\
\quad \mathrm{E}_{B_i}\ (p_B : \mathrm{P}_B)\ :=\ \xi(B,\ p_B,\ \mathrm{Constr}(i,\ B),\ C_{B_i}) & \mathrm{deindex}\ :=\ \mathrm{remove}\ (\mathrm{off}\ A\ B)
\end{array}
$$

Figure 3 Common definitions for both algorithms.

Notes on Syntax. The language the algorithms operate over is CIC_ω with primitive eliminators; this is a simplified version of the type theory underlying Coq. Figure 2 contains the syntax (which includes variables, sorts, product types, functions, inductive types, constructors, and eliminators), as well as the syntax for some judgments and operations, the rules for which are standard and thus omitted. For simplicity of presentation, we assume variables are names; we assume that all names are fresh. As in Coq, we assume the existence of an inductive type Σ for sigma types with projections π_l and π_r; for simplicity, we assume projections are primitive. Throughout, we use $\vec{i}$ and $\{t_1, \ldots, t_n\}$ to denote lists of terms, and we use $\vec{i}[j]$ to denote accessing the element of the list $\vec{i}$ at offset j.

Common Definitions. The algorithms assume list insertion and removal functions `insert` and `remove`, plus two functions Devoid implements: `off` computes the offset of the new index of type I_B in B's indices, and `new` determines whether a hypothesis in a case of the eliminator type of B is new. Figure 3 contains other common definitions, the names for which are reserved: The `index` and `deindex` functions insert an index into and remove an index from a list at the index computed by `off`. Input type A expands to an inductive type with indices of types $\vec{\mathrm{X}_A}$, sort s_A, and constructors $\{\mathrm{C}_{A_1}, \ldots, \mathrm{C}_{A_n}\}$. P_A denotes the type of the motive of the eliminator of A, and each E_{A_i} denotes the type of the eliminator for the ith constructor of A. Analogous names are also reserved for input type B.

4.1 Find ornament

The `Find ornament` algorithm implements the specification from Section 3.1. It builds on three intermediate steps: one to generate each of `indexer`, `promote`, and `forget`. Figure 4 shows the algorithm for generating `indexer`. The algorithms for generating `promote` and `forget` are similar; Figure 5 shows only the derivations for generating `promote` that are different from those for generating `indexer`, and the derivations for generating `forget` are omitted.

4.1.1 Searching for the Indexer

Search generates the `indexer` by traversing the types of the eliminators for A and B in parallel using the algorithm from Figure 4, which consists of three judgments: one to generate the motive, one to generate each case, and one to compose the motive and cases.

$$\boxed{\Gamma \vdash (T_A,\ T_B) \Downarrow_{i_m} t}$$

Index-Motive

$$\frac{}{\Gamma \vdash (A, B) \Downarrow_{i_m} \lambda(\vec{i_A} : \vec{X_A})(a : A\ \vec{i_A}).(I_B\ \vec{i_A})_\beta}$$

$$\boxed{\Gamma \vdash (T_A,\ T_B) \Downarrow_{i_c} t}$$

Index-Conclusion

$$\frac{}{\Gamma \vdash (p_A\ \vec{i_A}\ a,\ p_B\ \vec{i_B}\ b) \Downarrow_{i_c} \vec{i_B}[\text{off}\ A\ B]}$$

Index-Hypothesis

$$\frac{\text{new}\ n_B\ b_B \qquad \Gamma,\ n_B : t_B \vdash (\Pi(n_A : t_A).b_A,\ b_B) \Downarrow_{i_c} t}{\Gamma \vdash (\Pi(n_A : t_A).b_A,\ \Pi(n_B : t_B).b_B) \Downarrow_{i_c} t}$$

Index-IH

$$\frac{\Gamma \vdash (A, B) \Downarrow_{i_m} p \qquad \Gamma,\ n_A : p\ \vec{i_A}\ a \vdash (b_A,\ b_B[n_A/\vec{i_B}[\text{off}\ A\ B]]) \Downarrow_{i_c} t}{\Gamma \vdash (\Pi(n_A : p_A\ \vec{i_A}\ a).b_A,\ \Pi(n_B : p_B\ \vec{i_B}\ b).b_B) \Downarrow_{i_c} \lambda(n_A : p\ \vec{i_A}\ a).t}$$

Index-Prod

$$\frac{\Gamma,\ n_A : t_A \vdash (b_A,\ b_B[n_A/n_B]) \Downarrow_{i_c} t}{\Gamma \vdash (\Pi(n_A : t_A).b_A,\ \Pi(n_B : t_B).b_B) \Downarrow_{i_c} \lambda(n_A : t_A).t}$$

$$\boxed{\Gamma \vdash (T_A,\ T_B) \Downarrow_{i} t}$$

Index-Ind

$$\frac{\Gamma \vdash (A,\ B) \Downarrow_{i_m} p \qquad \Gamma,\ p_A : \mathrm{P}_A,\ p_B : \mathrm{P}_B \vdash \{(\mathrm{E}_{A_1}\ p_A,\ \mathrm{E}_{B_1}\ p_B), \ldots, (\mathrm{E}_{A_n}\ p_A,\ \mathrm{E}_{B_n}\ p_B)\} \Downarrow_{i_c} \vec{f}}{\Gamma \vdash (A,\ B) \Downarrow_{i} \lambda(\vec{i_a} : \vec{X_A})(a : A\ \vec{i_a}).\mathrm{Elim}(a, p)\vec{f}}$$

Figure 4 Identifying the indexer function.

Generating the Motive. The $(T_A,\ T_B) \Downarrow_{i_m} t$ judgment consists of only the derivation Index-Motive, which computes the indexer motive from the types A and B (expanded in Figure 3). It does this by constructing a function with A and its indices as premises, and the type I_B in the conclusion with the appropriate indices. Consider `list` and `vector`:

```
list T := Ind (Ty_A : Type) {...}     vector T := Ind (Ty_B : Π(n :  nat).Type) {...}
```

For these types, Index-Motive computes the motive:

```
λ (l:list T) . nat
```

Generating Each Case. The $\Gamma \vdash (T_A,\ T_B) \Downarrow_{i_c} t$ judgment generates each case of the indexer by traversing in parallel the corresponding cases of the eliminator types for A and B. It consists of four derivations: Index-Conclusion handles base cases and conclusions of inductive cases, while Index-Hypothesis, Index-IH, and Index-Prod recurse into products.

Index-Hypothesis handles each new hypothesis that corresponds to a new index in an inductive hypothesis of an inductive case of the eliminator type for B. It adds the new index to the environment, then recurses into the body of only the type for which the index already exists. For example, in the inductive case of `list` and `vector`, `new` determines that `n` is the new hypothesis. Index-Hypothesis then recurses into the body of only the `vector` case:

```
Π (t_l:T) (l:list T) (IH_l:p_A l), ...     Π (t_v:T) (v:vector T n) (IH_v:p_B n v), ...
```

Index-Prod is next. It recurses into product types when the hypothesis is neither a new index nor an inductive hypothesis. Here, it runs twice, recursing into the body and substituting names until it hits the inductive hypothesis for both types:

```
Π (IH_l:p_A l), p_A (cons t_l l)          Π (IH_v:p_B n l), p_B (S n) (consV n t_l l)
```

$$\boxed{\Gamma \vdash (T_A,\ T_B) \Downarrow_{p_m} t}$$

$$\textsc{Promote-Motive}\ \frac{\Gamma \vdash (A,\ B) \Downarrow_i \pi}{\Gamma \vdash (A,\ B) \Downarrow_{p_m} \lambda(\vec{i_a} : \vec{\mathrm{X}_A})(a : A\ \vec{i_a}).B\ (\mathrm{index}\ (\pi\ \vec{i_a}\ a)\ \vec{i_a})}$$

$$\boxed{\Gamma \vdash (T_A,\ T_B) \Downarrow_{p_c} t}$$

$$\textsc{Promote-Conclusion}\ \frac{}{\Gamma \vdash (p_A\ \vec{i_A}\ a,\ p_B\ \vec{i_B}\ b) \Downarrow_{p_c} b}$$

$$\textsc{Promote-IH}\ \frac{\Gamma \vdash (A,\ B) \Downarrow_i \pi \qquad \Gamma \vdash (A,\ B) \Downarrow_{p_m} p \qquad \Gamma,\ n_A : p\ \vec{i_A}\ a \vdash (b_A,\ b_B[n_A/b][\pi\ \vec{i_A}\ a/\vec{i_B}[\mathrm{off}\ A\ B]]) \Downarrow_{p_c} t}{\Gamma \vdash (\Pi(n_A : p_A\ \vec{i_A}\ a).b_A,\ \Pi(n_B : p_B\ \vec{i_B}\ b).b_B) \Downarrow_{p_c} \lambda(n_A : p\ \vec{i_A}\ a).t}$$

$$\boxed{\Gamma \vdash (T_A,\ T_B) \Downarrow_p t}$$

$$\textsc{Promote-Ind}\ \frac{\Gamma \vdash (A,\ B) \Downarrow_i \pi \qquad \Gamma \vdash (A,\ B) \Downarrow_{p_m} p \qquad \Gamma,\ p_A : \mathrm{P}_A,\ p_B : \mathrm{P}_B \vdash \{(\mathrm{E}_{A_1}\ p_A,\ \mathrm{E}_{B_1}\ p_B), \ldots, (\mathrm{E}_{A_n}\ p_A,\ \mathrm{E}_{B_n}\ p_B)\} \Downarrow_{p_c} \vec{f}}{\Gamma \vdash (A,\ B) \Downarrow_p \lambda(\vec{i_A} : \vec{\mathrm{X}_A})(a : A\ \vec{i_A}).\exists\ (\pi\ \vec{i_A}\ a)\ (\mathrm{Elim}(a, p)\vec{f})}$$

Figure 5 Identifying the promotion function.

Index-IH then takes over. It substitutes the new motive in the inductive hypothesis, then recurses into both bodies, substituting the new inductive hypothesis for the index in the eliminator type for B. Here, it substitutes the new motive for p_A in the type of `IH`$_l$, extends the environment with `IH`$_l$, then substitutes `IH`$_l$ for `n`, so that it recurses on these types:

```
pA (cons tl l)                    pB (S IHl) (consV IHl tl l)
```

Finally, Index-Conclusion computes the conclusion by taking the index of motive p_B at off $A\ B$, here `S IH`$_l$. In total, this produces a function that computes the length of `cons t l`:

```
λ (tl:T) (l:list T) (IHl:(λ (l:list T).nat) l).S IHl
```

Composing the Result. The $\Gamma \vdash (T_A,\ T_B) \Downarrow_i t$ judgment consists of only Index-Ind, which identifies the motive and each case using the other two judgments, then composes the result. In the case of `list` and `vector`, this produces a function that computes the length of a list:

```
λ (l:list T).Elim(l, λ (l:list T).nat)
  {0, λ (tl:T) (l:list T) (IHl:(λ (l:list T).nat) l).S IHl}
```

4.1.2 Searching for Promote and Forget

Figure 5 shows the interesting derivations for the judgment $(T_A,\ T_B) \Downarrow_p t$ that searches for `promote`: Promote-Motive identifies the motive as B with a new index (which it computes using `indexer`, denoted by metavariable π). When Promote-IH recurses, it substitutes the inductive hypothesis for the term rather than for its index, and it substitutes the new index (which it also computes using `indexer`) inside of that term. Promote-Conclusion returns the entire term, rather than its index. Finally, Promote-Ind not only recurses into each case, but also packs the result.

```
↑    {i⃗_a : X⃗_A} := promote i⃗_a.          ↓    {i⃗_b : X⃗_B} := forget i⃗_b.
π_{I_B} {i⃗_a : X⃗_A} := indexer i⃗_a.       ∃_{I_B} {i⃗_b : X⃗_B} (b : B i⃗_b) := ∃ i⃗_b[off] b.
↑_B  := π_r ∘ ↑.                          ↓_A  := ↓   ∘ ∃_{I_B}.
↑_{I_B} := π_l ∘ ↑.                       ↓_{I_B} := π_{I_B} ∘ ↓_A.
```

Figure 6 Common definitions for the core lifting algorithm.

The omitted derivations to search for `forget` are similar, except that the domain and range are switched. Consequentially, `indexer` is never needed; Forget-Motive removes the index rather than inserting it, and Forget-IH no longer substitutes the index. Additionally, Forget-Hypothesis adds the hypothesis for the new index rather than skipping it, and Forget-Ind eliminates over the projection rather than packing the result.

4.1.3 Core Search Algorithm

The core search algorithm produces `indexer`, `promote`, and `forget`, then composes them into a tuple. This tuple is how Devoid represents ornaments internally. Devoid includes an option to generate a proof that these components form the ornamental promotion isomorphism; by default, this is disabled, since `Lift` does not need this proof. The implementation of this option gives intuition for correctness of the search algorithm, and is described in Section 5.3.

4.2 Lift

The `Lift` algorithm implements the specification from Section 3.2. We show only one direction of the algorithm, promoting from A to packed B; the forgetful direction is similar. The core algorithm (Figure 9) builds on a set of common definitions (Figure 6) and two intermediate judgments: one to lift eliminators (Figure 7) and one to lift constructors (Figure 8).

Common Definitions. The common definitions (Figure 6) define some useful syntax: $\uparrow$ applies `promote`, $\downarrow$ applies `forget`, and π_{I_B} applies `indexer`. $\exists_{I_B}$ packs a term of type B into an existential with the index at the appropriate offset. $\uparrow_B$ and $\uparrow_{I_B}$ promote and then project; $\downarrow_A$ packs and forgets, and $\downarrow_{I_B}$ packs, forgets, and then applies `indexer` to project the index.

4.2.1 Lifting Eliminators

The $\Gamma \vdash t \Uparrow_E t'$ judgment (Figure 7) defines rules for lifting the motive and case of an eliminator, changing the *domain of induction* from A to B. The intuition is that any term of type A is the result of forgetting some term of type packed B. Then, since A and B have the same inductive structure, we can lift the eliminator of A to the eliminator of B, and move that forgetfulness *inside of each case*. For example, the following terms are propositionally equal:

```
Elim(↓_A b, p_A){                    Elim(b, λ(n:nat)(v:vector T n).p_A (↓_A v)){
  f_nil,                               f_nil,
  (λ(t_l:T)(l:list T)(IH_l:p_A l).     (λ(n:nat)(t_v:T)(v:vector T n)(IH_v:p_A (↓_A v)).
    f_cons t_l l IH_l)                   f_cons t_v (↓_A v) IH_v)
}                                    }
```

The induction rules implement this transformation. Case lifts a case of the eliminator by first recursively lifting the motive, then using the lifted motive to compute the type of the new case, and then using that type to compute the body of the new case. In the example

$\boxed{\Gamma \vdash (t,\ T) \Uparrow_{E_x} t'}$

$$\textsc{Drop-Index}\ \frac{\text{new } n\ b \qquad \Gamma,\ n : t \vdash (f,\ b) \Uparrow_{E_x} b'}{\Gamma \vdash (f,\ \Pi(n : t).b) \Uparrow_{E_x} \lambda(n : t).b'}$$

$$\textsc{Forget-Arg}\ \frac{\Gamma \vdash \vec{i} : \vec{\mathrm{X}_B} \qquad \Gamma,\ n : B\ \vec{i} \vdash ((f\ (\downarrow_A n))_\beta,\ b) \Uparrow_{E_x} b'}{\Gamma \vdash (f,\ \Pi(n : B\ \vec{i}).b) \Uparrow_{E_x} \lambda(n : B\ \vec{i}).b'}$$

$$\textsc{Arg}\ \frac{\Gamma,\ n : t \vdash ((f\ n)_\beta,\ b) \Uparrow_{E_x} b'}{\Gamma \vdash (f,\ \Pi(n : t).b) \Uparrow_{E_x} \lambda(n : t).b'}$$

$$\textsc{Concl}\ \frac{}{\Gamma \vdash (t,\ p_B\ \vec{y}) \Uparrow_{E_x} t}$$

$\boxed{\Gamma \vdash t \Uparrow_E t'}$

$$\textsc{Motive}\ \frac{\Gamma \vdash p_A : \mathrm{P}_A}{\Gamma \vdash p_A \Uparrow_E \lambda(\vec{i} : \vec{\mathrm{X}_B})(b : B\ \vec{i}).(p_A\ (\text{deindex } \vec{i})\ (\downarrow_A b))_\beta}$$

$$\textsc{Case}\ \frac{\Gamma \vdash p_A : \mathrm{P}_A \qquad \Gamma \vdash f_i : \mathrm{E}_{A_i}\ p_A \qquad \Gamma \vdash p_A \Uparrow_E p_B \qquad \Gamma \vdash (f_i,\ \mathrm{E}_{B_i}\ p_B) \Uparrow_{\mathrm{E_x}} f_i'}{\Gamma \vdash f_i \Uparrow_E f_i'}$$

Figure 7 Lifting eliminators.

$\boxed{\Gamma \vdash t \Uparrow_C t'}$

$$\textsc{Normalize}\ \frac{}{\Gamma \vdash \mathrm{Constr}(j,\ A)\ \vec{x} \Uparrow_C (\uparrow (\mathrm{Constr}(j,\ A)\ \vec{x}))_{\beta\delta\iota}}$$

Figure 8 Lifting constructors.

above, when lifting the inductive case, it first recursively lifts the motive p_A using Motive, which drops the index, packs and forgets the argument of type B, and then β-reduces the result, eliminating references to B. This produces the new motive:

```
λ(n:nat)(v:vector T n).pA (↓A v)
```

which Case then uses to compute the type of the inductive case of the eliminator for B:

```
Π(tv:T)(n:nat)(v:vector T n)(IHv:pA (↓A v)).pA (↓A (consV tv (S n) v))
```

The $\Gamma \vdash (t,\ T) \Uparrow_{E_x} t'$ judgment then uses that type to compute the lifted function body. It computes this in a similar way to Motive, except that there are as many indices to drop and arguments to pack and forget as there are inductive hypotheses, and these do not occur in predictable places, so more rules are involved. This computes the new function:

```
λ(n:nat)(tv:T)(v:vector T n)(IHv:pA (↓A v)).fcons tv (↓A v) IHv
```

4.2.2 Lifting Constructors

The $\Gamma \vdash t \Uparrow_C t'$ judgment (Figure 8) lifts applications of constructors of A to applications of constructors of B. This judgment computes one step of the promotion, leaving the recursive lifting of the arguments to the final algorithm. Using the same types, in the base case:

```
↑ nil ≡βδι ∃ O nilV
```

and in the inductive case:

```
↑ (cons t l) ≡βδι ∃ (S (↑IB l)) (consV (↑IB l) t (↑B l))
```

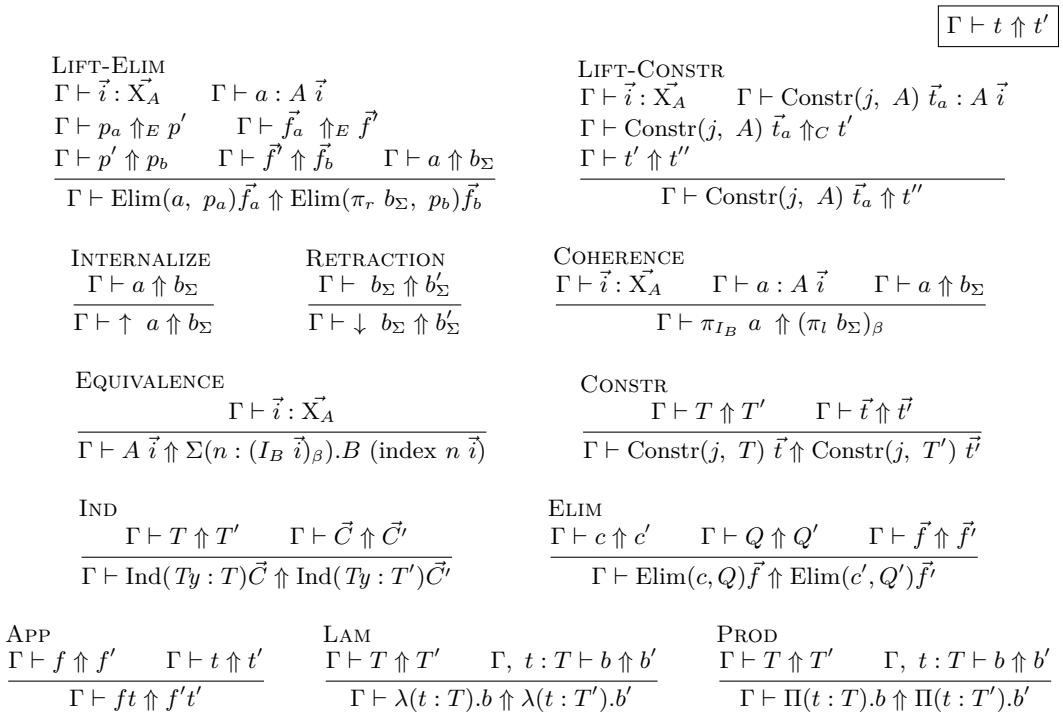


Figure 9 Core lifting algorithm.

This derivation consists of only one rule: Normalize, which normalizes the promotion of the constructor. This is guaranteed to succeed because the application of the constructor is fully η-expanded. The core algorithm later internalizes the promotion functions in the result.

4.2.3 Core Lifting Algorithm

The core algorithm (Figure 9) builds on these intermediate judgments. The interesting derivations for correctness are the first six: Lift-Elim and Lift-Constr use the judgments for lifting eliminators and constructors of A. Internalize internalizes the explicit `promote` functions from the lifted constructors to recursive applications of the algorithm. Retraction and Coherence use the respective properties of the ornamental promotion isomorphism metatheoretically: the first to drop the explicit `forget` functions from the lifted eliminators, and the second to lift the `indexer` to a projection (in the forgetful direction, Section replaces Retraction). Finally, Equivalence lifts A along the equivalence to packed B. The remaining derivations recurse predictably.

5 Implementation

The Devoid Coq plugin implements the algorithms from Section 4; the link to the code is in **Supplement Material**. Devoid cannot produce an ill-typed term, since Coq type checks all terms that plugins produce and rejects ill-typed terms. The implementations of `Find ornament` (`search.ml`) and `Lift` (`lift.ml`) are mostly the same as the algorithms, but with changes to address implementation challenges that scale the algorithms to a Coq tool for proof engineers. This section describes a sample of these changes from each of three categories: addressing differences between Coq and the type theory that the algorithms assume (Section 5.1), optimizing for efficiency (Section 5.2), and improving usability (Section 5.3).

5.1 Addressing Language Differences

Fixpoints. Coq implements eliminators in terms of pattern matching and fixpoints. To handle terms that use these features, Devoid includes a `Preprocess` command that translates these terms into equivalent eliminator applications. This command can preprocess a definition (like `zip` from Section 2) or an entire module (like `List`, as shown in `ListToVect.v`) for lifting. It currently supports fixpoints that are structurally recursive on only immediate substructures. To translate such a fixpoint, it first extracts a motive, then generates each case by partially reducing the function's body under a hypothetical context for the constructor arguments. This is enough to preprocess `List`; Section 8 discusses possible extensions.

Non-Primitive Projections. By default, projections in Coq are non-primitive. That is, this:

```
∀ (T : Type) (v : Σ (n : nat).vector T n), v = ∃ (π_l v) (π_r v).
```

cannot be proven by reflexivity alone (see `Projections.v`). Therefore, Devoid must pack terms like `v` into existentials; otherwise, lifting will sometimes fail. This is why the type of `zip_with_is_zipV_p` in the example from Section 2 packs $\mathtt{v}_1$ and $\mathtt{v}_2$. For the sake of performance and readability of lifted code, Devoid is strategic about when it packs.

Constants. Because Coq has constants, the implementation of Normalize refolds [3] after normalizing. That is, it acts like the `simpl` tactic in Coq, but with special support for sigma types. For example, to lift the `cons` constructor of a list, after normalizing the promotion of `cons t l`, Devoid substitutes the projections of the promotion of `l` for their normal forms, which determines and saves the following fact:

```
∀ {T} (l : list T), ↑ (cons t l) = ∃ (S (↑_{I_B} l)) (consV (↑_{I_B} l) t (↑_B l)).
```

Refolding helps produce more readable lifted code. It also improves lifting performance, since it occurs just once for each constructor.

5.2 Optimizing for Efficiency

Delayed Reduction. When lifting eliminators, Devoid computes a list of arguments and delays reduction. It computes this list backwards, storing the new indices that inductive hypotheses refer to as it recurses. This removes the call to `new` in the premise of Drop-Index.

Lazy η-Expansion. The lifting algorithm assumes that all terms are fully η-expanded. Sometimes, however, η-expansion is not necessary. For efficiency, rather than fully η-expand ahead of time, Devoid η-expands lazily, only when it is necessary for correctness.

Caching. To prevent extra recursion, Devoid caches the outputs of search, as well as lifted constants, inductive types, and constructors. Since these are constants, lookup is low-cost.

5.3 Improving Usability

Correctness Proofs. Devoid has options (used in `Example.v`) that tell search to generate proofs that its outputs are correct, thereby increasing confidence in and usefulness of those outputs. The proof of `coherence` is reflexivity. The intuition behind the automation to prove `section` and `retraction` (`equivalence.ml`) is that `promote` and `forget` map along corresponding constructors, so inductive cases preserve equalities. Thus, each inductive case of these proofs is generated by a fold that rewrites each recursive reference, with reflexivity as identity.

Unpacking. Devoid includes an `Unpack` command (used in `Example.v`) that unpacks packed types in functions and proofs. This way, users may access unpacked terms without writing boilerplate code. For simple functions, this command packs arguments and projects results. It splits higher-order functions into two functions. For proofs that use equality, it applies one lemma convert to dependent equality, and one lemma to deal with non-primitive projections.

User-Friendly Types. `Example.v` describes how the user can recover user-friendly types after unpacking. For example, to recover a function with an output of type `vector T n`, the user lifts a proof that the length of the output of the unlifted `list` version of that function is `n`, then rewrites by that lifted proof. The intuition behind this is that this equivalence holds:

```
{ l : list T & length l = n } ≃ vector T n
```

Recovering a user-friendly type for a proof relating these functions is more complex, since it necessitates reasoning at some point about equalities between equalities. For some index types like `nat`, this follows simply from the fact that the type forms an h-set [32]: all proofs of equality between the same two terms of that type are equal. There is preliminary work on determining a general methodology for deriving user-friendly types for proofs that does not rely on any properties of the index type. The idea is to use the adjunction condition along with the proof of `coherence` by reflexivity; see GitHub issue #39 for the status of this work.

6 Case Study

We used Devoid to automatically discover and lift along ornaments for two scenarios:

1. Single Iteration: from binary trees to sized binary trees
2. Multiple Iterations: from binary trees to binary search trees to AVL trees

For comparison, we also used the ornaments that Devoid discovered to lift functions and proofs using *Equivalences for Free!* [30] (EFF), a more general framework for lifting across equivalences. Devoid produced faster functions and smaller terms, especially when composing multiple iterations of lifting. In addition, Devoid imposed little burden on the user, and the ornaments Devoid discovered proved useful to EFF.

We chose EFF for comparison because Devoid is the only tool for ornaments in Coq, and because doing so demonstrates the benefits of specialized automation for ornaments. Devoid can handle only a small class of equivalences compared to EFF, and it can currently handle only incremental changes to types (one new index at a time). Our experiences suggest that it is possible to use both tools in concert. Section 7 discusses EFF in more detail.

Setup. The case study code is in the `eval` folder of the repository. For each scenario, we ran Devoid to search for an ornament, and then lifted functions and proofs along that ornament using both Devoid and EFF. We noted the amount of user interaction (Section 6.1), as well as the performance of lifted terms (Section 6.2). To test the performance of lifted terms, we tested runtime by taking the median of ten runs using `Time Eval vm_compute` with test values in Coq 8.8.0, and we tested size by normalizing and running `coqwc` on the result.[2]

[2] i5-5300U, at 2.30GHz, 16 GB RAM

In the first scenario, we lifted traversal functions along with proofs that their outputs are permutations of each other from binary trees (`tree`) to sized binary trees (`Sized.tree`). In the second scenario, we lifted the traversal functions to AVL trees (`avl`) through four intermediate types (one for each new index), and we lifted a search function from BSTs (`bst`) to AVL trees through one intermediate type. Both scenarios considered only full binary trees.

To fit `bst` and `avl` into algebraic ornaments for Devoid, we used boolean indices to track invariants. While the resulting types are not the most natural definitions, this scenario demonstrates that it is possible to express interesting changes to structured types as algebraic ornaments, and that lifting across these types in Devoid produces efficient functions.

6.1 User Experience

For each intermediate type in each scenario, we used Devoid to discover the components of the equivalence. These components were enough for Devoid to lift functions and proofs with no additional proof burden and no additional axioms. To use EFF, we also had to prove that these components form an equivalence; we set the appropriate option to generate these proofs using Devoid. In addition, to use EFF, we had to prove univalent parametricity of each inductive type; these proofs were small, but required specialized knowledge. To lift the proof of the theorem `pre_permutes` using EFF, we had to prove the univalent parametric relation between the unlifted and lifted versions of the functions that the theorem referenced; this pulled in the functional extensionality axiom, which was not necessary using Devoid.

In the second scenario, to simulate the incremental workflow Devoid requires, we lifted to each intermediate type, then unpacked the result. For example, the ornament from `bst` to `avl` passed through an intermediate type; we lifted `search` to this type first, unpacked the result, and then repeated this process. In this scenario, using EFF differently could have saved some work relative to Devoid, since with EFF, it is possible to skip the intermediate type;[3] Devoid is best fit where an incremental workflow is desirable.

6.2 Performance

Relative to EFF, Devoid produced faster functions. Table 1 summarizes runtime in the first scenario for `preorder`, and Table 2 summarizes runtime in the second scenario for `preorder` and `search`. The `inorder` and `postorder` functions performed similarly to `preorder`. The functions Devoid produced imposed modest overhead for smaller inputs, but were tens to hundreds of times faster than the functions that EFF produced for larger inputs. This performance gap was more pronounced over multiple iterations of lifting.

Devoid also produced smaller terms: in the first scenario, 13 vs. 25 LOC for `preorder`, 12 vs. 24 LOC for `inorder`, and 17 vs. 29 LOC for `postorder`; and in the second scenario, 21 vs. 120 LOC for `preorder`, 20 vs. 119 LOC for `inorder`, 24 vs. 125 LOC for `postorder`, and 31 vs. 52 LOC for `search`. In the first scenario, the lifted proof of `pre_permutes` using Devoid was 85 LOC; the lifted proof of `pre_permutes` using EFF was 1463184 LOC.

We suspect Devoid provided these performance benefits because it directly lifted induction principles, whereas EFF produced lifted functions in terms of unlifted functions. The multiple iteration case in particular highlights this, since EFF's approach makes lifted terms much slower and larger as the number of iterations increases, while Devoid's approach does not.

[3] The performances of the terms that EFF produces are sensitive to the equivalence used; for a 100 node tree, this alternate workflow produced a search function which is hundreds of times slower and traversal functions which are thousands of times slower than the functions that Devoid produced. In addition, the lifted proof of `pre_permutes` using EFF failed to normalize with a timeout of one hour.

Table 1 Median runtime (ms) of unlifted (`tree`) and lifted (`Sized.tree`) `preorder` over ten runs with test inputs ranging from about 10 to about 10000 nodes.

		10	100	1000	10000	100000
`preorder`	**Unlifted**	0.0	0.0	0.0	3.0 (1.00x)	37.0 (1.00x)
	Devoid	0.0	0.0	0.0	3.0 (1.00x)	35.0 (0.95x)
	EFF	0.0	1.0	27.0	486.5 (162.17x)	8078.5 (218.33x)

Table 2 Median runtime (ms) of unlifted (`tree`) and lifted (`avl`) `preorder`, plus unlifted (`bst`) and lifted (`avl`) `search`, over ten runs with inputs ranging from about 10 to about 100000 nodes.

		10	100	1000	10000	100000
`preorder`	**Unlifted**	0.0	0.0	0.0	3.0 (1.00x)	37.0 (1.00x)
	Devoid	71.5	71.0	69.0	75.0 (25.00x)	109.0 (2.95x)
	EFF	1.0	11.0	152.0	2976.5 (992.17x)	56636.5 (1530.72x)
`search`	**Unlifted**	0.0	0.0	2.0 (1.00x)	3.0 (1.00x)	29.0 (1.00x)
	Devoid	12.0	14.0	12.0 (6.00x)	15.0 (5.00x)	50.0 (1.72x)
	EFF	1.0	5.0	67.0 (33.50x)	1062.0 (354.00x)	15370.5 (530.02x)

7 Related Work

Ornaments. Devoid automates discovery of and lifting across algebraic ornaments in a higher-order dependently typed language. In the decade since the discovery of ornaments [23], there have been a number of formalizations and embedded implementations of ornaments [10, 19, 11, 20, 9]. Devoid is the first tool for ornamentation to operate over a non-embedded dependently typed language. It essentially moves the automation-heavy approach of Ornamentation in ML [33], which operates on non-embedded ML code, into the type theory that forms the basis of theorem provers like Coq. In doing so, it takes advantage of the properties of algebraic ornaments [23]. It also introduces the first search algorithm to identify ornaments, which in the past was identified as a "gap" in the literature [20].

Lifting Proofs. Devoid identifies and lifts proofs along a specific equivalence similar to that from existing ornaments work [20]. The need to automatically lift functions and proofs across equivalences and other relations is a long-standing challenge for proof engineers [22, 1, 21, 16, 34, 6]. The univalence axiom from Homotopy Type Theory [32] enables transparent transport of proofs; cubical type theory [5] gives univalence a constructive interpretation.

Our work is closely related to *Equivalences for Free!* [30], which brings this full circle, using mathematical properties of univalence to enable lifting across equivalences in a substantial subset of CIC_ω without relying on the univalence axiom. In doing so, it introduces and formalizes the relation that our specification depends upon, and implements a framework for lifting in Coq. This framework is more general than Devoid: It lifts along any equivalence, not just ornamental promotions, and can handle opaque terms, with the caveat that users must prove each equivalence themselves; Devoid requires non-opaque terms and lifts along the class of equivalences that correspond to ornamental promotions, taking advantage of the mathematical properties of ornaments to eliminate the need for explicit applications of section and retraction, and to discover and prove certain equivalences automatically. These mathematical properties allow us to automatically lift the induction principle and eliminate references to old terms, which is beneficial for performance.

Similarly, our work is related to CoqEAL [6], which transfers functions along arbitrary relations between types. As these relations do not necessarily need to be equivalences, this framework is more general than our work. Similar tradeoffs between automation and generality apply: CoqEAL produces functions that refer to the old type, and does not yet support automatic inference of relations. In addition, CoqEAL currently only supports automatic transfer of functions, and does not yet handle proofs.

These tools may provide an alternative backend for Devoid. Furthermore, our search algorithm may help discover relations that make these tools easier to use, and our lifting algorithm may help improve automation and efficiency for certain relations in these tools.

Program and Proof Reuse. The problem that we solve is fundamentally about proof reuse, which applies software reuse principles to ITPs. There is a wealth of work in proof reuse, from tactic languages [15] and logical frameworks [4], to tools for proof abstraction and generalization [26, 18], to domain-specific methodologies [12] and frameworks [13].

Devoid focuses on the specific problem of reuse when adding fully-determined indices to types. Other approaches to this problem include combinators which definitionally reduce to desirable terms [14] in the language Cedille, and automatic generation of conversion functions in Ghostbuster [24] for GADTs in Haskell. Our work focuses on a type theory different from both of these, in which the properties that allow for such combinators in Cedille are not present, and in which dependent types introduce challenges not present in Haskell.

Devoid is not the first tool to combine search with reuse. Optician [25] synthesizes bidirectional string transformations; a similar approach may help extend tooling to handle transformations for low-level data. Pumpkin Patch [27] searches the difference in proofs for patches that can be used to repair proofs broken by changes; Devoid uses a similar approach to identify functions that form an equivalence. The resulting tools are complementary: Devoid supports the addition of indices and hypotheses, which Pumpkin Patch does not support; Pumpkin Patch supports changes in values, which Devoid does not support.

8 Conclusions & Future Work

We presented Devoid: a tool for searching for and lifting across algebraic ornaments in Coq. Devoid is the first tool to lift across ornaments in a non-embedded dependently typed language, and to automatically infer certain kinds of ornaments from types alone. Our algorithms give efficient transport across equivalences arising from algebraic ornaments; our case study demonstrates that such automation can make lifted terms smaller and faster as part of an incremental workflow.

Future Work. A future version may support other ornaments beyond algebraic ornaments, with additional user interaction as needed; this may help support, for example, the ornament between `nat` and `list`, where `list` has a new element in the `cons` case. A future version may loosen restrictions on input types to support adding constructors while preserving inductive structure, recursive references under products, and coinductive types. Integrating with Pumpkin Patch [27] may help remove restrictions Devoid makes about the hypotheses of B. `Preprocess` currently supports only certain fixpoints; a more general translation may help Devoid support more terms, and discussions with Coq developers suggest that the implementation of such a translation building on work from the equations [28] plugin is in progress. Extending Devoid to generate proofs of coherence conditions for lifted terms

may increase user confidence. Proofs that the commands that DEVOID implements satisfy their specifications may also increase user confidence. Better automating the recovery of user-friendly types may improve user experience.

References

1 Gilles Barthe and Olivier Pons. Type isomorphisms and proof reuse in dependent type theory. In *International Conference on Foundations of Software Science and Computation Structures*, pages 57–71. Springer, 2001.

2 Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Proofs for free: Parametricity for dependent types. *Journal of Functional Programming*, 22:107–152, March 2012. doi:10.1017/S0956796812000056.

3 Pierre Boutillier. *New tool to compute with inductive in Coq*. Theses, Université Paris-Diderot - Paris VII, February 2014. URL: https://tel.archives-ouvertes.fr/tel-01054723.

4 Joshua E. Caplan and Mehdi T. Harandi. A logical framework for software proof reuse. In *ACM SIGSOFT Software Engineering Notes*, volume 20, pages 106–113. ACM, 1995.

5 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. *arXiv preprint*, 2016. arXiv:1611.02108.

6 Cyril Cohen and Damien Rouhling. A refinement-based approach to large scale reflection for algebra. In *JFLA 2017 - Vingt-huitième Journées Francophones des Langages Applicatifs*, Gourette, France, January 2017. URL: https://hal.inria.fr/hal-01414881.

7 coq-club. [Coq-Club] Dealing with equalities in dependent types. http://sympa.inria.fr/sympa/arc/coq-club/2017-01/msg00099.html, 2017. Accessed: 2019-03-25.

8 coq-club. [Coq-Club] Trouble with dependent induction. http://sympa.inria.fr/sympa/arc/coq-club/2017-12/msg00079.html, 2017. Accessed: 2019-03-25.

9 Pierre-Évariste Dagand. The essence of ornaments. *Journal of Functional Programming*, 27, 2017.

10 Pierre-Evariste Dagand and Conor McBride. A Categorical Treatment of Ornaments. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '13, pages 530–539, Washington, DC, USA, 2013. IEEE Computer Society. doi:10.1109/LICS.2013.60.

11 Pierre-Evariste Dagand and Conor McBride. Transporting functions across ornaments. *Journal of functional programming*, 24(2-3):316–383, 2014.

12 Benjamin Delaware, William Cook, and Don Batory. Product Lines of Theorems. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 595–608, New York, NY, USA, 2011. ACM. doi:10.1145/2048066.2048113.

13 Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à La Carte. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 207–218, New York, NY, USA, 2013. ACM. doi:10.1145/2429069.2429094.

14 Larry Diehl, Denis Firsov, and Aaron Stump. Generic Zero-Cost Reuse for Dependent Types. *CoRR*, abs/1803.08150, 2018. arXiv:1803.08150.

15 Amy Felty and Douglas Howe. Generalization and reuse of tactic proofs. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 1–15. Springer, 1994.

16 Brian Huffman and Ondřej Kunčar. Lifting and Transfer: A modular design for quotients in Isabelle/HOL. In *International Conference on Certified Programs and Proofs*, pages 131–146. Springer, 2013.

17 Inria. The Coq Standard Library. http://coq.inria.fr/distrib/current/stdlib. Accessed: 2019-03-15.

18 Einar Broch Johnsen and Christoph Lüth. Theorem reuse by proof term transformation. In *International Conference on Theorem Proving in Higher Order Logics*, pages 152–167. Springer, 2004.

19 Hsiang-Shang Ko and Jeremy Gibbons. Relational algebraic ornaments. In *Proceedings of the 2013 ACM SIGPLAN workshop on Dependently-typed programming*, pages 37–48. ACM, 2013.

20 Hsiang-Shang Ko and Jeremy Gibbons. Programming with ornaments. *Journal of Functional Programming*, 27, 2016.

21 Nicolas Magaud. Changing data representation within the Coq system. In *International Conference on Theorem Proving in Higher Order Logics*, pages 87–102. Springer, 2003.

22 Nicolas Magaud and Yves Bertot. Changing data structures in type theory: A study of natural numbers. In *International Workshop on Types for Proofs and Programs*, pages 181–196. Springer, 2000.

23 Conor McBride. Ornamental algebras, algebraic ornaments, 2011. URL: `http://plv.mpi-sws.org/plerg/papers/mcbride-ornaments-2up.pdf`.

24 Trevor L. McDonell, Timothy A. K. Zakian, Matteo Cimini, and Ryan R. Newton. Ghostbuster: A Tool for Simplifying and Converting GADTs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 338–350, New York, NY, USA, 2016. ACM. `doi:10.1145/2951913.2951914`.

25 Anders Miltner, Kathleen Fisher, Benjamin C Pierce, David Walker, and Steve Zdancewic. Synthesizing bijective lenses. *Proceedings of the ACM on Programming Languages*, 2(POPL):1, 2017.

26 Olivier Pons. Generalization in type theory based proof assistants. In *International Workshop on Types for Proofs and Programs*, pages 217–232. Springer, 2000.

27 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*, pages 115–129. ACM, 2018.

28 Matthieu Sozeau. Equations: A Dependent Pattern-Matching Compiler. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving*, pages 419–434, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

29 Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. hs-to-coq. `https://github.com/antalsz/hs-to-coq`, 2018-2019. Accessed: 2019-03-12.

30 Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. Equivalences for Free: Univalent Parametricity for Effective Transport. *Proc. ACM Program. Lang.*, 2(ICFP):92:1–92:29, July 2018. `doi:10.1145/3236787`.

31 Amin Timany and Bart Jacobs. First Steps Towards Cumulative Inductive Types in CIC. In *ICTAC*, 2015.

32 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

33 Thomas Williams and Didier Rémy. A Principled Approach to Ornamentation in ML. *Proc. ACM Program. Lang.*, 2(POPL):21:1–21:30, December 2017. `doi:10.1145/3158109`.

34 Theo Zimmermann and Hugo Herbelin. Automatic and transparent transfer of theorems along isomorphisms in the Coq proof assistant. *arXiv preprint*, 2015. `arXiv:1505.05028`.

Verifying That a Compiler Preserves Concurrent Value-Dependent Information-Flow Security

Robert Sison
Data61, CSIRO, Australia
UNSW Sydney, Australia
Robert.Sison@data61.csiro.au

Toby Murray
University of Melbourne, Australia
toby.murray@unimelb.edu.au

Abstract

It is common to prove by reasoning over source code that programs do not leak sensitive data. But doing so leaves a gap between reasoning and reality that can only be filled by accounting for the behaviour of the compiler. This task is complicated when programs enforce *value-dependent* information-flow security properties (in which classification of locations can vary depending on values in other locations) and complicated further when programs exploit shared-variable concurrency.

Prior work has formally defined a notion of concurrency-aware refinement for preserving value-dependent security properties. However, that notion is considerably more complex than standard refinement definitions typically applied in the verification of semantics preservation by compilers. To date it remains unclear whether it can be applied to a realistic compiler, because there exist no general decomposition principles for separating it into smaller, more familiar, proof obligations.

In this work, we provide such a decomposition principle, which we show can almost halve the complexity of proving secure refinement. Further, we demonstrate its applicability to secure compilation, by proving in Isabelle/HOL the preservation of value-dependent security by a proof-of-concept compiler from an imperative While language to a generic RISC-style assembly language, for programs with shared-memory concurrency mediated by locking primitives. Finally, we execute our compiler in Isabelle on a While language model of the Cross Domain Desktop Compositor, demonstrating to our knowledge the first use of a compiler verification result to carry an information-flow security property down to the assembly-level model of a non-trivial concurrent program.

2012 ACM Subject Classification Security and privacy → Logic and verification; Security and privacy → Information flow control; Software and its engineering → Compilers

Keywords and phrases Secure compilation, Information flow security, Concurrency, Verification

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.27

Related Version A full version of the paper is available at `https://arxiv.org/abs/1907.00713`.

Supplement Material The Isabelle/HOL theories are available at `https://covern.org/itp19.html`.

Funding *Robert Sison*: Australian Government RTP Scholarship & Data61 Research Project Award

Acknowledgements We would like to thank our anonymous reviewers, as well as Carroll Morgan, Kai Engelhardt, Gerwin Klein, Christine Rizkallah, Matthew Brecknell, Johannes Åman Pohjola, and Qian Ge, for their very helpful feedback on earlier versions of this paper.

1 Introduction

It is well known that program translations of the kind carried out by compilers can in principle break security properties like confidentiality [12, 2]. Yet source level reasoning about confidentiality remains common [20, 19, 18]. Existing verified compilers like CompCert

$$
\begin{aligned}
&\mathsf{coupling\text{-}inv\text{-}pres}\ \mathcal{B}\ \mathcal{R}\ \mathcal{I}\ \equiv \\
&\forall lc_{1A}\ lc_{1C}.\ (lc_{1A}, lc_{1C}) \in \mathcal{R} \longrightarrow \\
&\quad (\forall lc'_{1C}.\ lc_{1C} \rightsquigarrow_{\mathsf{C}} lc'_{1C} \longrightarrow \\
&\qquad (\exists n\ lc'_{1A}.\ lc_{1A} \rightsquigarrow^{n}_{\mathsf{A}} lc'_{1A}\ \wedge\ (lc'_{1A}, lc'_{1C}) \in \mathcal{R}\ \wedge \\
&\qquad\quad (\forall lc_{2A}\ lc_{2C}\ lc'_{2A}.\ (lc_{1A}, lc_{2A}) \in \mathcal{B} \wedge lc_{1A} =_{\mathsf{mds}} lc_{2A} \wedge \\
&\qquad\qquad (lc_{2A}, lc_{2C}) \in \mathcal{R}\ \wedge\ (lc_{1C}, lc_{2C}) \in \mathcal{I}\ \wedge \\
&\qquad\qquad lc_{1C} =_{\mathsf{mds}} lc_{2C}\ \wedge\ lc_{2A} \rightsquigarrow^{n}_{\mathsf{A}} lc'_{2A}\ \wedge\ lc'_{1A} =_{\mathsf{mds}} lc'_{2A} \\
&\qquad\qquad\quad \longrightarrow (\exists lc'_{2C}.\ lc_{2C} \rightsquigarrow_{\mathsf{C}} lc'_{2C}\ \wedge\ lc'_{1C} =_{\mathsf{mds}} lc'_{2C}\ \wedge \\
&\qquad\qquad\qquad\quad (lc'_{2A}, lc'_{2C}) \in \mathcal{R}\ \wedge\ (lc'_{1C}, lc'_{2C}) \in \mathcal{I}))))
\end{aligned}
$$

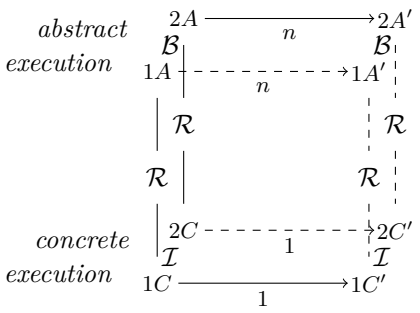


Figure 1 Definition, graphical depiction of refinement preservation for secure-refinement (Def. 6).

[15] and CakeML [14] preserve semantics, but semantics preservation alone may be insufficient to preserve confidentiality, especially for shared memory concurrent programs whose threads must guard against timing leaks in order to prevent them manifesting as storage leaks [22].

Supporting secure compilation of programs that must enforce *value-dependent* security policies poses an additional challenge, because in such policies the sensitivity of a memory location can depend on the values held in other memory locations. Thus, unlike prior work on secure compilation [4], preserving security under refinement requires a refinement relation that is strong enough to preserve those memory contents on which the policy depends.

In prior work [22], we presented a definition for a notion of value-dependent security-preserving refinement that is compositional for concurrent programs: by applying it to each thread individually, one can derive a secure refinement of the concurrent composition.

The essence of this notion of security-preserving refinement (presented fully in Subsection 2.2) is in its refinement preservation obligation (coupling-inv-pres in Figure 1). Here, the usual square-shaped commuting diagram that is commonly used to depict (semantics-preserving) refinement (Figure 4a) has been replaced by a *cube* (Figure 1). The additional dimension of this cube reflects that it preserves a 2-safety hyperproperty [6] that compares two executions rather than examining a single one. As such, it is significantly more complicated to prove than standard notions of semantics-preserving refinement typical in verified compilation [15, 14].

To date there exist no verified compilers for shared-variable concurrent programs proved to preserve value-dependent information-flow security. We argue that without a *decomposition principle* the cube-shaped refinement notion is too cumbersome to prove for realistic compilers.

In this paper, we tackle the central problem of making our notion of secure refinement applicable to verified secure compilation. Firstly, we present a decomposition principle that makes the cube-shaped notion more tractable. Secondly, we demonstrate its tractability with our major contribution: a machine-checked formal proof of concurrent value-dependent security preservation, for a proof-of-concept compiler.

In Section 3 we present our decomposition principle, which decomposes the cube (Figure 1) into three separate obligations (Figure 4). The first of these is akin to semantics-preserving refinement, while the second and third essentially ensure together that the refinement has not introduced any termination- and timing-leaks.

In Section 4 we show how the decomposition principle can almost halve the effort to prove secure refinement – in this case, of a program that is especially prone to introduced timing leaks because it branches on secrets (a feature not yet allowed by our compiler). There, we present a side-by-side comparison of the proof effort, both with and without the decomposition principle. We find that using it reduces the proof's complexity by 44%.

In Section 5, we present our compiler and its formal verification, as an application of the decomposition principle. This compiler translates concurrent programs written in an imperative While language, with locking primitives for mediating access to shared memory, into a RISC-style assembly language. It does so by compiling each thread individually, and in doing so preserves a formal security property that remains compositional between threads. Furthermore, our compiler demonstrates a way of formalising and proving when it is safe for a compiler to perform optimisations in the presence of concurrency. To ensure that the contents of shared memory locations are preserved under compilation despite potential interference from other threads, our compiler tracks which shared memory locations are *stable* (free from any such interference). It then makes use of this tracking to avoid redundant loads from stable shared variables safely, that would otherwise be considered unsafe to omit.

All results are mechanised in Isabelle/HOL,[1] and in Section 6 we explain how, in order to validate our theory, we instantiated it so that we could execute our compiler in Isabelle. This enabled us to execute it over a While language model of the Cross Domain Desktop Compositor [5] (CDDC), a concurrent program that enforces information flow control over value-dependently classified input. To our knowledge this is the first proof of information flow security for an assembly-level model of a non-trivial concurrent program, demonstrating the power of verified secure compilation for deriving security properties of compiled code.

2 Background and example

We begin by introducing with an illustrative example (Figure 2) the challenges of verifying *value-dependent information-flow security* in the presence of *shared-variable concurrency*.

Consider the task of verifying a multithreaded system that manages the user interface (UI) for a *dual-personality smartphone*, a phone that provides clearly distinguished user contexts (*personalities*), typically for work versus leisure. Specifically, our task is to verify that it does not leak *sensitive* information intended only for one of those personalities, which we classify High (Figure 2b), to locations belonging to the other, which we classify Low (Figure 2c).

Here and generally, our *attacker model* is an entity that can read from the system's *untrusted sinks*: some subset of permanently Low-classified locations not subject to synchronisation. In our example, this may include WLAN device registers in a hostile environment.

The smartphone's UI system consists of a number of threads running concurrently with a shared address space, and we aim to verify that as a whole it satisfies the security requirement. But to avoid a state space explosion that is exponential in the number of threads, we must do this *compositionally*: one thread at a time, then combining the results of these analyses.

We focus on a particular worker thread (Figure 2a), the one responsible for sending touchscreen input from the *source* variable to its intended destination.

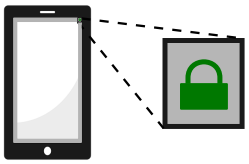
The first challenge is that the destination depends on which personality the phone is currently providing, which is indicated by the value of *domain*. This is reflected by the classification of *source* being dependent on the value of *domain*: *source* is classified Low exactly when $domain = \mathsf{LOW}$ (where LOW is a designated constant), and is classified High otherwise. Due to this dependency, *domain* is known as a *control variable* of *source*.

The second challenge is the worker thread runs in a shared address space that might be accessed or modified by other threads, for various purposes. One of these threads may be responsible for maintaining that $domain = \mathsf{LOW}$ exactly when the phone indicates it is

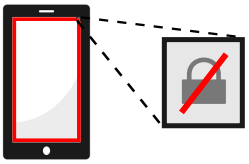
[1] The wr-compiler totals ∼7k lines, and verification + compilation of the 2-thread CDDC model totals ∼1.6k lines of Isabelle proof script, excluding whitespace and comments. See "Supplement Material".

```
while TRUE do
    lock(workspace_lock);
    while !suspended do
        lock(source_lock);
        workspace := source;
        /* ... operations on workspace ... */
        if domain = LOW then
            low_sink := workspace
        else
            high_sink := workspace;
            workspace := 0
        fi;
        unlock(source_lock)
    od;
    unlock(workspace_lock);
    while suspended do skip od
od
```

(a) Input processing worker thread program.



(b) The phone providing the High personality: *domain* ≠ LOW, and *source* is classified High to reflect that the user might type in secrets.



(c) The phone displaying visual indicators that it is providing the Low personality: *domain* = LOW, and *source* is classified Low to reflect that we trust the user not to type in secrets.

Figure 2 Example: Touchscreen input processing for a dual-personality smartphone.

providing the Low personality (Figure 2c), so the user knows not to type in anything sensitive. Another thread may be responsible for assigning *suspended* := TRUE when the user turns the phone's screen off, to make the worker stop processing touchscreen input. We may then wish for *workspace* to be usable by some other thread – e.g. processing input from a fingerprint scanner – in such a way that it can assume *workspace* no longer contains any sensitive values.

When we analyse one thread like this worker in terms of our compositional security property (Subsection 2.1), all of the other threads in the system are trusted to do two things:

1. They follow a *synchronisation scheme*: here, if read- or write-access to a certain variable is governed by a lock, they must hold it in order to access the variable in that manner.

2. They themselves do not leak values from High-classified locations (we refer to such values themselves as High) to Low-classified locations that are read-accessible to other threads. Note we are proving that the thread we are analysing can be trusted in the same way.

Even under these assumptions, the concurrency gives rise to some tricky considerations.

Firstly, it is important that no thread in the system (including the thread under analysis) modifies any control variables carelessly. For example, writing *domain* := LOW immediately after the worker reads a High value from *source*, will cause it to leak to *low_sink*. To prevent this, the worker uses *source_lock*, granting it *exclusive write-access* to *source* and *domain*.

Furthermore as noted above, we may want to ensure that a *non-attacker-observable* location is nevertheless cleared of any sensitive values before being used by another thread. In our example, we classify *workspace* Low for the analysis to enforce this when the worker is suspended, but as the worker sometimes uses it to process High values, it is important to know *workspace* is accessible only to the worker during that time. To ensure this, the worker uses *workspace_lock*, granting it *exclusive read- and write-access* to *workspace*. It is then responsible for clearing it of any High values by the time it releases exclusive read-access.

2.1 Concurrent value-dependent noninterference (CVDNI)

Having illustrated the challenges with an example, we now focus on the formalisation of our information-flow security property CVDNI, which we target with our per-thread analysis, and which our compiler preserves. It is defined in terms of two main elements:

1. a binary *strong low-bisimulation (modulo modes)* relation $\mathcal{B}$ between program configurations, that establishes the required information-flow security property. Like Goguen & Meseguer-style noninterference [10], any states it relates must agree on their "low" portions, and it demands that lock-step execution preserve that correspondence. This section will explain how it is specialised further for shared-variable concurrency.
2. a *classification* function $\mathcal{L}$ that determines the "low" portion of a program configuration, thus affecting $\mathcal{B}$'s requirements. Unlike [10] however, $\mathcal{L}$ here can depend on values in the program configuration itself, thus expressing dynamic and not just static classifications.

We now present definitions from Section III-2b of our previous work [22] simplified as noted. The theory is parameterised over the type of values *Val*, a finite set of shared variables *Var*, and a *deterministic evaluation step semantics* $\rightsquigarrow$ between *local configurations* (of a thread in a concurrent program) each denoted by a triple $\langle tps, mds, mem\rangle$:

- *tps* is the *thread-private state*, which is permanently inaccessible to the attacker and the other threads. Note that due to this inaccessibility, we allow the user of the theory to parameterise the type of *tps*, and do not impose any particular structure.
- $mds :: Mode \Rightarrow Var\ set$ is the *(access) mode state*, which is ghost state associating each $Mode = \{\mathbf{AsmNoW}, \mathbf{AsmNoRW}, \mathbf{GuarNoW}, \mathbf{GuarNoRW}\}$ with a set of shared variables. Intuitively, it identifies the set of variables for which the thread currently possesses (or respects) a kind of exclusivity of access granted (or obligated) by a synchronisation scheme. This facilitates compositional, assume-guarantee [11] style reasoning. For example, when our worker thread holds *source_lock*, it *assumes no other threads write* to *source* or its control variable ($\{source, domain\} \subseteq mds\ \mathbf{AsmNoW}$), otherwise it *guarantees it does not write* to them ($\mathbf{GuarNoW}$). Similarly, holding *workspace_lock* it assumes no other threads *read or write* to *workspace* ($workspace \in mds\ \mathbf{AsmNoRW}$), and at all other times it makes the corresponding guarantee ($\mathbf{GuarNoRW}$).
- $mem :: Mem$ is *shared memory* considered potentially accessible to the attacker and other threads. In order to make what is accessible amenable to analysis, we impose the structure $Mem = Var \Rightarrow Val$, a total map from shared variable names to their values.

The theory is then further parameterised by the value-dependent classification function $\mathcal{L} :: Mem \Rightarrow Var \Rightarrow \{\mathsf{High}, \mathsf{Low}\}$, and a function $\mathcal{C}\mathsf{vars} :: Var \Rightarrow Var\ set$ that returns all the control variables of a given variable. In our worker thread example, $\mathcal{L}\ mem\ x$ gives:

- High when x is *high_sink*, meaning *high_sink* is classified High at all times.
- when x is *source*: Low if $mem\ domain = \mathsf{LOW}$, and High otherwise.
- Low for all other variables x, meaning they are classified Low at all times.

The set $\mathcal{C} = \{y \mid \exists x.\ y \in \mathcal{C}\mathsf{vars}\ x\}$ is then defined to contain all control variables in the system. Thus in our worker thread example, $\mathcal{C}\mathsf{vars}\ source = \{domain\}$ and $\mathcal{C} = \{domain\}$.

To support compositionality for concurrent programs, the "low" portion demanded to be equal by the analysis is tightened up to be *modulo modes* – it includes non-control variables only if they are assumed to be *readable* by other threads according to the mode state: $\mathsf{readable}\ mds\ x \equiv x \notin mds\ \mathbf{AsmNoRW}$. Thus intuitively, the user of the theory should model permanent untrusted output sinks of the whole concurrent program, as variables for which $\mathcal{L}$ *always returns* Low, ungoverned by any synchronisation scheme that the attacker cannot be trusted to follow. (In our example, *low_sink* is untrusted permanently in this way, but *workspace* is untrusted only when unlocked.) The notion of observational indistinguishability used for the noninterference property is then defined over memories as follows.

▶ **Definition 1** (Low-equivalent memories modulo modes).

$$mem_1 =^{\mathsf{Low}}_{mds} mem_2 \equiv$$
$$\forall x.\ x \in \mathcal{C} \ \vee\ \mathcal{L}\ mem_1\ x = \mathsf{Low} \ \wedge\ \mathsf{readable}\ mds\ x \ \longrightarrow\ mem_1\ x = mem_2\ x$$

For this paper, we will use notation $lc_1 =^{\mathsf{Low}}_{\mathsf{mds}} lc_2$ to lift $=^{\mathsf{Low}}_{mds}$ to local program configurations, asserting also that lc_1 and lc_2 are *modes-equal* (have the same mode state). Additionally, we will use notation $lc_1 =_{\mathsf{mds}} lc_2$ to denote (alone) that lc_1 and lc_2 are modes-equal.

The per-thread compositional security property com-secure asserts the existence of a witness relation $\mathcal{B}$ for every possible observationally equivalent pair of starting configurations:

▶ **Definition 2** (Per-thread compositional CVDNI property).

$$\mathsf{com\text{-}secure}\ (tps, mds) \ \equiv\ \forall mem_1\ mem_2.\ mem_1 =^{\mathsf{Low}}_{mds} mem_2 \longrightarrow$$
$$(\exists \mathcal{B}.\ \mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ \mathcal{B}\ \wedge\ (\langle tps, mds, mem_1\rangle, \langle tps, mds, mem_2\rangle) \in \mathcal{B})$$

where all such witness relations $\mathcal{B}$ must be a *strong low-bisimulation (modulo modes)*:

$$\mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ \mathcal{B} \ \equiv\ \mathsf{cg\text{-}consistent}\ \mathcal{B}\ \wedge\ \mathsf{sym}\ \mathcal{B}\ \wedge$$
$$(\forall lc_1\ lc_2.\ (lc_1, lc_2) \in \mathcal{B}\ \wedge\ lc_1 =_{\mathsf{mds}} lc_2 \longrightarrow\ lc_1 =^{\mathsf{Low}}_{\mathsf{mds}} lc_2\ \wedge$$
$$(\forall lc_1'.\ lc_1 \rightsquigarrow lc_1' \longrightarrow\ (\exists lc_2'.\ lc_2 \rightsquigarrow lc_2'\ \wedge\ lc_1' =_{\mathsf{mds}} lc_2'\ \wedge\ (lc_1', lc_2') \in \mathcal{B})))$$

That is, $\mathcal{B}$ must maintain observational indistinguishability by requiring that all configuration pairs it relates that have the same mode state, are low-equivalent modulo modes.

Furthermore, it must be a *bisimulation* by being symmetric and *progressing to itself*: any step taken by one of the configurations must be able to be matched by a step taken by the configuration related to it, such that the destinations remain related by $\mathcal{B}$ (and modes-equal).

Finally – and the most crucial element ensuring the property's compositionality for concurrent programs – is the condition that $\mathcal{B}$ must be cg-consistent: *closed under globally consistent changes* made to memory by other threads, which is to say, changes that preserve low-equivalence and are permitted by the current mode state *mds*. Specifically, the environment (of other threads) is permitted to change either of variable x's value or its classification only when x is *writable*: $\mathsf{writable}\ mds\ x \equiv x \notin mds\ \mathbf{AsmNoW}\ \wedge\ x \notin mds\ \mathbf{AsmNoRW}$.

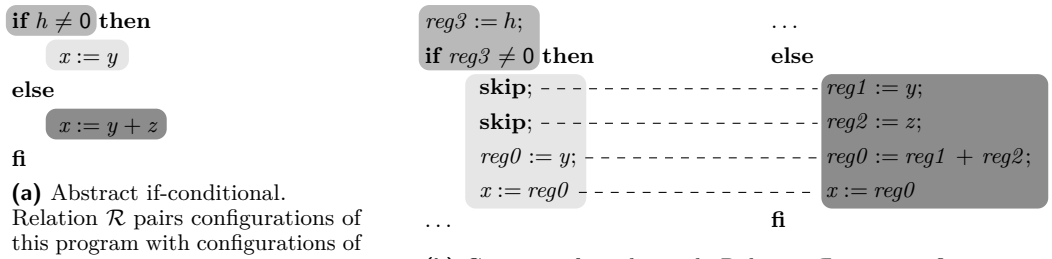
▶ **Definition 3** (Closedness under globally consistent changes).

$$\mathsf{cg\text{-}consistent}\ \mathcal{B} \ \equiv\ \forall tps_1\ mem_1\ tps_2\ mem_2\ mds.$$
$$(\langle tps_1, mds, mem_1\rangle, \langle tps_2, mds, mem_2\rangle) \in \mathcal{B} \ \longrightarrow$$
$$(\forall mem_1'\ mem_2'.\ (\forall x.\ (mem_1\ x \neq mem_1'\ x\ \vee\ mem_2\ x \neq mem_2'\ x\ \vee$$
$$\mathcal{L}\ mem_1\ x \neq \mathcal{L}\ mem_1'\ x) \ \longrightarrow\ \mathsf{writable}\ mds\ x)\ \wedge\ mem_1' =^{\mathsf{Low}}_{mds} mem_2' \ \longrightarrow$$
$$(\langle tps_1, mds, mem_1'\rangle, \langle tps_2, mds, mem_2'\rangle) \in \mathcal{B})$$

Theorem 3.1 of our prior work [22] then gives us that the parallel composition of com-secure programs is itself a program that enforces a system-wide value-dependent noninterference property (sys-secure, for whose details we refer the reader to Section III-2(a) of [22]).

2.2 CVDNI-preserving refinement

Having described the formal security property that we wish to be preserved under refinement (and compilation), we now define formally a suitable notion of secure refinement that preserves it. The proof of CVDNI-preserving refinement for a thread of a concurrent program relies on two binary relations (illustrated by Figure 3) to be nominated by the user of the theory:



(a) Abstract if-conditional. Relation $\mathcal{R}$ pairs configurations of this program with configurations of the program in Figure 3b that are of the same-shaded region.

(b) Concrete if-conditional. Relation $\mathcal{I}$ pairs configurations of this program as shown by the dashed lines.

Figure 3 Excerpts from refinement example [22] that was used to compare proof effort (Section 4).

1. a *refinement relation* $\mathcal{R}$ relating local configurations of the abstract program to local configurations of the concrete program: abstract must simulate concrete, in a sense typical of much other work on program refinement, including compiler verification efforts.
2. a *concrete coupling invariant* $\mathcal{I}$ that allows us to use $\mathcal{B}$ and $\mathcal{R}$ to build a new strong low-bisimulation (modulo modes) for the concrete program, by discarding unreachable pairs of local configurations *after the refinement*. It thereby witnesses that any changes a refinement (or compiler) makes to execution time, do not introduce any timing channels.

The essence of the proof technique is to require that a number of conditions – analogous to those for strong-low-bisim-mm – be imposed on the nominated $\mathcal{R}$ and $\mathcal{I}$ in relation to a given witness relation $\mathcal{B}$ establishing CVDNI for the abstract program. The definitions to follow are adapted from Murray et al. [22] Section V. For better readability, we present a simplified version in which no new shared variables are added by the refinement. Consequently we introduce the notation $=^{\mathsf{mem}}_{\mathsf{mds}}$ to denote that two local configurations have equal mode state and memory, regardless of whether relating configurations of the same or differing languages.

Regarding the maintenance of modes- and observational-equivalence across the relation, the restrictions on refinement are tighter than those that applied to strong-low-bisim-mm. The refinement relation $\mathcal{R}$ is required to preserve the shared memory in its entirety:

▶ **Definition 4** (Preservation of modes and memory).

$$\mathsf{preserves\text{-}modes\text{-}mem}\ \mathcal{R} \equiv \forall lc_A\ lc_C.\ (lc_A, lc_C) \in \mathcal{R} \longrightarrow lc_A =^{\mathsf{mem}}_{\mathsf{mds}} lc_C$$

Regarding the closedness under changes by other threads that ensures compositionality for concurrency, on $\mathcal{I}$ we again impose cg-consistent (Definition 3) from Subsection 2.1. However in the case of $\mathcal{R}$, we instead impose closed-others, a simplification of cg-consistent considering only environmental actions that affect the memories on both sides of the relation identically. Furthermore it ensures equality of *all* shared variables, not just those judged observable:

▶ **Definition 5** (Closedness of refinements under changes by others).

$$\begin{aligned}&\mathsf{closed\text{-}others}\ \mathcal{R} \equiv \forall tps_A\ tps_C\ mds\ mem\ mem'.\\ &\quad (\langle tps_A, mds, mem\rangle_{\mathsf{A}}, \langle tps_C, mds, mem\rangle_{\mathsf{C}}) \in \mathcal{R}) \wedge\\ &\quad (\forall x.\ (mem\ x \neq mem'\ x \vee \mathcal{L}\ mem\ x \neq \mathcal{L}\ mem'\ x) \longrightarrow \mathsf{writable}\ mds\ x) \longrightarrow\\ &\quad (\langle tps_A, mds, mem'\rangle_{\mathsf{A}}, \langle tps_C, mds, mem'\rangle_{\mathsf{C}}) \in \mathcal{R})\end{aligned}$$

The final major requirement for CVDNI-preservation is then to prove $\mathcal{R}$ and $\mathcal{I}$ closed simultaneously under the pairwise executions of the concrete and abstract programs, using the aforementioned cube-shaped diagram (coupling-inv-pres, Figure 1) whose edges are pairs in $\mathcal{B}$, $\mathcal{R}$, and $\mathcal{I}$. All that then remains is for the nominated concrete coupling invariant $\mathcal{I}$ to be symmetric, and the predicate secure-refinement puts together all the requirements:

▶ **Definition 6** (Requirements for secure refinement of the per-thread CVDNI property)**.**

$$\begin{aligned}\mathsf{secure\text{-}refinement}\ \mathcal{B}\ \mathcal{R}\ \mathcal{I}\ \equiv\ &\mathsf{preserves\text{-}modes\text{-}mem}\ \mathcal{R}\ \wedge\ \mathsf{closed\text{-}others}\ \mathcal{R}\ \wedge\\ &\mathsf{cg\text{-}consistent}\ \mathcal{I}\ \wedge\ \mathsf{sym}\ \mathcal{I}\ \wedge\ \mathsf{coupling\text{-}inv\text{-}pres}\ \mathcal{B}\ \mathcal{R}\ \mathcal{I}\end{aligned}$$

Theorem 5.1 of our prior work [22] gives us that under the aforementioned conditions,

$$\begin{aligned}\mathcal{B}_\mathsf{C}\mathsf{of}\ \mathcal{B}\ \mathcal{R}\ \mathcal{I} \equiv \{(lc_{1C}, lc_{2C}) \mid\ &\exists lc_{1A}\ lc_{2A}.\ (lc_{1A}, lc_{1C}) \in \mathcal{R}\ \wedge\ (lc_{2A}, lc_{2C}) \in \mathcal{R}\ \wedge\\ &(lc_{1A}, lc_{2A}) \in \mathcal{B}\ \wedge\ lc_{1C} =^{\mathsf{Low}}_{\mathsf{mds}} lc_{2C}\ \wedge\ (lc_{1C}, lc_{2C}) \in \mathcal{I}\}\end{aligned}$$

is a witness strong-low-bisim-mm for the concrete program:

$$\mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ \mathcal{B} \wedge \mathsf{secure\text{-}refinement}\ \mathcal{B}\ \mathcal{R}\ \mathcal{I} \implies \mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ (\mathcal{B}_\mathsf{C}\mathsf{of}\ \mathcal{B}\ \mathcal{R}\ \mathcal{I})$$

3 Decomposition principle for CVDNI-preserving refinement

Having presented our previous work [22]'s formalisation of our security property CVDNI and its preservation by refinement, we now present our first contribution: an alternative way of proving secure-refinement (Definition 6) that does away with the use of the cube-shaped, two-sided refinement obligation coupling-inv-pres $\mathcal{B}\ \mathcal{R}\ \mathcal{I}$ (depicted by Figure 1), by decomposing its concerns into (1) proving $\mathcal{R}$ closed under the pairwise executions of the concrete and abstract programs alone using a square-shaped diagram (depicted by Figure 4a, which is akin to ordinary semantics-preserving refinement), and (2) a number of smaller and more separable obligations gathered together under the side-condition predicate decomp-refinement-safe.

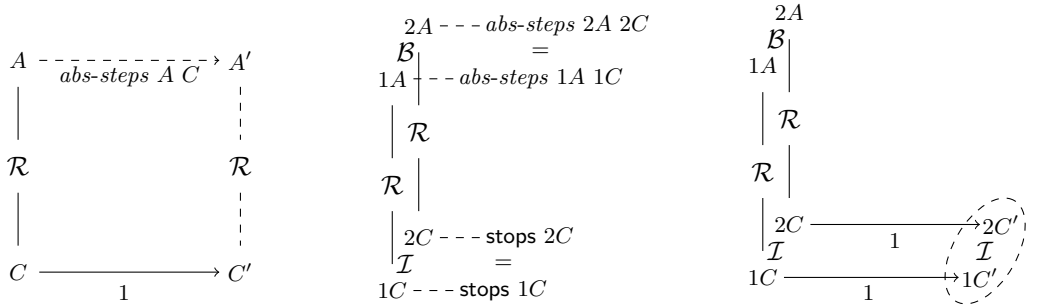
▶ **Definition 7** (Decomposed requirements for CVDNI-preserving secure refinement)**.**

$$\begin{aligned}&\mathsf{secure\text{-}refinement\text{-}decomp}\ \mathcal{B}\ \mathcal{R}\ \mathcal{I}\ \mathit{abs\text{-}steps}\ \equiv\\ &\mathsf{preserves\text{-}modes\text{-}mem}\ \mathcal{R}\ \wedge\ \mathsf{closed\text{-}others}\ \mathcal{R}\ \wedge\ \mathsf{cg\text{-}consistent}\ \mathcal{I}\ \wedge\ \mathsf{sym}\ \mathcal{I}\ \wedge\\ &\mathsf{decomp\text{-}refinement\text{-}safe}\ \mathcal{B}\ \mathcal{R}\ \mathcal{I}\ \mathit{abs\text{-}steps}\ \wedge\ (\forall lc_A\ lc_C.\ (lc_A, lc_C) \in \mathcal{R} \longrightarrow\\ &\quad (\forall lc'_C.\ lc_C \rightsquigarrow_\mathsf{C} lc'_C \longrightarrow\ (\exists lc'_A.\ lc_A \rightsquigarrow_\mathsf{A}^{(\mathit{abs\text{-}steps}\ lc_A\ lc_C)} lc'_A\ \wedge\ (lc'_A, lc'_C) \in \mathcal{R})))\end{aligned}$$

The decomposition requires the provision of a new refinement parameter that we will call *abs-steps* or the *pacing function*, whose role is to dictate the pace of the refinement by returning the number of abstract steps that ought to be taken for a single concrete step, for a given abstract-concrete local configuration pair related by $\mathcal{R}$. The side-conditions on all of the refinement parameters (depicted by Figures 4b, 4c) are then defined as follows:

▶ **Definition 8** (Side-conditions for CVDNI-preserving refinement decomposition)**.**

$$\begin{aligned}&\mathsf{decomp\text{-}refinement\text{-}safe}\ \mathcal{B}\ \mathcal{R}\ \mathcal{I}\ \mathit{abs\text{-}steps}\ \equiv\ \forall lc_{1A}\ lc_{2A}\ lc_{1C}\ lc_{2C}.\ (lc_{1A}, lc_{2A}) \in \mathcal{B} \wedge\\ &lc_{1A} =_\mathsf{mds} lc_{2A} \wedge (lc_{1A}, lc_{1C}) \in \mathcal{R} \wedge (lc_{2A}, lc_{2C}) \in \mathcal{R} \wedge (lc_{1C}, lc_{2C}) \in \mathcal{I} \wedge lc_{1C} =_\mathsf{mds} lc_{2C}\\ &\longrightarrow \mathsf{stops}\ lc_{1C} = \mathsf{stops}\ lc_{2C}\ \wedge\ \mathit{abs\text{-}steps}\ lc_{1A}\ lc_{1C} = \mathit{abs\text{-}steps}\ lc_{2A}\ lc_{2C} \wedge\\ &\quad (\forall lc'_{1C}\ lc'_{2C}.\ lc_{1C} \rightsquigarrow_\mathsf{C} lc'_{1C}\ \wedge\ lc_{2C} \rightsquigarrow_\mathsf{C} lc'_{2C} \longrightarrow\ (lc'_{1C}, lc'_{2C}) \in \mathcal{I} \wedge lc'_{1C} =_\mathsf{mds} lc'_{2C})\end{aligned}$$



(a) Refinement preservation for relation $\mathcal{R}$ under program execution paced by *abs-steps*.

(b) Consistency of pacing and stopping behaviour, to prevent timing and termination leaks.

(c) Closedness of the coupling invariant relation $\mathcal{I}$ under lockstep program execution.

Figure 4 Graphical depictions of refinement decomposition obligations.

On the intuitive meaning of the side-conditions in Definition 8:

- stops lc_{1C} = stops lc_{2C} ensures that the refinement has not introduced any termination leaks, by asserting *consistent stopping behaviour* for $\mathcal{I}$-related concrete program configurations, which we know to be observationally indistinguishable.
- *abs-steps* lc_{1A} lc_{1C} = *abs-steps* lc_{2A} lc_{2C} ensures that the refinement has not introduced any timing leaks, by asserting *consistency of the pace of the refinement* for $\mathcal{R}$-related program configurations, which we again know to be observationally indistinguishable.
- The final $\forall$-quantified clause asserts $\mathcal{I}$'s suitability as a coupling invariant, in that it must remain *closed under lockstep evaluation* of the concrete program configurations it relates. Furthermore it must *maintain mode state equality* with each lockstep evaluation, which ensures that the refinement has not introduced any inconsistencies in the memory access assumptions and guarantees needed for the concurrent compositionality of the property.

Note the $\mathcal{B}$- and $\mathcal{R}$-edges in Figure 4c may capture useful facts about a particular program verification technique and compiler, so their availability as assumptions is intended to reduce greatly the effort needed to specify a coupling invariant $\mathcal{I}$ and prove it satisfies the condition.

Assuming the fulfilment of all of the decomposed requirements, we obtain that they are a sound method for establishing secure refinement of the per-thread CVDNI property:

▶ **Theorem 9** (Soundness of secure-refinement-decomp)**.**

secure-refinement-decomp $\mathcal{B}$ $\mathcal{R}$ $\mathcal{I}$ *abs-steps* $\implies$ secure-refinement $\mathcal{B}$ $\mathcal{R}$ $\mathcal{I}$

In the interests of brevity we relegate proof sketches for all results to the extended version of the paper, and for fuller details we refer the reader to our Isabelle/HOL formalisation.

We now devote our attention to two instantiations of this new decomposition principle: (Section 4) for a proof of CVDNI-preservation for the refinement of a program that branches on a secret, and (Subsection 5.5) for the proof of CVDNI-preservation by a compiler.

4 Proof effort comparison

To demonstrate how the decomposition principle reduces proof complexity and effort, we returned to the example refinement discussed in Section V-E of our previous work [22], an excerpt of which is shown in Figure 3. The abstract program (9 imperative commands) branches on a sensitive value, and executes a single atomic expression assignment in each branch. Its refinement (to 16 commands) models expansion of the expressions into multiple steps, resolving a timing disparity between the two branches by padding with **skip**.

We use proof size as a proxy for proof effort, since the former is known to be strongly linearly correlated with the latter [28]. Formalised in Isabelle/HOL as `EgHighBranchRevC.thy` [21], the proof line count for that theory stood at about 4.6K lines of definitions and proof, of which approx. 3.6K line were proofs. Adapting the proof instead to use the decomposition principle secure-refinement-decomp (Definition 7), the proof line count drops from 3.6K to approx. 2K, a 44% reduction. Regarding definition changes, the new proof makes <10 lines of adaptations to a coupling invariant and pacing function used by the old proof, and adds about 30 lines worth of new helper definitions, for use with the decomposition principle. The rest of the theory and its external dependencies remain in common between the two versions.

As would be expected, the bulk of the deletions are from the full cube-shaped refinement diagram proof (Figure 1) of secure-refinement (Definition 6) for the refinement relation. The surviving parts of that proof just become the square-shaped refinement diagram proof (Figure 4a) of secure-refinement-decomp without much modification. The deletions are replaced by newly added proofs of the three sub-obligations of decomp-refinement-safe (Definition 8).

5 The Covern wr-compiler

Having presented our new decomposition principle for CVDNI-preserving refinement, we now turn to our compiler, whose most notable features for formal proof of secure refinement are:

1. Its implementation tracks variable stability (Subsection 5.4) responsive to use of locking primitives, to know when accesses to shared variables are safe to optimise, and when register contents can be still be considered consistent with shared variable contents.
2. Its verification uses a pacing function (Subsubsection 5.5.2) and coupling invariant (Subsubsection 5.5.3) as the decomposition demands, to ensure it does not introduce timing leaks.

First, we describe its source and target languages, and parameters to the compilation.

5.1 Source language

The Covern wr-compiler – short for *While-to-RISC compiler* – takes the simple imperative language with while-looping and lock-based synchronisation targeted by the Covern program logic [20], which we will refer to as `While`, consisting of the commands *cmd*:

$$
\begin{aligned}
\mathit{exp} &::= n \mid v \mid \mathit{exp} \ \oplus \mathit{exp} \\
\mathit{cmd} &::= \textbf{skip} \mid \mathit{cmd} \,;\, \mathit{cmd} \mid \textbf{if } \mathit{exp} \textbf{ then } \mathit{cmd} \textbf{ else } \mathit{cmd} \textbf{ fi} \mid \\
&\quad\ \ \textbf{while } \mathit{exp} \textbf{ do } \mathit{cmd} \textbf{ od} \mid v := \mathit{exp} \mid \\
&\quad\ \ \textbf{lock}(k) \mid \textbf{unlock}(k)
\end{aligned}
$$

The language is parameterised over a type of values *Val*, and binary operators $\oplus :: \mathit{Val} \Rightarrow \mathit{Val} \Rightarrow \mathit{Val}$. Constants $n :: \mathit{Val}$; $v :: \mathit{Var}$ and $k :: \mathit{Lock}$ are (resp.) shared program- and lock-variables. The semantics of the locking primitives $\textbf{lock}(k)$ and $\textbf{unlock}(k)$ is informed by a locking discipline provided by the user of the theory as a parameter (see Subsection 5.3). We leave for future work adding support for pointers and arrays, which we believe will be straightforward because our assume-guarantee framework already provides the means to encode the memory footprint of a command in a way that depends on values in memory.

We assume that the underlying concurrent execution model (e.g. operating system, scheduler) for the `While` language prevents threads from seeing each others' current program location, and thus (as in previous work [22, 19]) the `While` program command $c :: \mathit{cmd}$ being executed we model as thread-private state: $\langle c, \mathit{mds}, \mathit{mem} \rangle_{\mathtt{w}}$. In contrast, all program variables $v :: \mathit{Var}$ and lock variables $k :: \mathit{Lock}$ reside in the shared memory *mem*.

5.2 Target language

The wr-compiler's target is a generic RISC-style assembly language like that of Tedesco et al. [29] but with lock-based synchronisation primitives added, which we will refer to as `RISC`:

$$
\begin{aligned}
I &::= [l :]B \\
B &::= \mathbf{Load}\ r\ v \mid \mathbf{Store}\ v\ r \mid \mathbf{Jmp}\ l \mid \mathbf{Jz}\ l\ r \mid \mathbf{Nop} \\
&\quad\ \mathbf{MoveK}\ r\ n \mid \mathbf{MoveR}\ r\ r \mid \mathbf{Op}\ \oplus\ r\ r \\
&\quad\ \mathbf{LockAcq}\ k \mid \mathbf{LockRel}\ k
\end{aligned}
$$

The language is parameterised over the same value type Val and binary operators $\oplus$, shared program variables $v :: Var$ and shared lock variables $k :: Lock$ as the `While` language. Presently, direct-addressing **Load** and **Store** instructions (referring to registers $r :: Reg$) are adequate for `RISC` to implement all existing `While` features, and we expect adding indirect addressing to `RISC` to be as straightforward as adding pointer and array support to `While`.

`RISC` program texts P are just lists of binary instructions I, each optionally associated with a label $l :: Lab$. We assume that the underlying concurrency model for the `RISC` language (e.g. OS, scheduler etc.) prevents one thread from reading the program code (instructions) of another,[2] as well as another's registers (including the program counter). Thus, we model the distinguished program counter register's value $pc :: nat$, program text P, and register bank $regs :: Reg \Rightarrow Val$ as thread-private state: $\langle((pc, P), regs), mds, mem\rangle_{\mathsf{r}}$. Apart from this adaptation to our triple format, evaluation semantics follows that of the `RISC` target of [29].

Finally, like Tedesco et al. [29] we generalise over the (user-supplied) register allocation scheme, and assume there are enough registers to service the maximum depth of expressions in the source program. We leave for future work the modelling and analysis of a compiler phase that spills register contents to memory, in order to make this assumption unnecessary.

5.3 Locking discipline

Like the Covern logic [20], we assume that the `While` language program being compiled follows a certain locking discipline, about which the compiler has knowledge, so as to ensure that the `RISC` program it produces follows the same discipline.

The user of the theory provides the details of the locking discipline in the form of a *lock interpretation* parameter: $lock\text{-}interp :: Lock \Rightarrow (Var\ set \times Var\ set)$, which for each lock gives the two non-overlapping sets of program variables over which acquiring the lock grants exclusive permission to write, (resp.) read and write. These permissions are then reflected in the way the semantics of the `While` and `RISC` locking primitives act on the mode state.

Regarding lock interpretations and the way they interact with the user-provided value-dependent classification function $\mathcal{L}$ (see Subsection 2.1), we inherit a few cleanliness conditions from that earlier work [20], chief of which are that lock variables k cannot be control variables, a lock variable k governing access to a program variable v must govern the same kind of access to all of v's control variables, and $\mathcal{L}$ must classify all lock variables as Low.

[2] As is usual for program analyses, we omit any explicit modelling of the microarchitectural state used by superscalar processors (like CPU caches, and state relied on by speculative and out-of-order execution, on whose behaviour attacks like Spectre [13] and Meltdown [16] relied). We argue however that our present assumptions are reasonable under two circumstances: when there is no such state (e.g. on microcontrollers like AVR [7]), or when such state is correctly *partitioned* by the underlying hardware [30] or the OS [8] – if the hardware allows it [9]! In the latter case, our analysis assumes that microarchitectural state footprints are partitioned according to thread (for memory containing program text) and according to classification by $\mathcal{L}$ (for shared memory), and furthermore that each value-dependently classified region is given a distinct partition that is flushed on reclassification.

5.4 Compiler implementation and tracking of shared variable stability

We chose as a starting point the compilation scheme of [29], on the basis of their preserving a noninterference property that like ours exhibits resilience to changes made by an environment – in their case, intended for fault-resilience. Aiming to repurpose that for shared-variable concurrency, we adapted it to Isabelle, implementing it as a primitive recursive function:

$$\begin{aligned}&\mathsf{compile\text{-}cmd} :: \mathit{CompRec} \Rightarrow \mathit{Lab}\ \mathit{option} \Rightarrow \mathit{Lab} \Rightarrow \mathit{cmd} \Rightarrow \\ &\qquad (I \times \mathit{CompRec})\ \mathit{list} \times \mathit{Lab}\ \mathit{option} \times \mathit{Lab} \times \mathit{CompRec} \times \mathit{bool}\end{aligned}$$

where we choose $\mathit{Lab} = \mathit{nat}$ for `RISC` instruction labels, and the *compilation record* type $\mathit{CompRec}$ is bookkeeping maintained by the compiler that we will describe further below.

A typical invocation to compile a `While` program $c :: \mathit{cmd}$ takes the form:

$$(\mathit{PCs}, l', nl', C', \mathit{failed}) = \mathsf{compile\text{-}cmd}\ C\ l\ nl\ c \tag{1}$$

Here, compile-cmd takes an *initial compilation record* C, an optional *entry label* l, and the *next available label* nl, and for the benefit of the next invocation returns an optional *exit label* l' if one is used by the program just compiled, the *new next available label* nl', and a *final compilation record* C'. We leave details of label allocation and its impact on achieving sequential composability for compiled `RISC` programs to the extended version of the paper.

In addition to the output `RISC` program $P :: I\ \mathit{list}$ itself, a call to compile-cmd also outputs every $\mathit{CompRec}$ associated with the state of the program just before executing every instruction in P. These are returned zipped up together with P as the *$\mathit{CompRec}$-annotated `RISC` program* $\mathit{PCs} :: (I \times \mathit{CompRec})\ \mathit{list}$. ($P$ can trivially be recovered as map fst PCs.) Finally, compile-cmd may return True for *failed* to reject the input program, such as when it detects a data race (see below), or if expression depth exceeds the assumed limit (Subsection 5.2).

In the style of the compilation scheme on which it was based [29], the wr-compiler maintains a *register record* $\Phi :: \mathit{reg} \rightharpoonup \mathit{exp}$, i.e. a partial map of registers to expressions on shared variables. In addition to using it to compile away any unnecessary loads from variables in shared memory, we also use it to ensure that an expression calculated by `RISC` in registers is equal to the value of the expression as if it had all been calculated by `While` in one step. This is especially important when writing the result of an expression back to shared memory, because the refinement is required to maintain all shared memory values.

New to the wr-compiler is the responsibility of maintaining an *assumption record*, which it uses primarily to detect and reject programs with data races on shared memory, and to rule out the introduction of any new ones. Each assumption record $\mathcal{S} :: (\mathit{Var}\ \mathit{set} \times \mathit{Var}\ \mathit{set})$ is a pair tracking the set of variables on which (resp.) **AsmNoW**, **AsmNoRW** assumptions are currently active at a given point in the program being compiled. As a secondary concern we also use it to assert that the two sides of any if-conditional branches act consistently on the mode state, and that while-loops restore the original mode state on termination.

A compilation record $C = (\Phi, \mathcal{S}) :: \mathit{CompRec}$ is then just a register/assumption record pair. For readability, we use regrec, asmrec to denote (resp.) a $\mathit{CompRec}$'s fst, snd projections.

To explain how the compilation record is used to rule out data races, and to ensure consistency of expression evaluation between source and target program, firstly we must introduce the concept of *stability* of a variable v according to an assumption record $\mathcal{S}$:

$$\mathsf{var\text{-}stable}\ \mathcal{S}\ v\ \equiv\ v \in (\mathsf{fst}\ \mathcal{S} \cup \mathsf{snd}\ \mathcal{S})\ \wedge\ (\forall v' \in \mathcal{C}\mathsf{vars}\ v.\ v' \in (\mathsf{fst}\ \mathcal{S} \cup \mathsf{snd}\ \mathcal{S}))$$

In short, this means that the variable and all its control variables ($\mathcal{C}$vars v) are recorded as having either of **AsmNoW** or **AsmNoRW** active on them.

For register record entries to be of any help in ensuring consistency of `While` and `RISC` expression evaluation, we exclude expression evaluation on data race-prone variables by lifting the concept of stability to register records. The following predicate asserts internal consistency of the compilation record C created by compile-cmd, in the sense that the register record may only map to expressions that mention variables that are recorded as stable by the assumption record accompanying it. (Here, ran denotes the *range* of a map.)

$$\text{regrec-stable } C \equiv \forall e \in \text{ran } (\text{regrec } C).\ (\forall v \in \text{exp-vars } e.\ \text{var-stable } (\text{asmrec } C)\ v)$$

To ensure that an input `While` program maintains register record stability, we define the predicate no-unstable-exprs $c\ C$ to capture the requirement that a program c, if started with a configuration consistent with compilation record C, will never access a lock-protected variable without holding the relevant lock. (It also checks the secondary, mode-state consistency concerns of the assumption record mentioned earlier.) We implement it as a simple static check carried out by a primitive recursive function on the structure of `While` programs.

Together, regrec-stable and no-unstable-exprs make up the main two requirements of a predicate compile-cmd-input-reqs $C\ l\ nl\ c$ imposed on the input arguments to compile-cmd, which gives us enough information to prove a lemma that compile-cmd only ever outputs stable register records. Full details of these we leave to the extended version of the paper.

5.5 Proof of CVDNI-preserving compilation

Having covered the most significant aspects of the Covern wr-compiler's parameters and machinery, we can now present the refinement relation $\mathcal{R}_{\text{wr}}$ (Subsubsection 5.5.1), pacing function $\text{abs-steps}_{\text{wr}}$ (Subsubsection 5.5.2), and coupling invariant $\mathcal{I}_{\text{wr}}$ (Subsubsection 5.5.3) that we use with our new decomposition principle (of Section 3) to prove that it preserves CVDNI (Subsubsection 5.5.4).

5.5.1 Refinement relation $\mathcal{R}_{\text{wr}}$ and its invariants

Just like our example $\mathcal{R}$ of Figure 3, $\mathcal{R}_{\text{wr}}$ pairs abstract with concrete configurations.

Here, we will focus on $\mathcal{R}_{\text{wr}}$'s most notable characteristics for understanding why it is suitable to describe a CVDNI-preserving compilation.[3] We focus on the case `if_expr` of $\mathcal{R}_{\text{wr}}$, which relates the expression evaluation part of the `While` program **if** e **then** c_1 **else** c_2 **fi**, with the corresponding part (including the conditional jump **Jz** after expression evaluation) of the `RISC` program obtained by running compile-cmd on it. (Variables ignored are in gray.)

▶ **Example 10** (Introduction rule for case `if_expr` of $\mathcal{R}_{\text{wr}}$).

$$\frac{\begin{array}{c} c = \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ fi} \qquad \text{compile-cmd-input-reqs } C\ l\ nl\ c \\ (PCs, l', nl_2, C', \text{False}) = \text{compile-cmd } C\ l\ nl\ c \qquad (P_e, r, C_1, \text{False}) = \text{compile-expr } C\ \varnothing\ l\ e \\ (P_1, l_1, nl_1, C_2, \text{False}) = \text{compile-cmd } C_1\ \text{None } (\text{Suc } (\text{Suc } nl))\ c_1 \qquad pc \leq \text{length } P_e \\ (P_2, l_2, nl_2, C_3, \text{False}) = \text{compile-cmd } C_1\ (\text{Some } nl)\ nl_1\ c_2 \qquad C_{pc} = (\text{map snd } PCs\ !\ pc) \\ \text{compiled-cmd-config-consistent } C_{pc}\ regs\ mds\ mem \qquad \text{regrec-stable } C_{pc} \\ \forall mds'\ mem'\ regs'.\ \text{compiled-cmd-config-consistent } C_1\ regs'\ mds'\ mem' \wedge \text{regrec-stable } C_1 \\ \longrightarrow (\langle\langle c_1, mds', mem'\rangle_{\text{w}}, \langle((0, \text{map fst } P_1), regs'), mds', mem'\rangle_{\text{r}}\rangle \in \mathcal{R}_{\text{wr}} \wedge \\ \langle\langle c_2, mds', mem'\rangle_{\text{w}}, \langle((0, \text{map fst } P_2), regs'), mds', mem'\rangle_{\text{r}}\rangle \in \mathcal{R}_{\text{wr}}) \end{array}}{\langle\langle c, mds, mem\rangle_{\text{w}}, \langle((pc, \text{map fst } PCs), regs), mds, mem\rangle_{\text{r}}\rangle \in \mathcal{R}_{\text{wr}}}$$

[3] We provide an informal description of all of the cases, their purpose, and the invariants they maintain, along with a code listing from compile-cmd relevant to the part that will be presented, in the extended version of our paper. For full details, we refer the reader to the Isabelle formalisation.

This is a fairly typical case of $\mathcal{R}_{\mathsf{wr}}$ in a number of respects:

Firstly, there is a direct reference to the call to compile-cmd for the given `While` program. Secondly, various guards (compiled-cmd-config-consistent introduced below, and regrec-stable defined in Subsection 5.4) are asserted in order to restrict the scope of $\mathcal{R}_{\mathsf{wr}}$ only to consider wellformed local program configurations that line up with the conditions captured by the compilation record. Thirdly, the inductive references to $\mathcal{R}_{\mathsf{wr}}$ for P_1 and P_2, the branches of the conditional *that have not been reached yet*, are quantified over all configurations that obey the guards compiled-cmd-config-consistent and regrec-stable relative to C_1, the initial compilation record for each of the sub-calls to compile-cmd for those sub-programs.

The guard compiled-cmd-config-consistent mentioned above asserts that the compilation record C is consistent with the registers *regs*, memory *mem* and mode state *mds*.

$$\begin{aligned}&\mathsf{compiled\text{-}cmd\text{-}config\text{-}consistent}\ C\ regs\ mds\ mem\ \equiv\\ &\quad(\forall r\ e.\ (\mathsf{regrec}\ C)\ r = \mathsf{Some}\ e \longrightarrow regs\ r = \mathsf{ev}_{\mathsf{exp}}\ mem\ e)\ \wedge\\ &\quad\mathsf{asmrec}\ C = (mds\ \mathbf{AsmNoW},\ mds\ \mathbf{AsmNoRW})\end{aligned}$$

Firstly, for all entries in register record mapping some register r to some expression e, the value held in r of the register bank *regs* must match the value of e if evaluated under memory *mem*. Secondly, the assumption record must consist exactly of the program variables the mode state *mds* says have **AsmNoW**, **AsmNoRW** on them respectively.

As we will see in Theorem 17, compiled-cmd-config-consistent also serves as *initial configuration requirements* for compiled programs: only configurations obeying them may be used to initialise a `RISC` program compiled by the wr-compiler with initial compilation record C.

With $\mathcal{R}_{\mathsf{wr}}$ specified, we then prove the two requirements for secure-refinement-decomp that pertain to $\mathcal{R}_{\mathsf{wr}}$ alone: preserves-modes-mem (Definition 4) and closed-others (Definition 5).

▶ **Lemma 11** ($\mathcal{R}_{\mathsf{wr}}$ preserves modes and memory)**.** preserves-modes-mem $\mathcal{R}_{\mathsf{wr}}$

▶ **Lemma 12** ($\mathcal{R}_{\mathsf{wr}}$ is closed under changes by others)**.** closed-others $\mathcal{R}_{\mathsf{wr}}$

5.5.2 Refinement pacing function abs-steps$_{\mathsf{wr}}$

We now nominate an *abs-steps* function, determining the pace at which `While` programs progress in comparison to the `RISC` programs that they are compiled to by the wr-compiler.

To assist here and elsewhere, we define a primitive recursive helper leftmost-cmd that given a sequence of ;-separated `While` commands, strips all but the first: given $c_1\,;c_2$ it returns leftmost-cmd c_1, and given any other `While` program c it returns c.

Our pacing function abs-steps$_{\mathsf{wr}}$ primarily looks at the form of the `RISC` program instruction about to be executed. The `RISC` instructions are divided into three categories:

- Instructions output by compile-expr: **Load**, **Op**, and **MoveK**. For these, abs-steps$_{\mathsf{wr}}$ returns 1 if the leftmost-cmd of the `While` program is **while** e **do** c **od**, to allow it to step to **if** e **then** $(c\,;$ **while** e **do** c **od**$)$ **else stop fi** concurrently with the first `RISC` step of the compiled expression itself. Otherwise, abs-steps$_{\mathsf{wr}}$ returns 0 to indicate the `While` program standing still while the `RISC` program takes *new* steps to evaluate the expression.
- "Epilogue" steps: **Jmp** and **Nop** when used for control flow at the end of a smaller compiled program in the context of a larger one. For these, abs-steps$_{\mathsf{wr}}$ returns 0.
- All other `RISC` instructions are assumed to proceed at a lockstep pace with the `While` command they were compiled from, and for these abs-steps$_{\mathsf{wr}}$ returns 1.

Having nominated abs-steps$_{\mathsf{wr}}$ and $\mathcal{R}_{\mathsf{wr}}$, we now have the parameters over which we are obliged to prove refinement preservation (Figure 4a) as demanded by secure-refinement-decomp (Definition 7). To this end, we prove firstly (elided to the extended version) that every step of

execution of a RISC program produced by the wr-compiler from a While program, maintains the consistency demanded by compiled-cmd-config-consistent between configurations and compilation records. Also, we must prove a correctness lemma for the expression compiler:

▶ **Lemma 13.** $(PCs, r, C', \mathsf{False}) = \mathsf{compile\text{-}expr}\ C\ A\ l\ e \implies (\mathsf{regrec}\ C')\ r = \mathsf{Some}\ e$

Armed with these facts, we can now prove the main refinement preservation result:

▶ **Lemma 14** ($\mathcal{R}_{\mathsf{wr}}$ is a refinement paced by $\mathsf{abs\text{-}steps}_{\mathsf{wr}}$).

$$\forall lc_w\ lc_r.\ (lc_w, lc_r) \in \mathcal{R}_{\mathsf{wr}} \longrightarrow (\forall lc_r'.\ lc_r \rightsquigarrow_{\mathsf{r}} lc_r' \longrightarrow (\exists lc_w'.\ lc_w \overset{(\mathsf{abs\text{-}steps}_{\mathsf{wr}}\ lc_w\ lc_r)}{\rightsquigarrow_{\mathsf{w}}} lc_w' \ \wedge\ (lc_w', lc_r') \in \mathcal{R}_{\mathsf{wr}}))$$

5.5.3 Concrete coupling invariant $\mathcal{I}_{\mathsf{wr}}$

The next element needed is the concrete coupling invariant $\mathcal{I}_{\mathsf{wr}}$, which we define as follows:

$$\mathcal{I}_{\mathsf{wr}} \equiv \{(\langle(((pc, P), regs), mds, mem\rangle_{\mathsf{r}}, \langle((pc', P'), regs'), mds', mem'\rangle_{\mathsf{r}}) \mid (pc, P) = (pc', P')\}$$

In other words, $\mathcal{I}_{\mathsf{wr}}$ asserts that we only need compare local configurations that are at the same location $pc = pc'$ of the same RISC program $P = P'$. When used in concert with a no-high-branching $\mathcal{B}$ (see Subsubsection 5.5.4), the effect of $\mathcal{I}_{\mathsf{wr}}$ is to ensure that the wr-compiler has not introduced any *new* branching on sensitive values.

5.5.4 Successful compilations are CVDNI-preserving refinements

We are ready to prove preservation. First we qualify that we allow only strong-low-bisim-mm $\mathcal{B}$ that describe only While-programs with no branching on High-classified values, as follows:

$$\mathsf{no\text{-}high\text{-}branching}\ \mathcal{B} \equiv$$
$$\forall c\ c'\ mds\ mem\ mem'.\ (\langle c, mds, mem\rangle_{\mathsf{w}}, \langle c', mds, mem'\rangle_{\mathsf{w}}) \in \mathcal{B} \longrightarrow c = c' \wedge (\forall e\ c_1\ c_2.\ \mathsf{leftmost\text{-}cmd}\ c = \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi} \longrightarrow \mathsf{ev}_{\mathsf{exp}}\ mem\ e = \mathsf{ev}_{\mathsf{exp}}\ mem'\ e)$$

That is, it refuses to relate configurations at different program locations. Furthermore if it is at a conditional branching point, the expression e determining which branch will be taken evaluates to the same boolean value for both configurations' memories. When imposed on a relation that already ensures Low-equivalent memory modulo modes, this effectively disallows any present or past branching on sensitive values. Then, for such programs:

▶ **Lemma 15.**
$$\frac{\mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ \mathcal{B} \qquad \mathsf{no\text{-}high\text{-}branching}\ \mathcal{B}}{\mathsf{secure\text{-}refinement\text{-}decomp}\ \mathcal{B}\ \mathcal{R}_{\mathsf{wr}}\ \mathcal{I}_{\mathsf{wr}}\ \mathsf{abs\text{-}steps}_{\mathsf{wr}}}$$

From this it follows immediately via Theorem 9 that $\mathcal{R}_{\mathsf{wr}}$ with the help of $\mathcal{I}_{\mathsf{wr}}$ describes a CVDNI-preserving refinement for non-High-branching While programs:

▶ **Corollary 16** ($\mathcal{R}_{\mathsf{wr}}$ is a CVDNI-preserving refinement for non-High-branching programs).

$$\mathsf{strong\text{-}low\text{-}bisim\text{-}mm}\ \mathcal{B} \wedge \mathsf{no\text{-}high\text{-}branching}\ \mathcal{B} \implies \mathsf{secure\text{-}refinement}\ \mathcal{B}\ \mathcal{R}_{\mathsf{wr}}\ \mathcal{I}_{\mathsf{wr}}$$

Finally, we prove that successful compilation produces a RISC program related by $\mathcal{R}_{\mathsf{wr}}$ to its input While program, when started with corresponding and reasonable initial configurations:

▶ **Theorem 17** (Successful compilations are refinements in $\mathcal{R}_{\mathsf{wr}}$).

$$\frac{(PCs, l', nl', C', failed) = \mathsf{compile\text{-}cmd}\ C\ l\ nl\ c \qquad \mathsf{compile\text{-}cmd\text{-}input\text{-}reqs}\ C\ l\ nl\ c \qquad failed = \mathsf{False} \qquad \mathsf{compiled\text{-}cmd\text{-}config\text{-}consistent}\ C\ regs\ mds\ mem \qquad P = \mathsf{map\ fst}\ PCs}{(\langle c, mds, mem\rangle_{\mathsf{w}}, \langle((0, P), regs), mds, mem\rangle_{\mathsf{r}}) \in \mathcal{R}_{\mathsf{wr}}}$$

6 Case study: the wr-compiler in action

To test the theory, we instantiated it and applied the wr-compiler to a `While`-language model of the Cross Domain Desktop Compositor [5] (CDDC), a non-trivial concurrent program that facilitates a trusted user's interaction with multiple desktop machines of differing clearance.

The CDDC model to which we applied the compiler is a 2-thread program that was a precursor to the 3-thread model that was verified using the Covern program logic [20].[4] Each of the threads of the CDDC program (together about 150 lines of `While`) we proved satisfy the compositional security property com-secure (Definition 2), using a precursor to the Covern logic that yields CVDNI-witness bisimulations that are non-High-branching.

The resulting compiler is *executable* in Isabelle, meaning that compile-cmd can be executed on the `While` program text for each of the two threads to obtain their compilations (together totalling about 250 `RISC` instructions) using the Isabelle tactic **eval**. The secure compilation theorems (Subsubsection 5.5.4), together with strong-low-bisim-mm preservation and compositionality for com-secure (Theorems 5.1, 3.1 of [22], mentioned in Section 2) then allow us to derive that the compiled program is secure when its threads are run concurrently.

To our knowledge this is the first proof of source-level information-flow security being carried by a verified compiler to an assembly-level model of a non-trivial concurrent program.

7 Related work

The following three works, like ours, focus on compilation preserving a form of noninterference.

Tedesco et al. [29] present a type-directed compilation scheme that preserves a *fault-resilient* noninterference property. The compilation scheme of our wr-compiler was inspired by theirs. Like our com-secure CVDNI security property that wr-compiler preserves, Tedesco et al.'s security property is also *strong bisimulation*-based [27]. But where our property accounts (via mode states) for *controlled interference* by other threads, theirs instead quantifies over all possible interference by the environment with the memory contents. While this simplifies their task of proving that their security property is preserved under compilation – as it need not require the compiler to preserve the contents of memory – it means their security property cannot capture value-dependent noninterference. In contrast, our wr-compiler must obey our secure-refinement notion's requirement that memory contents are preserved.[5]

Barthe et al. [2] consider the problem of preserving *cryptographic constant-time policies*, a class of noninterference properties similar to CVDNI in its explicit consideration for capturing timing-sensitivity. Barthe et al. consider a wider scope of common categories of compile-time optimisations (than those performed by our wr-compiler), and mechanise proofs in Coq that such optimisations preserve various constant-time security properties. The sharing of variables in our setting severely limits the scope of our optimisations, to those that the compiler can perform knowing that a shared variable is stable because it has been locked. At present, our wr-compiler avoids redundant loads during expression compilation, but other optimisations like loop hoisting and constant folding we are yet to implement. Their preservation proof technique, *constant-time simulation* was developed independently to our original cube-shaped

[4] We leave for future work an adaptation of the refinement theory and wr-compiler in order to support the *shared data invariants* added by the Covern logic, required to verify the 3-thread CDDC model.

[5] Consequently, we found and fixed a bug in their expression compiler (acknowledged privately) whereby registers in use were incorrectly reallocated. Expressions like $v + (v + 1)$ were thus compiled incorrectly to programs yielding $(v + 1) + (v + 1)$ instead, causing a violation of memory contents preservation.

secure refinement definition [22]. Like ours, theirs is also a cube-shaped obligation and makes use of a pacing function analogous to our *abs-steps*. Unlike our work here, Barthe et al. do not give a general method for decomposing their cube-shaped simulation diagrams.

Neither of the above consider per-thread compositional compilation of concurrent, shared memory programs, nor value-dependent noninterference policies – the focus of our theory and compiler. Barthe et al. [4] however did aim to preserve noninterference of multithreaded programs by compilation, extending a prior *(security) type-preserving* compilation approach [3]. Their noninterference property however was termination- and timing-*insensitive*, so preventing internal timing leaks relied on the scheduler disallowing certain interleavings between threads. Also, their type-preservation argument was derived from a big-step semantics preservation property for their compiler. Here we instead rely on preservation of a small-step semantics (specifically memory contents), which is necessary for us to preserve value-dependent security under compilation, as well as to avoid imposing non-standard requirements on the scheduler.

Other recent works have improved on *fully abstract compilation* (surveyed [23]) by mapping out the spectrum [1] or developing specific forms [25] of *robust property preservation*, concerned with *robustness* of source program (hyper)properties to concrete *adversarial* contexts. Like Tedesco et al. [29], these works differ from ours in quantifying over a wider range of hostile interference. They also focus prominently on changes to data types, which we do not support. Thus, as a 2-safety hyperproperty quantifying over a lesser range of interference, we expect CVDNI-preservation to be implied by R2HSP (robust 2-hypersafety preservation), but do not expect it to imply any other secure compilation criterion on Abate et al.'s [1] spectrum.

While recently Patrignani and Garg [25] instantiated their *robustly safe compilation* for shared-memory fork-join concurrent programs, it only preserves (1-)safety properties. Previously however, Patrignani et al. [24] proved their *trace-preserving compilation* preserves k-safety hyperproperties [6], including noninterference properties. However, it disallows the removal or addition of trace entries, which would be necessary to change the passage of time as seen in the observable trace events. Thus it excludes optimisations carried out by our compiler (when it permits changes to pacing regulated by *abs-steps*) and studied by the two other works [29, 2] on timing-sensitive security-preserving compilation mentioned above.

Finally, there has been much work on large-scale verified compilation [15, 14] some of which has also treated compilation of shared-memory concurrent programs [17] including taking weak-memory consistency into account [26]. Our work here does not consider the effects of weak-memory models. However, it differs to prior work on verified concurrent compilation, in that it formalises and proves a compiler's ability to use information about the application's locking protocol, to exclude unsafe access to shared variables, and conversely to know when it is safe to allow optimisations that would typically be excluded (see Subsection 5.4).

8 Conclusion

To our knowledge, we have presented the first mechanised verification that a compiler preserves concurrent, value-dependent noninterference. To this end, we provided a general decomposition principle for compositional, secure refinement. Although our compiler is a proof-of-concept targeting simple source and target languages, we nevertheless applied it to produce a verified assembly-level model of the CDDC [5], a non-trivial concurrent program.

This work serves to demonstrate that verified security-preserving compilation for concurrent programs is now within reach, by augmenting traditional proof obligations for verified compilation (e.g. square-shaped semantics preservation) with those specific to security (e.g. absence of termination- and timing-leaks) as depicted in Figure 4. We hope that this work paves the way for future large-scale verified security-preserving compilation efforts.

References

1. Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. Exploring Robust Property Preservation for Secure Compilation. *CoRR*, abs/1807.04603, 2018. arXiv:1807.04603.
2. G. Barthe, B. Grégoire, and V. Laporte. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time". In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 328–343, July 2018.
3. Gilles Barthe, Tamara Rezk, and Amitabh Basu. Security Types Preserving Compilation. *Comput. Lang. Syst. Struct.*, 33(2):35–59, July 2007. doi:10.1016/j.cl.2005.05.002.
4. Gilles Barthe, Tamara Rezk, Alejandro Russo, and Andrei Sabelfeld. Security of Multithreaded Programs by Compilation. *ACM Trans. Inf. Syst. Secur.*, 13(3):21:1–21:32, July 2010. doi:10.1145/1805974.1805977.
5. Mark Beaumont, Jim McCarthy, and Toby Murray. The Cross Domain Desktop Compositor: Using Hardware-Based Video Compositing for a Multi-Level Secure User Interface. In *Annual Computer Security Applications Conference (ACSAC)*, pages 533–545, 2016.
6. Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, September 2010. URL: http://dl.acm.org/citation.cfm?id=1891823.1891830.
7. Florian Dewald, Heiko Mantel, and Alexandra Weber. AVR processors as a platform for language-based security. In *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I*, pages 427–445, 2017. doi:10.1007/978-3-319-66402-6_25.
8. Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time Protection: the Missing OS Abstraction. In *Eurosys19*, Dresden, Germany, March 2019. ACM.
9. Qian Ge, Yuval Yarom, and Gernot Heiser. No Security Without Time Protection: We Need a New Hardware-Software Contract. In *Asia-Pacific Workshop on Systems (APSys)*, Korea, August 2018. ACM SIGOPS.
10. Joseph Goguen and José Meseguer. Security Policies and Security Models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, California, USA, April 1982. IEEE Computer Society.
11. Cliff B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. D.Phil. thesis, University of Oxford, June 1981.
12. Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. When Constant-Time Source Yields Variable-Time Binary: Exploiting Curve25519-donna Built with MSVC 2015. In *Cryptology and Network Security*, pages 573–582, Cham, 2016. Springer International Publishing.
13. Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.
14. Ramana Kumar, Magnus Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 179–191, San Diego, January 2014. ACM Press.
15. Xavier Leroy. A Formally Verified Compiler Back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009. doi:10.1007/s10817-009-9155-4.
16. Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.
17. Andreas Lochbihler. Mechanising a Type-Safe Model of Multithreaded Java with a Verified Compiler. *Journal of Automated Reasoning*, 61(1):243–332, June 2018. doi:10.1007/s10817-018-9452-x.

18 Luísa Lourenço and Luís Caires. Dependent Information Flow Types. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 317–328, Mumbai, India, January 2015. ACM.

19 Heiko Mantel, David Sands, and Henning Sudbrock. Assumptions and Guarantees for Compositional Noninterference. In *IEEE Computer Security Foundations Symposium*, pages 218–232, Cernay-la-Ville, France, June 2011. IEEE.

20 Toby Murray, Robert Sison, and Kai Engelhardt. COVERN: A logic for compositional verification of information flow control. In *European Symposium on Security and Privacy*, London, United Kingdom, April 2018. IEEE.

21 Toby Murray, Robert Sison, Edward Pierzchalski, and Christine Rizkallah. Compositional Security-Preserving Refinement for Concurrent Imperative Programs. *Archive of Formal Proofs*, June 2016. , Formal proof development. URL: `http://isa-afp.org/entries/Dependent_SIFUM_Refinement.shtml`.

22 Toby Murray, Robert Sison, Edward Pierzchalski, and Christine Rizkallah. Compositional Verification and Refinement of Concurrent Value-Dependent Noninterference. In *IEEE Computer Security Foundations Symposium*, pages 417–431, Lisbon, Portugal, June 2016.

23 Marco Patrignani, Amal Ahmed, and Dave Clarke. Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work. *ACM Comput. Surv.*, 51(6):125:1–125:36, February 2019. `doi:10.1145/3280984`.

24 Marco Patrignani and Deepak Garg. Secure Compilation and Hyperproperty Preservation. In *IEEE 30th Computer Security Foundations Symposium, CSF 2017, Santa Barbara, USA, August 21 - 25, 2017*, CSF'17, 2017.

25 Marco Patrignani and Deepak Garg. Robustly Safe Compilation. In *Programming Languages and Systems*, pages 469–498, Cham, 2019. Springer International Publishing.

26 Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. Bridging the Gap Between Programming Languages and Hardware Weak Memory Models. *Proc. ACM Program. Lang.*, 3(POPL):69:1–69:31, January 2019. `doi:10.1145/3290382`.

27 Andrei Sabelfeld and David Sands. Probabilistic Noninterference for Multi-Threaded Programs. In *Proceedings of the 13th IEEE Workshop on Computer Security Foundations*, CSFW '00, pages 200–, Washington, DC, USA, 2000. IEEE Computer Society. URL: `http://dl.acm.org/citation.cfm?id=794200.795151`.

28 Mark Staples, Ross Jeffery, June Andronick, Toby Murray, Gerwin Klein, and Rafal Kolanski. Productivity for Proof Engineering. In *Empirical Software Engineering and Measurement*, page 15, Turin, Italy, September 2014.

29 F. Del Tedesco, D. Sands, and A. Russo. Fault-Resilient Non-interference. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 401–416, June 2016.

30 Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 503–516, New York, NY, USA, 2015. ACM. `doi:10.1145/2694344.2694372`.

Quantitative Continuity and Computable Analysis in Coq

Florian Steinberg
INRIA Saclay, France
https://floriansteinberg.github.io
florian.steinberg@inria.fr

Laurent Théry
INRIA Sophia-Antipolis, France
http://www-sop.inria.fr/marelle/thery.html
Laurent.Thery@inria.fr

Holger Thies
Kyushu University, Japan
http://www.holgerthies.com
thies@inf.kyushu-u.ac.jp

Abstract

We give a number of formal proofs of theorems from the field of computable analysis. Many of our results specify executable algorithms that work on infinite inputs by means of operating on finite approximations and are proven correct in the sense of computable analysis. The development is done in the proof assistant Coq and heavily relies on the Incone library for information theoretic continuity. This library is developed by one of the authors and the results of this paper extend the library. While full executability in a formal development of mathematical statements about real numbers and the like is not a feature that is unique to the Incone library, its original contribution is to adhere to the conventions of computable analysis to provide a general purpose interface for algorithmic reasoning on continuous structures. The paper includes a brief description of the most important concepts of Incone and its sub libraries mf and Metric.

The results that provide complete computational content include that the algebraic operations and the efficient limit operator on the reals are computable, that the countably infinite product of a space with itself is isomorphic to a space of functions, compatibility of the enumeration representation of subsets of natural numbers with the abstract definition of the space of open subsets of the natural numbers, and that continuous realizability implies sequential continuity. We also describe many non-computational results that support the correctness of definitions from the library. These include that the information theoretic notion of continuity used in the library is equivalent to the metric notion of continuity on Baire space, a complete comparison of the different concepts of continuity that arise from metric and represented space structures and the discontinuity of the unrestricted limit operator on the real numbers and the task of selecting an element of a closed subset of the natural numbers.

2012 ACM Subject Classification Mathematics of computing → Continuous functions; Theory of computation → Models of computation; Software and its engineering → Formal methods

Keywords and phrases computable analysis, Coq, contionuous functionals, discontinuity, closed choice on the naturals

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.28

Related Version A full version is available on HAL: https://hal.inria.fr/hal-02088293.

Supplement Material The project page of this paper: https://holgerthies.github.io/continuity

Funding *Florian Steinberg*: Supported by the ANR project FastRelax (ANR-14-CE25-0018-01) of the French National Agency for Research and by EU-MSCA-RISE project 731143 "Computing with Infinite Data" (CID).

Laurent Théry: Supported by the ANR project FastRelax (ANR-14-CE25-0018-01) of the French National Agency for Research.
Holger Thies: Supported by JSPS KAKENHI Grant Number JP18J10407 and by the Japan Society for the Promotion of Science (JSPS), Core-to-Core Program (A. Advanced Research Networks).

Acknowledgements The first and last authors would like to thank Hugo Férée, Akitoshi Kawamura and Matthias Schröder for discussion on the topics of this paper.

1 Introduction

Computable analysis is the theory of computing on continuous structures. Its roots are often cited as going back to Turing's fundamental paper from 1936 in which he introduced his mathematical model of computation later known as Turing machine [62]. Turing's original definitions rely on the binary representation and he adapted them to the ones still used today in his 1937 correction [63] with a pointer to earlier work by Brouwer. The theory of computable functions on the real numbers was further developed in the 1950s by Grzegorczyk and Lacombe in parallel [21, 33]. Later on, Kreitz and Weihrauch extended the theory to apply to more general spaces and introduced the formal framework of representations that is standard today [32, 65, 34]. The basic idea behind computable analysis is fairly simple: To make uncountable structures available to computation, one encodes them by infinitary objects that can still be operated on mechanically. Most commonly infinite strings are used, but more conveniently one may use functions between discrete structures. An example for a reasonable encoding of a real number is a function that provides arbitrarily accurate approximations. To compute functions on the real numbers, one operates on such codes by means of allowing calls to their values. Since the inputs and outputs of such functions can be chosen rational and thus be described by finite means, this leads to a computational model that can properly handle infinite inputs while remaining realistic in the sense of being implementable.

The model used in computable analysis is by far not the only popular model for computing with functional inputs. Alternative approaches use similar access models but assume all inputs to be computable, or deal with functional input by encoding via a Gödel numbering. Many of these models are special cases from the perspective of computable analysis [2, 34]. The former of the two mentioned above should for instance be understood to impose a weaker notion of correctness of algorithms as they are only required to behave appropriately on a countable subset of all possible inputs, namely the computable ones. Yet another take on computation on the real numbers are algebraic approaches like the BSS model [8]. These models have the disadvantage of not providing directly implementable algorithms and the advantage that they closely resemble how numerical analysts proceed in practice: the mathematical proof of correctness of the algorithm underlying an implementation often uses mathematical methods that assume the capability to carry out exact operations on real numbers. For the actual implementation, real variables are substituted with machine numbers so that highly optimized and hardware-supported floating-point operations can be used for fast computations. As machine numbers fail to satisfy basic mathematical properties like associativity, the mathematical proof of correctness of the algorithm does not need to have any direct implications for validity of the values the implementation returns even if everything is done correctly. This problem is well aware to algorithm designers and in applications that demand high reliability, correctness may be recovered in an additional step by estimation of rounding errors. These often lead to laborious computations that are error-prone themselves and quickly become infeasible to do by hand.

Recent advances in formal proofs provide a toolset that can be used to make the loop back from numerical practice to the theory of computation [11, 9, 5, 10]. A popular tool in these works is the classical formalization of real numbers in Coq's standard library. This is because working conservative over this axiomatization in Coq provides capabilities fairly similar to working in the BSS model. The computable analysis community has shown an increase of interest in these developments [41]. Algorithms from computable analysis are notoriously difficult to implement in a way that makes them competitive in terms of speed and memory consumption [7, 30] and thus applications often highlight reliability which naturally goes well with verification. Furthermore, popular methods to overcome the efficiency problems use a toolset similar to that used by the verified numerics community [40].

As a step of bringing formal methods and computable analysis closer together, this paper formulates some more theoretical algorithms in the proof assistant Coq. The produced code is fully executable and proven correct in the sense of computable analysis. Where the real numbers turn up, the axiomatization from Coq's standard library is used. We do not make attempts to make these algorithms competitive in terms of speed or memory usage. For example, we currently use rational numbers for approximating reals and no kind of efficiency can be expected before these are not at least replaced by arbitrary precision floating-point numbers. However, it should be kept in mind that this is possible in principle and we believe our framework to have realistic applications. Indeed, for the formalization we use the Incone library whose long term goal is to provide an environment in which the intersection of formal proofs, computable and numerical analysis can conveniently be investigated in Coq and their merits can be combined in attempts to prove efficient algorithms with practical relevance correct.

1.1 Proofs about continuous structures in Coq and related research

Few if any of our results are mathematically original, but most are known facts from computable analysis. Parts of our development of real numbers has previously been covered by fully constructive developments such as the C-CoRn library. Some of these results are also covered by a smaller project that implemented Cauchy reals to use them and the Mathematical Components library to give a definition of the algebraic real numbers in Coq [13]. To the best of our knowledge most of the rest of our results falls outside of the scope of any other formal development in Coq or in other proof assistants for that matter. We consider these formalizations original to this paper.

As our development heavily relies on the Incone library, we make some effort to describe its central concepts and how they were formalized. We tried to keep the presentation of the background theory from computable analysis close to the formal development in the Incone library. The standard references for computable analysis are [48, 65, 29]. The main topics are also presented in a way somewhat closer to how this paper proceeds in [52, 46, 3]. Due to the page restriction, we had to cut some corners in the presentation of the internals of the Incone library and point to the full version of this article for a more exhaustive treatment [60]. By relying on the toolset that the library provides, most of our proofs went quite smoothly and stayed close to the informal proofs from computable analysis. Where the proofs turned out to be more complicated, this paper includes informal descriptions of the formal proofs and the difficulties encountered. For a more thorough description of the interesting parts of the proofs and some details of the simpler proofs we also point the interested reader to the full version.

The C-CoRn library for constructive analysis is by far the most advanced fully computational Coq development that deals with real numbers [14]. It provides a wide range of results about functions on real numbers and some about operators on function spaces and includes

an exhaustive treatment of metric spaces and uniformly continuous functions between metric spaces [44]. While the mathematical contents that are the topics of the C-CoRn library, this paper and the Incone library are similar, the approach and scope are quite different. The C-CoRn library is inspired by, and roughly follows the development of constructive analysis by Bishop and Bridges [6]. Executability is achieved by restricting to constructive proofs. This constructive focus makes the C-CoRn library and the publications related to it difficult to access for some classically trained mathematicians. The Incone library follows the tradition of computable analysis where computational content is extra information that should follow a mathematical understanding of the structures under consideration. For the formulation of a clean mathematical theory, classical reasoning and well justified axioms may be used where they simplify the proofs and clean up the statement of theorems. It should thus be understood as a complementary approach.

The use of axioms always comes with the danger of introducing inconsistencies. We attempted to minimize their use in many places and only use axioms from Coq's standard library which are commonly used in the Coq community. The parts that involve real numbers rely on the axiomatization of the real numbers as an archimedian ordered field from Coq's standard library. Other axioms that we use fairly often include classical reasoning, functional extensionality and weak choice principles like countable choice or choice principles on countable types, some parts also use proof irrelevance. Throughout the paper we make some effort to discuss where we believe the use of axioms to be essential and why. How much work we put into minimizing the use of axiom depends on the use cases of the results. For instance, there exists a line of lemmas that mostly act as sanity checks for the library and are best understood when interpreted in the sense of category theory (universal properties of products etc.). The parts of these that do not feature computational content were given a lower priority in optimizations for axiom use.

1.2 Realizability, computable analysis and computing on infinite data

In computable analysis the elements of an abstract set X to compute over are encoded over Baire space by use of a partial surjective function from Baire space to X that is called a representation. An element of Baire space that is mapped to $x \in X$ by the representation is considered to provide on demand information about x. The description of real numbers via functions that take rational accuracy requirements and return rational approximations is an example for such a representation. A set with a designated representation is called a represented space and there exist natural notions of what it means for a function between represented spaces to be continuous and to be computable. Both of these notions, and in particular where they diverge, are central points to computable analysis. An informal rule of thumb is that any function that is continuous and whose definition is sufficiently 'natural' is also computable.

The Incone library follows these ideas to provide a formal definition of represented spaces in Coq. However, as implicitly done in the example of real numbers, it adds an additional layer of abstraction where the inputs and outputs of a description need not always be explicitly encoded as natural numbers but are allowed to use any countable and inhabited types. The Incone library includes a definition of continuity of functions between represented spaces and, if Coq's types are interpreted as sets and a classical setting is assumed, the continuity part of computable analysis is captured. If one wants to reason about computability as refinement of continuity, more care has to be taken. For instance, to avoid difficulties with the input and output types, one should guarantee that these types are either finite or there is an effective bijection with the natural numbers. This may be forced

by requiring the construction of Mathematical Components `countType` structure for the input and output types [38]. In presence of this additional information, the Incone library provides tools to capture the notion of computability used in computable analysis in Coq. It provides a way to specify functions on Baire space such that the functions computable in the sense of computable analysis are exactly those that can be instantiated with pure Coq terms, i.e., Coq terms that do not involve any axioms. This construction is compatible with Coq's code-extraction capabilities.

However, the Incone library does not give a formal definition of computability of functions between represented spaces or even Baire space, but only reasons about it on the meta-level. This is due to a reflection problem where checking a term for use of axioms can not be done internally. The additional value of such a definition would be the possibility to give computability-theoretic proofs of incomputability. Such proofs are fairly rare in computable analysis due to the rule of thumb mentioned above: any sufficiently natural function that is incomputable should already be discontinuous. This is in particular true for all instances where we have proven incomputability so far. Once computability theoretic proofs of incomputability move to the center of our attention, a formal definition of computability may be added either using a self-reflection library or more directly by relying on a full formalization of a model of computation [20, 66].

1.3 Structure of the paper and pointers to the main results

All theorems, propositions and lemmas in this paper have been formalized in Coq and were made part of the Incone library. They come with explicit pointers to their name in the library. The statements of the results in the library and in the paper are fairly close. The only notable exception is what was discussed at the end of the last section: whenever the paper claims computability, the formal version proves continuity by explicitly specifying a term that witnesses the continuity and this term is axiom-free as can be checked by the user. Many of the claims that are stated in the plain text, as corollaries or as examples are also supported by formal proofs and occasionally library names are put in brackets after the statement. The identifiers of the exact versions of the mf, Rlzrs, Metric and Incone libraries that this paper refers to can be found in the references [58, 57, 59, 56] or downloaded from the project homepage.

The results whose formalization we consider the main contributions are that the algebraic operations and the efficient limit operator on the reals are computable (Examples 3 and 5), that the countably infinite product is isomorphic to a space of functions (Theorem 7), compatibility of the enumeration representation of subsets of natural numbers with the abstract definition of the space of open subsets of the natural numbers (Theorem 16), and that continuous realizability implies sequential continuity. The previous results are fully algorithmic, but we also describe many non-computational theorems. These include that Incone's information theoretic notion of continuity is equivalent to the metric notion on Baire space (Theorem 14), a complete comparison of the different concepts of continuity that arise from metric and represented space structures (Corollary 9 and Lemma 10) and the discontinuity of the unrestricted limit operator on the real numbers (Example 5) and the task of selecting an element of a closed subset of the natural numbers (Corollary 18).

Coq uses a type-theoretic setting, while the mathematics that we formalize is more commonly formulated over a set theoretic background theory. As is very common in these situations, the paper uses a mix of set-theoretic and type-theoretic notations. In particular we identify subsets of a given type T with functions of type $T \to$ `Prop` and borrow the elementhood notation from set theory, i.e., we write $t \in T$ for $T(t)$. We also use the

mathematical notation for subsets, subset inclusion and partial functions. Finally, we avoid the use of the colon for typing when referring to elementhood of certain function spaces. This is because of the confusing ambiguity in interpretation of function types. For our purposes it is more natural to consider elements of some function spaces as mathematical functions and not elements of a function type. This is because we do not want to restrict to the computable functions only and also expressed through regular use of the functional extensionality axiom and of choice principles to construct elements.

2 Multifunctions and partial operators on Baire space

In computable analysis, the computability and topological structure of Baire space are carried over to more general spaces by means of encodings that are called representations. Before we go into detail about how this is done, this section describes the structure on Baire space that we need. Classically, Baire space is the space of all total functions from natural numbers to natural numbers, i.e., functions of type $\mathbb{N} \to \mathbb{N}$. We more generally refer to any space of the form $\mathbf{Q} \to \mathbf{A}$ as Baire space if $\mathbf{Q}$ and $\mathbf{A}$ are countable and inhabited types. Classically these assumption imply that the types are either finite or bijectively related to the natural numbers. Of course, constructively this is far from true. Indeed, if computability considerations come in, one has to be more careful as the bijections with the natural numbers need not be computable. In the applications considered in this paper, however, the substitution by natural numbers are extremely simple, obviously computable and can even be carried out by hand. The critical reader may therefore replace any occurrence of $\mathbf{Q}, \mathbf{A}$ and their dashed variants in the following by $\mathbb{N}$ and assume that the difference in naming is merely for easy distinction of different in- and outputs and readability.

Computable analysis heavily relies on the theory of continuous partial operators on Baire space. In Coq, functions are always total and to find an appropriate notion of partiality, which is important for a proper treatment of continuity, we first need to discuss how functions can be specified through relations [1, 47, 45, 15]. A multivalued function $F\colon S \rightrightarrows T$ (notation `_ ->> _` in the library) is a function that assigns to each $s\colon S$ a possibly empty subset $F(s)$ of T. While this gives F the type $S \to T \to$ `Prop` and one could identify F with a binary relation, the intuition behind a multivalued function is different as S is treated as input type and T as output type. The **domain** of a multifunction F is given by $\mathrm{dom}(F) := \{s\colon S \mid \exists t\colon T, t \in F(s)\}$ and for $s \in \mathrm{dom}(F)$ the set $F(s)$ should be interpreted as the set of eligible return values. A multivalued function is called **total** if its domain is all of S, and **single-valued** if each of the sets $F(s)$ has at most one element.

A multivalued function can be considered a specification for functions: A function $f\colon S \to T$ fulfills the specification $F\colon S \rightrightarrows T$ if $s \in \mathrm{dom}(F) \implies f(s) \in F(s)$ for all $s\colon S$. In this case we say that f **is a choice for** F (`icf` in the library with notation `_ \is_choice_for _`). The operations on multivalued functions are chosen such that they behave well with respect to the interpretation as specifications. For instance, the **composition** $F \circ G$ of two multivalued functions $G\colon R \rightrightarrows S$ and $F\colon S \rightrightarrows T$ is given by

$$F \circ G(r) := \{t\colon T \mid G(r) \subseteq \mathrm{dom}(F) \wedge \exists s, t \in F(s) \wedge s \in G(r)\}.$$

(notation `_ \o _` in the library). This is an associative operation and the second half, namely $F \circ_R G(r) := \{t\colon T \mid \exists s, t \in F(s) \wedge s \in G(r)\}$, is what is commonly used as composition for relations. The domain condition is a modifier that addresses the difference in interpretations and in particular leads to a loss of the symmetry under exchange of the input and output types. In particular for the multifunction composition it is true that if f is a choice for F and g is a choice for G then $f \circ g$ is a choice for $F \circ G$, which may fail for the relational composition (compare Figure 1a).

There is a very straightforward way to generate multifunctions from functions or partial functions. Namely, for a function $f\colon S \to T$ just use the specification $\texttt{F2MF}f\colon S \rightrightarrows T$ that uniquely determines it, i.e., $\texttt{F2MF}f(s) := \{t\colon T \mid t = f(s)\}$. Clearly, this multifunction is always total and single-valued and assuming that T is not empty each total single-valued multifunction arises in this way (`fun_spec`). This construction can be extended to partial functions by assigning to $g\colon S \to \operatorname{opt} T$ the function $\texttt{PF2MF}g(s) := \{t\colon T \mid g(s) = \operatorname{Some} t\}$, which is still single-valued but need not be total anymore. We are mostly interested in operators on Baire spaces, whose domains are rarely decidable. Coding a partial function as a function to an option type may be understood to indicate that the domain of the function should be decidable and we thus avoid it. Instead, we choose the mathematical notation $g\colon \subseteq S \to T$ for partial functions and in the Incone library they are usually treated as single-valued multifunctions right away. The assignments `F2MF` and `PF2MF` are compatible with the multifunction composition and many other operations.

Note that in contrast to functions, any multifunction can be assigned a reverse multifunction where the input and output is simply switched. All properties of a multifunction have a co-version that requires the same property for the reverse multifunction. Many of the co-properties have nice characterizations for the special case of functions. For instance, a function f is injective if and only if $\texttt{F2MF}f$ is co-single-valued and a partial function f is surjective if and only if $\texttt{PF2MF}f$ is co-total.

An important concept for our purposes is the notion of a tightening (`tight` in the library with notation `_ \tightens _`). For multifunctions $F, G\colon S \rightrightarrows T$ we say that F **tightens** G if it is more restrictive as a specification. That is, if

$$\operatorname{dom}(G) \subseteq \operatorname{dom}(F) \quad \text{and} \quad \forall s \in \operatorname{dom}(G), F(s) \subseteq G(s).$$

Indeed, under appropriate assumptions F tightens G if and only if being a choice for F implies being a choice for G (`icf_tight` and `tight_icf`, also compare Figure 1b). A function f is a choice for a multifunction F if and only if $\texttt{F2MF}f$ tightens F (`icf_spec`) and if $\texttt{PF2MF}f$ tightens F we say that f is a **partial choice** for F. An exhaustive overview over the concepts and notations for multifunctions the MF library provides can be found in the preamble of the `mf.v` file [58].

For the purposes of this paper another construction is important. A multifunction Φ_N of type $S \rightrightarrows T$ can be obtained from a function N of type $\mathbb{N} \times S \to \operatorname{opt} T$ via

$$\Phi_N(s) := \{t\colon T \mid \exists n, N(n, s) = \operatorname{Some} t\}.$$

In the special case where $S = \mathbb{N} = T$ the specification of any partial computable function can be expressed using a primitive recursive function N and this is particularly interesting to us as any primitive recursive function has a definition in Coq that is closed under the global context [43]. The core idea behind why this is true is a version of the Kleene normal-form theorem [55], although there are some technical differences. For a fixed partial computable function, a primitive recursive function N that works can be obtained from any Turing machine computing the function as follows: on input (n, s) return $\operatorname{Some} t$ if the machine on input s terminates within the first n time-steps and returns t and None otherwise. Under the reasonable assumption that any Coq-function is computable we obtain a characterization of the partial computable functions. Thus, the above correspondence can be used to talk about computable functions in Coq at least on a meta-level. A priori, the multifunction Φ_N need neither be total nor single-valued but a single-valued tightening $\Phi_{N'}$ of Φ_N can be obtained.

2.1 Continuity of partial operators between Baire spaces

Fix some types $\mathbf{Q}$, $\mathbf{A}$, $\mathbf{Q}'$ and $\mathbf{A}'$ and set $\mathcal{B} := \mathbf{Q} \to \mathbf{A}$ and $\mathcal{B}' := \mathbf{Q}' \to \mathbf{A}'$. An important class of objects of investigation in computable analysis are computable, or at least continuous partial operators on Baire space, or in our generalized setting of type $F\colon \subseteq \mathcal{B} \to \mathcal{B}'$. One way to produce specifications of such operators is to relativize the Φ assignment from the previous section and assign to a function $M : \mathbb{N} \times \mathcal{B} \times \mathbf{Q}' \to \operatorname{opt} \mathbf{A}'$ the specification $F_M\colon \mathcal{B} \rightrightarrows \mathcal{B}'$ such that

$$\psi \in F_M(\varphi) \quad \iff \quad \forall q'\colon \mathbf{Q}', \exists n\colon \mathbb{N}, M(n, \varphi, q') = \operatorname{Some} \psi(q').$$

(`operator` in the library with notation `\F_( _ )`, compare Example 15). The relativization adds complexity as it can for instance be seen on the example of composition: on the one hand it is easy to realize composition for the Φ assignment, finding a tightening of $F_M \circ F_{M'}$ from M and M' alone, on the other hand,this is problematic : M' allows to produce arbitrary good approximations to a functional input for M, but no information is known about how good this approximation must be for M to return a correct value. Indeed, these approximations are sufficient to obtain correct values of the composition only if F_M is continuous.

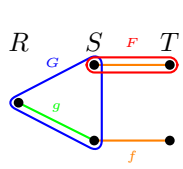
Continuity of F_M can be made sense of by equipping $\mathcal{B}$ and $\mathcal{B}'$ with the topologies of pointwise convergence, or equivalently by using an appropriate metric on these spaces. For our purposes a slightly different, information based description of the same concept is more adequate. Intuitively continuity means that the return-values of an operator $F\colon \mathcal{B} \to \mathcal{B}'$ interpreted as functional of type $F\colon \mathcal{B} \times \mathbf{Q}' \to \mathbf{A}'$ do only depend on finite information about the values of the functional input from $\mathcal{B}$ and thus can be thought of as being represented by a diagram as depicted in Figure 1c. Mathematically, a function $F\colon \mathcal{B} \to \mathcal{B}'$ is **continuous** if for any element φ of $\mathcal{B}$ and any $q'\colon \mathbf{Q}'$ there exists a **certificate**, i.e., a finite list $L\colon \operatorname{seq} \mathbf{Q}$ such that for any ψ that coincides with φ on L it holds that $F(\psi)(q') = F(\varphi)(q')$. Here, two functions are said to **coincide on** a finite list L if $\varphi(q) = \psi(q)$ for any q contained in L. A partial operator $F\colon \subseteq \mathcal{B} \to \mathcal{B}'$ is continuous if for all $\varphi \in \operatorname{dom}(F)$ and $q'\colon \mathbf{Q}'$ there exists a certificate, i.e., a finite list $L \subseteq \mathbf{Q}$ such that the above statement holds for any $\psi \in \operatorname{dom}(F)$.

The definition of continuity in the Incone library follows the mathematical definition given above mostly literally. The only notable difference is that instead of a separate list for each $q'\colon \mathbf{Q}'$ a Skolem-function $\mu\colon \mathbf{Q}' \to \operatorname{seq} \mathbf{Q}$ is used. This is equivalent to the above definition whenever an appropriate choice principle is available (`choice_cont`) and avoids assuming any axioms in the proof that the composition of continuous operators is continuous. From a meta-level many of the proofs of continuity that can be found in the Incone library proceed by specifying an axiom-free Coq-function interpreted either through the `F2MF` or through the $F.$ assignment and may thus be understood as proofs of computability. All claims of computability in the rest of the paper should be understood in this sense.

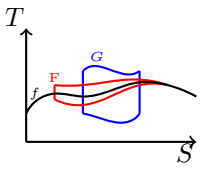
Partiality is treated by using multifunctions and the statement of continuity of a multifunction is chosen in such a way that continuity implies the function to be single-valued (`cont_sing`). This definition works well with the composition of multivalued functions:

▶ **Theorem 1** (`cont_comp`). *Let $F\colon \subseteq \mathcal{B} \to \mathcal{B}'$ and $G\colon \subseteq \mathcal{B}' \to \mathcal{B}''$ be continuous partial operators. The operator $G \circ F\colon \subseteq \mathcal{B} \to \mathcal{B}''$ is continuous.*

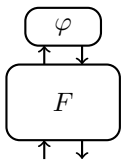
The idea behind the proof is that the certificate functions μ and ν whose existence is guaranteed by the continuity of F and G can be interpreted as multivalued functions and composed relationally to obtain a certificate function for the composition of the operators. The necessary relational composition can be realized constructively.



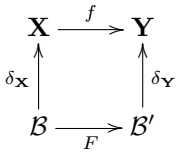
(a) f chooses through F, g chooses through G. $f \circ g$ does not choose through $F \circ_R G$, but through $F \circ G$ which is empty.



(b) f chooses through F which tightens G. Thus f also chooses through G.



(c) A continuous operator F can be thought of as determining its return value by asking finitely many questions to the input function φ.



(d) $F :\subseteq \mathcal{B} \to \mathcal{B}'$ is a realizer of $f : \mathbf{X} \to \mathbf{Y}$.

Figure 1

2.2 Represented spaces and continuous realizability

A **representation** δ of a space X is a partial surjective mapping $\delta :\subseteq \mathcal{B} \to X$. If $\delta(\varphi) = x$ then φ is called a δ-name, or just name, of x. A pair $\mathbf{X} = (X, \delta_{\mathbf{X}})$ of a set and a representation of that set is called a **represented space**. The definition of represented spaces in the Incone library replaces the Baire space $\mathbb{N}^{\mathbb{N}}$ from the definition used in computable analysis with some space $\mathcal{B} = \mathbf{Q} \to \mathbf{A}$, where $\mathbf{Q}$ and $\mathbf{A}$ should be countable inhabited types, i.e., with a Baire space according to the conventions we fixed. Thus, a represented space $\mathbf{X}$ is defined as a record containing a type X (with a coercion from $\mathbf{X}$ to X) together with types $\mathbf{Q}_{\mathbf{X}}$ and $\mathbf{A}_{\mathbf{X}}$ and proofs that these are countable and inhabited and additionally a multivalued function $\delta_{\mathbf{X}} : (\mathbf{Q}_{\mathbf{X}} \to \mathbf{A}_{\mathbf{X}}) \rightrightarrows X$ and proofs that it is single-valued and co-total, where the last requirement is equivalent to being surjective for partial functions. We use the notation $\mathcal{B}_{\mathbf{X}} := \mathbf{Q}_{\mathbf{X}} \to \mathbf{A}_{\mathbf{X}}$.

As an example let us equip the real numbers with the representation that is used for motivation and as a point of reference throughout this section.

▶ **Example 2** (`examples/Q_reals.v`). Choose $\mathbf{Q}_{\mathbb{R}}, \mathbf{A}_{\mathbb{R}} := \mathbb{Q}$, i.e., $\mathcal{B}_{\mathbb{R}} = \mathbb{Q} \to \mathbb{Q}$. It is straight forward to prove that the rational numbers provided by Coq's standard library are countable and inhabited. The multifunction $\delta_{\mathbb{R}} : \mathcal{B}_{\mathbb{R}} \rightrightarrows \mathbb{R}$ (`rep_RQ` in Incone) specified by:

$$\delta_{\mathbb{R}}(\varphi) = x \quad \iff \quad \forall \varepsilon \in \mathbb{Q}, 0 < \varepsilon \implies |x - \varphi(\varepsilon)| < \varepsilon$$

is a representation. Indeed, using the axiomatization of the real numbers provided by Coq's standard library $\delta_{\mathbb{R}}$ can be proven single-valued and surjective and we refer to the represented space $(\mathbb{R}, \delta_{\mathbb{R}})$ (`RQ` in the library) simply by $\mathbb{R}$.

The topological and computability structure of Baire space can be pushed forward through a representation: A partial operator on Baire space is a **realizer** of a function $f : \mathbf{X} \to \mathbf{Y}$ between represented spaces if it assigns to each name of x a name of $f(x)$ (compare Figure 1d). A function between represented spaces is **continuous** if it has a continuous realizer and **computable** if it has a computable realizer. Represented spaces form a Cartesian closed category both with the continuous and the computable functions as morphisms.

With little effort, the definition of being a realizer can be made sense of if both operators on Baire space and functions between represented spaces are multivalued. For the full definitions we point the interested reader to [34] or the Rlzrs library. While we are mostly

interested in continuous, and therefore single-valued, realizers the case where f is multivalued is of interest to us as it is needed for the concrete example of closed choice on the natural numbers that we discuss in Section 3.3. We call a multifunction between represented spaces **continuously realizable** if there exists a continuous realizer that maps any name of an input to a name of some eligible return-value. Multivaluedness can also be used to recover continuity: the sign function on the real numbers is discontinuous, but can be approximated by the family of continuously realizable ε-sign multifunctions whose set of eligible return values is increased to $\{-1, 0, 1\}$ whenever $|x|$ is smaller than ε. Another popular and similar example is the use of an ε-equality test to account for the undecidability of equality on the real numbers. That continuity and continuous realizability are preserved under composition follows from content of the RLZRS library together with the fact that continuity of operators on Baire space is preserved under composition from Theorem 1.

Any Baire space can be made a represented space by using the identity function as a representation. While a partial operation between Baire spaces is continuous if and only if it is continuously realizable with respect to these representations, there are many multivalued functions between Baire spaces that are continuously realizable but not continuous. This is because continuity implies single-valuedness and continuous realizability, to the contrary, is stable under increasing the set of eligible return values. On Baire spaces continuous realizability of a multifunction is equivalent to the existence of a continuous choice function. However, this is specific to Baire spaces and fails for more general represented spaces as can for instance be seen at the example of an ε-sign function or an ε-equality test as given above.

2.3 Basic constructions and examples for represented spaces

Now that we can talk about continuity and computability on the real numbers, a reasonable next step is to attempt to prove addition and multiplication computable.

▶ **Example 3** (`examples/Q_reals.v`)**.** The arithmetic operations on the real numbers are of type $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$ and to make sense of continuity of functions of this type we need to specify a represented space structure on $\mathbb{R} \times \mathbb{R}$. The INCONE library automatically generates a represented space $\mathbf{X} \times \mathbf{Y}$ from arbitrary represented spaces $\mathbf{X}$ and $\mathbf{Y}$ and proves correctness of this construction and continuity of the basic functions. Relying on this one can prove that addition and multiplication of real numbers is continuous (`Rplus_cont` and `Rmult_cont`). The realizers are defined directly using the `F2MF` assignment and are computable (in the sense described in Section 2.1).

Let I be a countable inhabited type and let $\mathbf{X}$ be a represented space. Consider the set of families $(x_i)_{i \in I}$ indexed over I. A reasonable description of such a family would be a function that takes an additional argument from I and if this argument is fixed to i results in a name for x_i. Formally this can be captured by defining a represented space $\prod_I \mathbf{X}$, where the underlying set are the functions of type $I \to X$, the questions given by $\mathbf{Q}_{\prod_I \mathbf{X}} := I \times \mathbf{Q}_{\mathbf{X}}$, the answers by $\mathbf{A}_{\prod_I \mathbf{X}} := \mathbf{A}_{\mathbf{X}}$ and using the representation

$$(x_i) \in \delta_{\prod_I \mathbf{X}}(\varphi) \quad \iff \quad \forall i \colon I, x_i \in \delta_{\mathbf{X}}(q \mapsto \varphi(i, q)),$$

where (x_i) is short for the function $i \mapsto x_i$.

For the understanding of the following proposition recall that the universal property that is required from an infinite product $\prod_I \mathbf{X}_i$ is that for each represented space $\mathbf{Y}$ and family (f_i) of continuous functions $f_i \colon \mathbf{Y} \to \mathbf{X}_i$ there exists a unique continuous function $F \colon \mathbf{Y} \to \prod_I \mathbf{X}_i$ such that for all $i \in I$ and $y \in \mathbf{Y}$ it holds that $F(y)_i = f_i(y)$. The following proposition says that in the special case where all the spaces $\mathbf{X}_i$ coincide and $I = \mathbb{N}$, the space constructed above has this universal property.

▶ **Proposition 4** (`rep_Iprod_sing`, `rep_Iprod_sur` and `cprd_uprp_cont`). *$\prod_I \mathbf{X}$ is a represented space and $\mathbf{X}^\omega := \prod_{\mathbb{N}} \mathbf{X}$ is a countably infinite product in the category of represented spaces and continuous functions.*

The use of the symbol ω instead of $\mathbb{N}$ is to differentiate the space $\mathbf{X}^\omega$ of sequences (notation `_ \^w` in the library) from a function space. The proof of single-valuedness of the infinite product representation assumes functional extensionality and the proof of surjectivity needs a choice principle over the index set I. Since $I = \mathbb{N}$ is by far the most common use-case and I is assumed countable, this will usually boil down to the axiom of countable choice. The proof of the universal property relies on stronger choice principles, classical reasoning and proof irrelevance. Since the category of represented spaces with computable functions fails to have countably infinite products, the universal property should not be provable without axioms. This makes this result more of a sanity result than something that may actually be of use, and minimizing the strength of the axioms used is not our highest priority.

The limit operator is a good example of a multi-function whose natural source space is the space of sequences. Consider the multivalued function $\lim_{\mathbf{X}} \colon \mathbf{X}^\omega \rightrightarrows \mathbf{X}$ where $x \in \lim_{\mathbf{X}}(x_n)$ if and only if there is a convergent sequence of names $(\varphi_n) \subseteq \mathcal{B}_{\mathbf{X}}$ and some φ such that φ is a name of x, each φ_n is a name for x_n and the sequence (φ_n) converges to φ in $\mathcal{B}_{\mathbf{X}}$, i.e., $\lim_{\mathcal{B}_{\mathbf{X}}}(\varphi_n) = \varphi$ where $\mathcal{B}_{\mathbf{X}}$ is given the topology of pointwise convergence of functions between discrete spaces. A function $f \colon \mathbf{X} \to \mathbf{Y}$ between represented spaces is called sequentially continuous if it preserves this notion of a limit, i.e., if $\lim_{\mathbf{X}} x_n = x$ implies that $\lim_{\mathbf{Y}} f(x_n) = f(x)$. While the limit operator on Baire space is single-valued, this may not be true for the limit operator on a general represented space, as can be seen at the example of Sierpiński space that is discussed in Section 3.3. In most spaces that are relevant for numerical analysis, the limit operator is single-valued but discontinuous and has an appropriate computable restriction.

▶ **Example 5** (`examples/Q_reals.v`). On the real numbers $\mathbb{R}$ the limit operator $\lim_{\mathbb{R}}$ captures the usual notion of convergence of sequences of real numbers. Furthermore, $\lim_{\mathbb{R}}$ is discontinuous (`lim_not_cont`), but its restriction to those sequences (x_n) that are efficiently Cauchy in that $|x_n - x_m| \leq 2^{-n} + 2^{-m}$ is computable (`lim_eff_hcr`).

The Incone library defines and proves correct a continuous universal U (one may think of either Kleene-Kreisel associateship [28, 31, 17] or Weihrauch's η [65]). Let $\mathbf{X}$ and $\mathbf{Y}$ be represented spaces and consider the collection of all continuously realizable functions from $\mathbf{X}$ to $\mathbf{Y}$. In Incone, the continuous universal U is used to construct a representation for this collection of functions by

$$f \in \delta_{\mathbf{Y}\mathbf{x}}(\psi) \quad \iff \quad F_{U(\psi)} \text{ realizes } f.$$

Since functions (as opposed to partial or multifunctions) are uniquely determined by each of their realizers, $\delta_{\mathbf{Y}\mathbf{x}}$ is single-valued. That $\delta_{\mathbf{Y}\mathbf{x}}$ is co-total is the distinguishing property of continuous universals like U. Thus, $\delta_{\mathbf{Y}\mathbf{x}}$ is a representation and we refer to the represented space of continuously realizable functions from $\mathbf{X}$ to $\mathbf{Y}$ with this representation as $\mathbf{Y}^{\mathbf{X}}$ (notation `_ c-> _` in Incone).

Recall that spaces are called isomorphic, in symbols $\mathbf{X} \simeq \mathbf{Y}$, if they are connected by a continuous bijection with continuous inverse and computably isomorphic if there exists a computable bijection and with computable inverse. The natural numbers come with a natural representation and the space $\mathbf{X}^{\mathbb{N}}$ of functions with respect to this structure is isomorphic to the space $\mathbf{X}^\omega$ of sequences that was constructed as an infinite product at the beginning of this section. I.e. $\mathbf{X}^{\mathbb{N}} \simeq \mathbf{X}^\omega$. This means that there is an overlap in scope between the

function space construction and the infinite product. To understand this in more detail, let I be any countable and inhabited type. Set $\mathbf{Q_I} := \{\star\}$ and $\mathbf{A_I} := I$. Then $\delta_{\mathbf{I}}(\varphi) := \varphi(\star)$ makes $\mathbf{I} := (I, \delta_{\mathbf{I}})$ a represented space that is discrete in the following sense:

▶ **Lemma 6** (`cs_id_dscrt`). *For any countable, inhabited type I the represented space $\mathbf{I}$ from above is discrete in the sense that any function that has $\mathbf{I}$ as its domain is continuous.*

The set underlying the space $\prod_I \mathbf{X}$ is the set of functions from I to $\mathbf{X}$. Since $\mathbf{I}$ is discrete all functions from $\mathbf{I}$ to $\mathbf{X}$ are continuous and the sets underlying $\prod_I \mathbf{X}$ and $\mathbf{X}^{\mathbf{I}}$ are identical. Indeed these spaces are computably isomorphic and we formalized the proof of this.

▶ **Theorem 7** (`sig_iso_fun`). *For any represented space $\mathbf{X}$ and countable inhabited type I the space $\prod_I \mathbf{X}$ is computably isomorphic to $\mathbf{X}^{\mathbf{I}}$, where $\mathbf{I}$ is the discrete space over I.*

The realizer that translates from sequences to functions is defined using the simpler `F2MF` assignment, but relies on the details of how Incone implements the universal. This may be attributed to the fact that the above theorem need not be true in an arbitrary Cartesian closed category. The construction of a sequence from a continuous function proceeds by using a variation of the realizer of the evaluation operation that is proven computable for arbitrary represented spaces in the Incone library. On the one hand this makes it mostly independent of the implementation of the universal. On the other hand it means that the universal has to be executed and is thus an instance where a realizer uses the more complicated $F.$ assignment. An axiom-free definition of a realizer using the `F2MF` assignment is likely to be impossible. This is related to the fact that the construction of the reals from Dedekind cuts and Cauchy sequences are not fully equivalent in a constructive setting [35].

3 Metric spaces and closed choice on the naturals

A function $d\colon M \times M \to \mathbb{R}$ is called a **pseudo-metric** on a set M if it is positive, symmetric and fulfills $d(x,x) = 0$ and the triangle inequality $d(x,z) \leq d(x,y) + d(y,z)$. It is called a **metric** if $d(x,y) = 0$ implies $x = y$. A pair (M,d) is called a **pseudo-metric space** if d is a pseudo-metric on M and a **metric space** if d is a metric. Every pseudo-metric space comes with a topology that is generated by the open balls and therefore with notions of continuity of functions between and limits of sequences in such spaces. The latter is of particular importance since any pseudo-metric space is first-countable and thus knowing the limits is sufficient for characterizing continuity. A more accessible definition of continuity can be given using the well-known ε-δ-criterion that does not require any knowledge about topology. A function $f\colon N \to M$ between pseudo-metric spaces (N, d_N) and (M, d_M) is called **continuous in** x if

$$\forall \varepsilon, \exists \delta, \forall y, d_N(x,y) \leq \delta \implies d_M(f(x), f(y)) \leq \varepsilon.$$

The function is called **continuous** if it is continuous in any point of M. Here, ε and δ are a priori reals but may be replaced by rationals for density reasons. An element x of a pseudo-metric or metric space (M,d) is said to be the **limit** of a sequence (x_n) in M, in symbols $\lim_{(M,d)}(x_n) = x$, if

$$\forall \varepsilon, \exists N, \forall n, N \leq n \implies d(x, x_n) \leq \varepsilon.$$

The function f is then called **sequentially continuous** if $\lim_{(N,d_N)}(x_n) = x$ implies $\lim_{(M,d_M)}(f(x_n)) = f(x)$.

Metric spaces have received considerable attention in their formal treatment [36]. There exists a definition of the concept of a metric space and continuity of functions between metric spaces in the standard library of Coq. Several external libraries come with their own versions of metric spaces and continuity. Metric spaces and uniformly continuous functions are some of the core concepts of the C-CoRn library [44]. Another example is the Coquelicot library, which uses a concept it refers to as uniform space that closely resembles pseudo-metric spaces (`cntp_cntp`). The Incone library comes with its own version of metric spaces that is kept close to the classical mathematical treatment and is thus most similar to the metric spaces that can be found in Coq's standard library. It provides interfaces with both the standard library of Coq (`MS2M_S`, `M_S2MS`, `Uncv_lim`, `cont_limin`, etc.) and the Coquelicot library (`US2MS`, `MS2US`, `cntp_cntp`, etc.) so that it is possible to reuse results proven there. In contrast to the Coquelicot library, the metric library does not attempt to be conservative over the background theory of the real numbers.

While the naming of notions for metric spaces is identical to what we used for represented spaces, there are some conceptual differences. First off, a function between metric spaces is continuous if and only if it is sequentially continuous, where for represented spaces the backward implication can fail. A sufficient condition to recover it is admissibility of the involved representations [50]. Secondly, metric continuity can be recovered from a pointwise notion while continuous realizability can not. The pointwise notion introduces subtle problems in the treatment of subspaces. Even in the most well-behaved cases like a closed interval as subspace of the real numbers there is a difference between a function on the reals being continuous in each point of the interval and the restriction of the function to the interval being continuous. The statements of important theorems from the standard library (for instance the mean value theorem) do not account for this difference and diverge slightly from what a mathematician would expect. The metric library assumes proof-irrelevance to allow for a treatment of subspaces as dependent types.

3.1 Comparing continuity in represented and in metric spaces

Metric spaces are well investigated in computable analysis [64]. In particular in the case where (M, d) is a metric space and (r_n) is a designated dense sequence in M, M can be made a represented space $\mathbf{M} := (M, \delta_{\mathbf{M}})$ using the representation defined by

$$x \in \delta_{\mathbf{M}}(\varphi) \quad \iff \quad \forall n, d(x, r_{\varphi(n)}) \leq 2^{-n}.$$

Note that the idea behind this construction is nearly identical to that behind the representations of the reals: A name takes a precision requirement, now encoded as integer, and returns an approximation, or rather an index of an approximation.

A metric space is **separable** if there exists a dense sequence and even though the sequence goes into the definition of the corresponding Cauchy representation, we decide to not mention it explicitly in the following. This is justified in a continuity setting as different choices of dense sequences lead to isomorphic represented spaces. The situation is more complicated if computability is considered and the proofs in the library explicitly carry the sequences along.

▶ **Theorem 8** (`lim_mlim`). *Whenever* (M, d) *is a separable metric space and* $\mathbf{M}$ *as above then* $\lim_{(M,d)} = \lim_{\mathbf{M}}$.

The proof that the sequential notions of continuity on metric and represented space coincide follows immediately from this theorem. Each direction of the proof requires to translate limits in both directions and is thus as constructive or non-constructive as the worse direction of the previous theorem (which requires to assume mild choice principles).

▶ **Corollary 9** (`scnt_mscnt`). *If (M, d) and (M', d') are separable metric spaces, then a function $f : M \to N$, is sequentially continuous as a function between metric spaces if and only if it is sequentially continuous as function $f \colon \mathbf{M} \to \mathbf{M}'$.*

For the equivalence of ε-δ-continuity and continuous realizability one direction needs stronger assumptions and for the Incone library we have thus separated the proofs.

▶ **Lemma 10** (`cont_mcont` and `mcont_cont`). *Let (M, d) and (M', d') be two separable metric spaces. A function $f \colon M \to M'$ is ε-δ-continuous if and only if $f \colon \mathbf{M} \to \mathbf{M}'$ is continuous.*

The proof that continuous realizability implies ε-δ-continuity is straight forward, the proof of the other implication has turned out to be more complicated. We sketch some of the details.

Call a function $\mu \colon \mathbb{N} \to \mathbb{N}$ a **modulus of metric continuity** of f in x if

$$\forall y, d(x, y) \leq 2^{-\mu(n)} \implies d(f(x), f(y)) \leq 2^{-n}$$

And call such a modulus **minimal** if it is minimal in the obvious way.

▶ **Lemma 11** (`exists_minmod_met`). *For any continuous function f between metric spaces and any argument x for f there exists a minimal modulus of f in x.*

The proof relies on classical reasoning. This is a case where the use of axioms can not be avoided: In the special case where the metric space is Baire space the existence of a minimal modulus cannot be proven constructively [61].

If the source space is Baire space, it can be shown that the minimal modulus function is continuous in each x and from this Lemma 10 can be deduced. For general metric spaces, however, this strategy is bound to fail: If the metric space is connected, the function assigning to each x the minimal modulus function of f in x cannot be continuous as it takes values in the totally disconnected space $\mathbb{N}^{\mathbb{N}}$. One might expect that this is due to the awkward typing, and that making μ have type $\mathbb{R} \to \mathbb{R}$ instead would help, but it does not. It is known that also in this case the minimal modulus need not be continuous and that a construction of a continuous modulus of continuity, while possible in general, takes considerably more effort [16]. Our proof that ε-δ-continuity implies continuous realizability therefore proceeds differently and uses a notion of being almost-selfmodulating instead, where the value of the minimal modulus on slightly disturbed input from the metric space is bounded in terms of a shift of the minimal modulus in the original value.

Interestingly, similar tools as those in the above proof turn out to be useful in other parts of the Incone library. More specifically, the proof of correctness of the continuous universal that the library uses for the construction of function spaces also makes use of minimal moduli.

3.2 Recovering continuity on Baire space from a metric structure

Fix some types $\mathbf{Q}$ and $\mathbf{A}$ and set $\mathcal{B} := \mathbf{Q} \to \mathbf{A}$. Recall from the discussion in Section 2.1 that if $\mathcal{B}$ is a Baire space, then there exists a canonical way to make this space a represented space and that the elementary notion of continuity coincides with the represented space notion for partial functions. The limit operator $\lim_{\mathcal{B}}$ that this space gets as a represented space captures pointwise convergence with respect to the discrete topology on $\mathbf{A}$. The information theoretic notion of continuity on $\mathcal{B}$ from Section 2.1 is equivalent to sequential continuity in the associated represented space and a proof of this can be found in the Incone library.

For each function $\mathrm{cnt}\colon \mathbb{N} \to \mathbf{Q}$ define a mapping $d_{\mathrm{cnt}}\colon \mathcal{B} \times \mathcal{B} \to \mathbb{R}$ by

$$d_{\mathrm{cnt}}(\varphi, \psi) := \begin{cases} 2^{-k} & \text{if } \varphi \neq \psi \text{ and } k = \min\{n, \varphi(\mathrm{cnt}(n)) \neq \psi(\mathrm{cnt}(n))\} \\ 0 & \text{otherwise.} \end{cases}$$

Note that if $\mathcal{B}$ is a Baire space, then $\mathbf{Q}$ is countable and there exists some surjective function $\mathrm{cnt}\colon \mathbb{N} \to \mathbf{Q}$. This makes the above mapping a metric.

▶ **Proposition 12** (`dst_pos`, `dst_sym`, `dstxx`, `dst_trngl`, `dst_eq`). *Whenever* $\mathrm{cnt}\colon \mathbb{N} \to \mathbf{Q}$ *is surjective,* $(\mathcal{B}, d_{\mathrm{cnt}})$ *is a metric space.*

The core of the proof is an implementation of a function that approximates an unbounded search and developing some of its properties.

▶ **Theorem 13** (`lim_lim`). *Let* $\mathcal{B}$ *be a Baire space and* cnt *surjective, then* $\lim_{(\mathcal{B}, d_{\mathrm{cnt}})} = \lim_{\mathcal{B}}$.

The above theorem directly implies that the concepts of sequential continuity between Baire spaces coincides with the corresponding metric notion. For metric spaces sequential continuity and continuity are equivalent by combination of Corollary 9 and Lemma 10, thus:

▶ **Corollary 14** (`cont_cont`). *Whenever* $\mathcal{B}$ *and* $\mathcal{B}'$ *are Baire spaces and* cnt *and* cnt' *are appropriate surjective functions then* $F\colon \subseteq \mathcal{B} \to \mathcal{B}'$ *is continuous in the sense of Section 2.1 if and only if it is continuous as function from* $(\mathrm{dom}(F), d_{\mathrm{cnt}})$ *to* $(\mathcal{B}', d_{\mathrm{cnt}'})$.

▶ **Example 15** (`examples/continuous_search.v`). An instructive example is the search operator whose domain are those functions from $\mathbb{N}^{\mathbb{N}}$ that eventually return zero and whose value is the first argument on which such an input returns zero. This operator is continuous and does not have a continuous total extension. As the regular notion of continuity on the original Baire space $\mathbb{N}^{\mathbb{N}}$ is captured by the continuity introduced in Section 2.1, this is true for both the metric notion as well as the information-theoretic notion. This operator is not only continuous but computable and this is witnessed by the function that was used in the proof of Proposition 12 to approximate an unbounded search and is a good example for the mechanisms discussed in Section 2.1.

3.3 Sierpiński space and closed choice on the naturals

This section describes the content of the file `examples/closed_choice.v` from the Incone library. Sierpiński space $\mathbb{S}$ (`cs_Sirp` in the library) is the space whose base set is the two point set $\{\bot, \top\}$ equipped with the total representation $\delta_{\mathbb{S}}\colon \subseteq (\mathbb{N} \to \mathbb{B}) \to \mathbb{S}$ specified by

$$\delta_{\mathbb{S}}(\varphi) = \top \quad \iff \quad \exists n \in \mathbb{N}\ \varphi(n) \neq \mathrm{false}.$$

For a subset $U \subseteq \mathbf{X}$ denote by $\chi_U\colon \mathbf{X} \to \mathbb{S}$ its characteristic function. One reason for the importance of Sierpiński space in computable analysis is that a set $U \subseteq \mathbf{X}$ is open if and only if this characteristic function χ_U is continuous as a function from $\mathbf{X}$ to $\mathbb{S}$. Thus we can identify the space $\mathcal{O}(\mathbf{X})$ of open subsets of $\mathbf{X}$ with the function space $\mathbb{S}^{\mathbf{X}}$ [46]. Similarly, the space $\mathcal{A}(\mathbf{X})$ of closed subsets of $\mathbf{X}$ is represented as the complements of opens.

For many concrete spaces $\mathbf{X}$ simpler descriptions of $\mathcal{O}(\mathbf{X})$ and $\mathcal{A}(\mathbf{X})$ are available. If the represented space $\mathbf{X} = \mathbb{N}$ are the natural numbers, for instance, the infinite product construction and in particular of the special case $I = \mathbb{N}$ and $\mathbf{X} = \mathbb{S}$ of the statement $\mathbf{X}^{\mathbf{I}} \simeq \prod_I \mathbf{X}$ of Lemma 7 guarantees that $\mathcal{O}(\mathbb{N}) = \mathbb{S}^{\mathbb{N}} \simeq \prod_{\mathbb{N}} \mathbb{S} = \mathbb{S}^{\omega}$. There exists a fully concrete description that is often used for reasoning about $\mathcal{O}(\mathbb{N})$. Consider the enumeration

representation of the open subsets of the natural numbers, where a name of an open set enumerates its elements. We call the corresponding space $\mathcal{O}_{\mathbb{N}}$. The representation of the concrete space $\mathcal{A}_{\mathbb{N}}$ of the closed subsets of the natural numbers is given by $\delta_{\mathcal{A}_{\mathbb{N}}}(\varphi) = \mathbb{N} \setminus \{n\colon \mathbb{N} \mid \exists m\colon \mathbb{N},\ \varphi(m) = n+1\}$. The information a name specifies about a closed set is an enumeration of its complement. The underlying sets of the spaces of opens and closeds of $\mathbb{N}$ are all subsets, but the information about such sets that is made available by names differs.

We provide a formal proof that the enumeration representations of the open and closed subsets of the natural numbers capture the abstract structure these spaces can be given through the exponential in the category of represented spaces and Sierpiński space.

▶ **Theorem 16** (`AN_iso_Anat`, `ON_iso_Onat` and `clsd_iso_open`). *$\mathcal{A}(\mathbb{N}) \simeq \mathcal{A}_{\mathbb{N}}$, $\mathcal{O}(\mathbb{N}) \simeq \mathcal{O}_{\mathbb{N}}$ and $\mathcal{A}(\mathbb{N}) \simeq \mathcal{O}(\mathbb{N})$.*

The last of these isomorphies is trivial: the isomorphism is taking the complement and it is realized by the identity function. The isomorphism of $\mathcal{O}(\mathbb{N})$ and $\mathcal{O}_{\mathbb{N}}$ is proven by first replacing $\mathcal{O}(\mathbb{N})$ by $\mathbb{S}^\omega$ as described above. The realizers for the isomorphism between $\mathbb{S}^\omega$ and $\mathcal{O}_{\mathbb{N}}$ uses the Cantor paring function provided by the Mathematical Components library.

As an application let us consider some choice operators that are popular for classification of computational tasks with respect to their Weihrauch degree. Solving the task of choosing an element of a non-empty closed subset of a represented space $\mathbf{X}$ can be formalized as asking for a realizer for the multivalued function $C_{\mathbf{X}}$ defined by

$$C_{\mathbf{X}}\colon \mathcal{A}(\mathbf{X}) \rightrightarrows \mathbf{X}, \quad a \in C_{\mathbf{X}}(A) \iff a \in A.$$

Or in words: a is an acceptable return value of $C_{\mathbf{X}}$ on input A if and only if a is an element of A. The domain of $C_{\mathbf{X}}$ are the non-empty subsets of $\mathbf{X}$ and this means that a realizer can behave arbitrarily on names of the empty set and may even diverge. As the input A is given as a closed set where a name specifies negative information about element-hood, this task does not have a continuous, let alone computable, solution for most spaces $\mathbf{X}$.

Consider the case $\mathbf{X} = \mathbb{N}$. The domain of the multivalued function $C_{\mathbb{N}}$ is $\mathcal{A}(\mathbb{N})$ but the same definition also specifies a multifunction $C'_{\mathbb{N}}\colon \mathcal{A}_{\mathbb{N}} \rightrightarrows \mathbb{N}$ whose source space $\mathcal{A}_{\mathbb{N}}$ uses the enumeration representation. A mathematician may even consider it pointless to give this function a different name as isomorphic spaces are regularly identified. For the question whether $C_{\mathbb{N}}$ has a continuous realizer $\mathcal{A}(\mathbb{N})$ may be substituted with $\mathcal{A}_{\mathbb{N}}$ (`CN_CN'_hcr`).

▶ **Proposition 17** (`CN'_not_cont`). *$C'_{\mathbb{N}}$ does not have a continuous realizer.*

Our formal proof follows the standard proof by contradiction literally: Assume that to the contrary that F was a continuous realizer of $C'_{\mathbb{N}}$. Pick any name φ of the one point set $\{0\}$. As F is a realizer, it has to return a name of 0 on input φ, i.e., $F(\varphi)(\star) = 0$. Since F is continuous there is a list $L \subseteq \mathbb{N}$ such that $F(\varphi)(\star) = F(\psi)(\star)$ for all $\psi\colon \mathbb{N} \to \mathbb{N}$ that coincide with φ on L. Consider the name φ' of the non-empty set $A := \mathbb{N} \setminus (\{n \mid \exists m \in L, \varphi(m) = n+1\} \cup \{0\})$ defined by $\varphi'(n) := \varphi(n)$ if $n \in L$ and 1 otherwise. On the one hand, $F(\varphi')(\star) \in A$ since F is a realizer. On the other hand $F(\varphi')(\star) = F(\varphi)(\star) = 0$ as φ and φ' coincide on L. By definition of A it holds that $0 \notin A$, which is a contradiction and completes the proof.

From the previous result together with exchangeability of $C_{\mathbb{N}}$ and $C'_{\mathbb{N}}$ it is follows that:

▶ **Corollary 18** (`CN_not_cont`). *Closed choice on the natural numbers is discontinuous.*

4 Conclusion

The Incone library formalizes ideas from computable analysis in Coq. There exists some overlap with other developments, in particular with C-CoRn. However, the emphasis of the library is different and many of our examples fall outside of the scope of C-CoRn and similar developments. It may be considered complementary as it provides general purpose tools for enriching abstract mathematical objects with computational structure. We feel that the example from the last section of this paper showcases the capabilities of the library well. The abstract definition of the space of open subsets is based on Incone's function space construction and the proof that it is equivalent to the concrete representation relies on the libraries results about infinite products of represented spaces. We believe the Incone library to be reasonably accessible to people familiar with the setting that computable analysis works in. We hope that combination with recent developments in computable analysis [41] could open it to an even wider audience including parts of the numerical analysis community.

The Incone library keeps close to recent work about complexity theory for computable analysis [25, 18, 42, 27] such that it should be possible to add capabilities to at least do qualitative complexity theory in terms of tracking the rate of decrease in accuracy of approximations. Recently there has been a lot of progress on the formalization of models of computation [20, 66] and methods from implicit complexity theory [19] that may even allow to do quantitative complexity theory. Another way to gain insight into such efficiency considerations would be to capture the trace of the basic feasible functionals on the operators on Baire space [39, 23, 24].

The replacement of Baire space by more general spaces means that we maintain the ability to benefit from Coq's machinery in the low-level manipulations of data. From an abstract point of view this makes our approach look like an attempt to interpret a class of generalized Kleene-Kreisel continuous functionals as a computational model in presence of an ambient model of computation. This is a backwards approach to the more common idea of identifying a sub-algebra that captures computability in a given partial combinatory algebra, in this case K_2 [34, 3]. Most of the methods from the Rlzrs library are not original and have been implemented independently of a specific proof assistant before [4]. An implementation in other proof assistants one could trade convenience in computationally operating on discrete data against bigger mathematical libraries.

We feel that this paper provides sufficient evidence that the concepts developed in the Incone library can be used as a foundation for proving statements from computable analysis in Coq. The possible applications we are interested to look into are manifold. One that would be a particularly fitting extension of the contents of this paper is a proof that $C([0,1]) \simeq \mathbb{R}^{[0,1]}$. This statement is called the Computable Weierstraß Theorem [49]: $C([0,1])$ is represented as separable metric space with supremum norm and the rational polynomials as dense sequence and $\mathbb{R}^{[0,1]}$ is a function space. Other possibilities include:

- A more computation-efficient representation of real numbers and results about ODE solving [22, 37, 26]. This may be done by providing an interface with C-CoRn, parts of it could also be done separately by relying on libraries like Coq-Interval.
- Duality theory for spaces of summable sequences (ℓ_p-spaces) which provide a pool of examples where subspaces of exponentials can be treated complexity theoretically [53, 51]. Additionally it constitutes a step towards capturing popular methods for solving partial differential equations [12, 54, 10].
- A characterization of continuity via preimages of open sets and similar results [46, 52].

References

1 Klaus Ambos-Spies, Ulrike Brandt, and Martin Ziegler. Real Benefit of Promises and Advice. In Paola Bonizzoni, Vasco Brattka, and Benedikt Löwe, editors, *The Nature of Computation. Logic, Algorithms, Applications*, pages 1–11, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

2 Jeremy Avigad and Vasco Brattka. *Computability and analysis: the legacy of Alan Turing*, page 1–47. Lecture Notes in Logic. Cambridge University Press, 2014. doi:10.1017/CBO9781107338579.002.

3 Andrej Bauer. *The Realizability Approach to Computable Analysis and Topology.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2000. AAI3002721.

4 Andrej Bauer and C.A. Stone. RZ: A tool for bringing constructive and computable mathematics closer to programming practice. *Computation and Logic in the Real World. CiE 2007. Lecture Notes in Computer Science, vol 4497.*, 2007.

5 Yves Bertot, Laurence Rideau, and Laurent Théry. Distant decimals of π. *Journal of Automated Reasoning*, pages 1–45, 2017. URL: https://hal.inria.fr/hal-01582524.

6 Errett Bishop and Douglas Bridges. *Constructive analysis*, volume 279. Springer Science & Business Media, 2012.

7 Jens Blanck. Exact real arithmetic systems: Results of competition. In *Computability and complexity in analysis. 4th international workshop, CCA 2000. Swansea, GB, September 17–19, 2000. Selected papers*, pages 389–393. Berlin: Springer, 2001.

8 Lenore Blum, Mike Shub, and Steve Smale. On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines. *Bulletin of the American Mathematical Society*, 21(1):1–46, 1989.

9 Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Wave Equation Numerical Resolution: a Comprehensive Mechanized Proof of a C Program. *Journal of Automated Reasoning*, 50(4):423–456, April 2013. doi:10.1007/s10817-012-9255-4.

10 Sylvie Boldo, François Clément, Florian Faissole, Vincent Martin, and Micaela Mayero. A Coq Formal Proof of the Lax–Milgram theorem. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 79–89, Paris, France, January 2017. ACM. doi:10.1145/3018610.3018625.

11 Sylvie Boldo and Guillaume Melquiond. *Computer Arithmetic and Formal Proofs.* ISTE Press - Elsevier, December 2017. URL: https://hal.inria.fr/hal-01632617.

12 Vasco Brattka and Atsushi Yoshikawa. Towards computability of elliptic boundary value problems in variational formulation. *Journal of Complexity*, 22(6):858–880, 2006. Computability and Complexity in Analysis. doi:10.1016/j.jco.2006.04.007.

13 Cyril Cohen. Construction of real algebraic numbers in Coq. In Lennart Beringer and Amy Felty, editors, *ITP - 3rd International Conference on Interactive Theorem Proving - 2012*, Princeton, United States, August 2012. Springer.

14 Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-CoRN, the constructive Coq repository at Nijmegen. In *International Conference on Mathematical Knowledge Management*, pages 88–103. Springer, 2004.

15 K. Deimling. *Multivalued Differential Equations.* De Gruyter series in nonlinear analysis and applications. W. de Gruyter, 1992.

16 Ali Enayat. δ as a continuous function of x and ε. *The American Mathematical Monthly*, 107(2):151–155, 2000.

17 Martín Escardó and Chuangjie Xu. A constructive manifestation of the Kleene–Kreisel continuous functionals. *Annals of Pure and Applied Logic*, 167(9):770–793, 2016. Fourth Workshop on Formal Topology (4WFTop). doi:10.1016/j.apal.2016.04.011.

18 Hugo Feree. Game Semantics Approach to Higher-order Complexity. *J. Comput. Syst. Sci.*, 87(C):1–15, August 2017. doi:10.1016/j.jcss.2017.02.003.

19 Hugo Férée, Samuel Hym, Micaela Mayero, Jean-Yves Moyen, and David Nowak. Formal proof of polynomial-time complexity with quasi-interpretations. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 146–157. ACM, 2018.

20 Yannick Forster and Gert Smolka. Call-by-Value Lambda Calculus as a Model of Computation in Coq. *Journal of Automated Reasoning*, 2018.

21 A. Grzegorczyk. On the definitions of computable real continuous functions. *Fund. Math.*, 44:61–71, 1957.

22 Fabian Immler and Johannes Hölzl. Numerical Analysis of Ordinary Differential Equations in Isabelle/HOL. In *ITP*, volume 7406 of *LNCS*, pages 377–392, 2012.

23 Bruce M. Kapron and Stephen A. Cook. A New Characterization of Type-2 Feasibility. *SIAM J. Comput.*, 25:117–132, 1996.

24 Bruce M. Kapron and Florian Steinberg. Type-two polynomial-time and restricted lookahead. In *LICS*, 2018.

25 Akitoshi Kawamura and Stephen Cook. Complexity theory for operators in analysis. *ACM Transactions in Computation Theory*, 4(2):Article 5, 2012.

26 Akitoshi Kawamura, Florian Steinberg, and Holger Thies. Parameterized Complexity for Uniform Operators on Multidimensional Analytic Functions and ODE Solving. In *International Workshop on Logic, Language, Information, and Computation*, pages 223–236. Springer, 2018.

27 Akitoshi Kawamura, Florian Steinberg, and Holger Thies. Second-order linear-time computability with applications in computable analysis. *15th Annual Conference on Theory and Applications of Models of Computation*, 2019. extended abstract accepted for presentation at TAMC 2019.

28 S.C. Kleene. Countable functionals. *Constructivity in Mathematics: proceedings of the colloquium held at Amsterdam*, 1959.

29 Ker-I Ko. *Complexity theory of real functions*. Progress in Theoretical Computer Science. Birkhäuser Boston Inc., Boston, MA, 1991.

30 Michal Konecný and Eike Neumann. Representations and evaluation strategies for feasibly approximable functions. *CoRR*, abs/1710.03702, 2017. `arXiv:1710.03702`.

31 Georg Kreisel. Interpretation of analysis by means of constructive functionals of finite type, Constructivity in Mathematics, 1959.

32 Christoph Kreitz and Klaus Weihrauch. Theory of representations. *Theoretical computer science*, 38:35–53, 1985.

33 Daniel Lacombe. Sur les possibilités d'extension de la notion de fonction récursive aux fonctions d'une ou plusieurs variables réelles. In *Le raisonnement en mathématiques et en sciences expérimentales*, Colloques Internationaux du Centre National de la Recherche Scientifique, LXX, pages 67–75. Editions du Centre National de la Recherche Scientifique, Paris, 1958.

34 John Longley and Dag Normann. *Higher-order computability*, volume 100. Springer, 2015.

35 Robert S. Lubarsky and Michael Rathjen. On the constructive Dedekind reals. *Logic and Analysis*, 1(2):131–152, May 2008. `doi:10.1007/s11813-007-0005-6`.

36 Marco Maggesi. A Formalization of Metric Spaces in HOL Light. *J. Autom. Reasoning*, 60(2):237–254, 2018.

37 Evgeny Makarov and Bas Spitters. The Picard Algorithm for Ordinary Differential Equations in Coq. In *ITP*, volume 7998 of *Lecture Notes in Computer Science*, pages 463–468. Springer, 2013.

38 The Mathematical Components library. `https://math-comp.github.io/math-comp/`.

39 Kurt Mehlhorn. Polynomial and abstract subrecursive classes. *Journal of Computer and System Sciences*, 12(2):147–178, 1976. `doi:10.1016/S0022-0000(76)80035-9`.

40 Norbert Th. Müller. The iRRAM: Exact arithmetic in C++. In *Computability and complexity in analysis. 4th international workshop, CCA 2000. Swansea, GB, September 17–19, 2000. Selected papers*, pages 222–252. Berlin: Springer, 2001.

41 Norbert Th. Müller, Sewon Park, Norbert Preining, and Martin Ziegler. On Formal Verification in Imperative Multivalued Programming over Continuous Data Types. *CoRR*, abs/1608.05787, 2016. arXiv:1608.05787.

42 Eike Neumann and Florian Steinberg. Parametrised second-order complexity theory with applications to the study of interval computation. *CoRR*, abs/1711.10530, 2017. submitted for publication. arXiv:1711.10530.

43 Russell O'Connor. Essential Incompleteness of Arithmetic Verified by Coq. In Joe Hurd and Tom Melham, editors, *Theorem Proving in Higher Order Logics*, pages 245–260, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

44 Russell O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. PhD thesis, Radboud Universiteit Nijmegen, 2009.

45 Arno Pauly. Multi-valued functions in computability theory. In *Conference on Computability in Europe*, pages 571–580. Springer, 2012.

46 Arno Pauly. On the topological aspects of the theory of represented spaces. *Computability*, 5(2):159–180, 2016.

47 Arno Pauly and Martin Ziegler. Relative computability and uniform continuity of relations. *J. Logic & Analysis*, 5, 2013.

48 Marian B. Pour-El and J. Ian Richards. *Computability in Analysis and Physics*, volume Volume 1 of *Perspectives in Mathematical Logic*. Springer-Verlag, Berlin, 1989.

49 Marian Boykan Pour-El and Jerome Caldwell. On a simple definition of computable function of a real variable-with applications to functions of a complex variable. *Mathematical Logic Quarterly*, 21(1):1–19, 1975.

50 Matthias Schröder. Extended admissibility. *Theoretical computer science*, 284(2):519–538, 2002.

51 Matthias Schröder and Florian Steinberg. Bounded time computation on metric spaces and Banach spaces. 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.2017.8005139.

52 Matthias Schröeder. *Admissible Representations for Continuous Computations*. PhD thesis, FernUniversität Hagen, 2002.

53 Matthias Schröder. Spaces allowing Type-2 Complexity Theory revisited. *Math. Log. Q.*, 50:443–459, September 2004. doi:10.1002/malq.200310111.

54 Svetlana Selivanova and Victor Selivanov. Computing Solutions of Symmetric Hyperbolic Systems of PDE's. *Electron. Notes Theor. Comput. Sci.*, 221:243–255, December 2008. doi:10.1016/j.entcs.2008.12.021.

55 Robert I Soare. Recursively enumerable sets and degrees. *Bulletin of the American Mathematical Society*, 84(6):1149–1181, 1978.

56 Florian Steinberg. The Incone library. https://github.com/FlorianSteinberg/incone, 2019. release v1.0.

57 Florian Steinberg. The Metric library. https://github.com/FlorianSteinberg/metric, 2019. release v1.0.

58 Florian Steinberg. The Mf library. https://github.com/FlorianSteinberg/mf, 2019. release v1.0.

59 Florian Steinberg. The Rlzrs library. https://github.com/FlorianSteinberg/rlzrs, 2019. release v1.0.

60 Florian Steinberg, Laurent Thery, and Holger Thies. Quantitative continuity and computable analysis in Coq. working paper or preprint, April 2019. URL: https://hal.inria.fr/hal-02088293.

61 A. S. Troelstra and D. van Dalen. *Constructivism in mathematics. Vol. II*, volume 123 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Co., Amsterdam, 1988.

62 A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(1):230–265, 1936. doi:10.1112/plms/s2-42.1.230.

63 Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. A correction. *Proceedings of the London Mathematical Society*, 2(1):544–546, 1938.

64 Klaus Weihrauch. Computability on computable metric spaces. *Theoretical Computer Science*, 113(2):191–210, 1993.

65 Klaus Weihrauch. *Computable Analysis*. Springer, Berlin/Heidelberg, 2000.

66 Maximilian Wuttke. Verified Programming of Turing Machines in Coq. Master's thesis, Saarland University, 2018.

Deriving Proved Equality Tests in Coq-Elpi: Stronger Induction Principles for Containers in Coq

Enrico Tassi
Université Côte d'Azur – Inria, France
Enrico.Tassi@inria.fr

Abstract

We describe a procedure to derive equality tests and their correctness proofs from inductive type declarations in Coq. Programs and proofs are derived compositionally, reusing code and proofs derived previously.

The key steps are two. First, we design appropriate induction principles for data types defined using parametric containers. Second, we develop a technique to work around the modularity limitations imposed by the purely syntactic termination check Coq performs on recursive proofs. The unary parametricity translation of inductive data types turns out to be the key to both steps.

Last but not least, we provide an implementation of the procedure for the Coq proof assistant based on the Elpi [6] extension language.

2012 ACM Subject Classification Software and its engineering → General programming languages

Keywords and phrases Coq, Containers, Induction, Equality test, Parametricity translation

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.29

Supplement Material Source code of the Coq package: `https://github.com/LPCIC/coq-elpi`

1 Introduction

Modern typed programming languages come with the ability of generating boilerplate code automatically. Typically when a data type is declared a substantial amount of code is made available to the programmer at little cost, code such as an equality test, a printing function, generic visitors etc. For example the `derive` directive of Haskell or the `ppx_deriving` OCaml preprocessor provide these features for the respective programming language.

The situation is less than ideal in the Coq proof assistant. It is capable of synthesizing the recursor of a data type, that, following the Curry-Howard isomorphism, implements the induction principle associated to that data type. It supports all data types, containers such as lists included, but generates a quite weak principle when a data type *uses* a container. Take for example the data type rose tree (where `U` stands for a universe such as `Prop` or `Type`):

```
Inductive rtree A : U :=
| Leaf (a : A)
| Node (l : list (rtree A)).
```

Its associated induction principle is the following one:

```
1 Lemma rtree_ind : ∀A (P : rtree A → U),
2     (∀a : A, P (Leaf A a)) →
3     (∀l : list (rtree A), P (Node A l)) →
4   ∀t : rtree A, P t.
```

Remark that the recursive step, line 3, lacks any induction hypotheses on (the elements of) `l` while one would expect `P` to hold on each and every subtree. Even a very basic recursive program such as an equality test cannot be proved correct using this induction principle. To

be honest, the Coq user is not even supposed to write equality tests by hand, nor to prove them correct interactively. Coq provides two facilities to synthesize equality tests and their correctness proofs called `Scheme Equality` and `decide equality`. The former is fully automatic but is unfortunately very limited, for example it does not support containers. The latter requires human intervention and generates a single, large, term that mixes code and proofs.

As a consequence, users often need to manually write induction principles, equality tests and their correctness proofs. This situation is very unfortunate because the need for the automatic generation of boilerplate code such as equality tests is higher than ever in the Coq ecosystem. All modern formal libraries structure their contents in a hierarchy of interfaces and some machinery such as Type Classes [18] or Canonical Structures [9] are used to link the abstract library to the concrete instances the user is working on. For example the first interface one is required to implement in order to use the theorems in the Mathematical Components library [10] on a type `T` is the `eqType` one, requiring a correct equality test on `T`.

In this paper we use the framework for meta programming based on Elpi [6, 19] developed by the author and we focus on the derivation of equality tests. It turns out that generating equality tests is easy, while their correctness proofs are hard to synthesize, for two reasons. The first problem is that the standard induction principles generated by Coq, as shown before, are too weak. In order to strengthen them one needs quite some extra boilerplate, such as the derivation of the unary parametricity translation of the data types involved. The second reason is that termination checking is purely syntactic in Coq: in order to check that the induction hypothesis is applied to a smaller term, Coq may need to unfold all theorems involved in the proof. This forces proofs to be transparent that, in turn, breaks modularity: A statement is no more a contract, changing its proof may impact users.

In this paper we describe a derivation procedure for equality tests and their correctness proofs where programs and proofs are both derived compositionally, reusing code and proofs derived previously. This procedure also confines the termination check issue, allowing proofs to be mostly opaque. More precisely the contributions of this paper are the following ones:

- A technique to confine the issue stemming from the purely syntactic termination check implemented by Coq out of the main proofs. In this paper we apply it to the correctness proof of equality tests, but the technique is applicable to all proofs that proceed by structural induction.
- A modular and structured process to derive proved equality tests and, en passant, stronger induction principles for inductive types defined using containers.
- An implementation based on the Elpi extension language for the Coq proof assistant.

By installing the `coq-elpi` package[1] and issuing the command `Elpi derive rtree` one gets the following terms synthesized out of the type declaration for `rtree`:

```
Definition eq_axiom T f x := ∀y, reflect (x = y) (f x y).

Definition rtree_eq : ∀A, (A → A → bool) → rtree A → rtree A → bool.

Lemma rtree_eq_OK : ∀A (A_eq : A → A → bool), (∀a, eq_axiom A A_eq a) →
  ∀t, eq_axiom (rtree A) (rtree_eq A A_eq) t.
```

`reflect` is a predicate stating the equivalence between the proposition `(x = y)` and the boolean test `(f x y)`; `rtree_eq` is a (transparent) equality test and `rtree_eq_OK` is its (opaque) correctness proof under the assumption that the equality test `A_eq` is correct.

[1] See the supplementary material URL for the installation instructions.

The paper introduces the problem in section 2 by describing the shape of an equality test and of its correctness proof and explaining the modularity problem that stems for the termination checker of Coq. It then presents the main idea behind the modular derivation procedure in section 3. Section 4 briefly introduces the Elpi extension language and section 5 describes the full derivation.

2 The problem: opaque proofs v.s. syntactic termination checking

Recursors, or induction principles, are not primitive notions in Coq. The language provides constructors for fix point and pattern matching that work on any inductive data the user can declare. For example in order to test two lists `l1` and `l2` for equality one typically takes in input an equality test `A_eq` for the elements of type `A` and then performs the recursion:

```
1  Definition list_eq A (A_eq : A → A → bool) :=
2    fix rec (l1 l2 : list A) {struct l1} : bool :=
3      match l1, l2 with
4      | nil, nil => true
5      | x :: xs, y :: ys => A_eq x y && rec xs ys
6      | _, _ => false
7      end.
```

Coq accepts this definition because the recursive call is on `xs` that is a syntactically smaller term of the argument labelled as decreasing by the `{struct l1}` annotation.

We can define the equality test for `rtree` by reusing the equality test for lists:

```
8  Definition rtree_eq B (B_eq : B → B → bool) :=
9    fix rec (t1 t2 : rtree B) {struct t1} : bool :=
10     match t1, t2 with
11     | Leaf x, Leaf y => B_eq x y
12     | Node l1, Node l2 => list_eq (rtree B) rec l1 l2
13     | _, _ => false
14     end.
```

Note that `list_eq` is called passing as the `A_eq` argument the fixpoint `rec` itself (line 12). In order to check that the latter definition is sound, Coq looks at the body of `list_eq` to see whether its parameter `A_eq` is applied to a term smaller than `t1`. Since `l1` is a subterm of `t1` and since `x` is a subterm of `l1`, then the recursive call `(rec x y)` at line 5 is legit.

The fact that checking `rtree_eq` requires inspecting the body of `list_eq` is not very annoying: we want both `list_eq` and `rtree_eq` to compute, hence their body matters to us.

On the contrary proof terms are typically hidden to the type checker once they have been validated, for both performance and modularity reasons. The desire is to make only the statement of theorems binding, and keep the freedom to clean, refactor, simplify proofs without breaking the rest of the formal development.

For example, lets assume that `list_eq_OK` is an opaque proof that `list_eq` is correct.

```
1  Lemma list_eq_OK : ∀A (A_eq : A → A → bool),
2     (∀a, eq_axiom A A_eq a) →
3    ∀l, eq_axiom (list A) (list_eq A A_eq) l.
4  Proof. .. Qed. (* proof is opaque, hence hidden *)
```

It seems desirable to use this lemma in order to prove the correctness of `rtree_eq`, since it calls `list_eq`.

```
5  Lemma rtree_eq_OK B B_eq (HB: ∀b, eq_axiom B B_eq b) :
6    ∀t, eq_axiom (rtree B) (rtree_eq B B_eq) t
7  :=
8    fix IH (t1 t2 : rtree B) {struct t1} :=
9      match t1, t2 with
10     | Node l1, Node l2 => .. list_eq_OK (rtree B) (tree_eq B B_eq) IH l1 l2 ..
11     | Leaf b1, Leaf b2 => .. HB b1 b2 ..
12     | .. => ..
13     end.
```

Unfortunately this term is rejected: we pass `IH`, the induction hypothesis, as the witness that `(tree_eq B B_eq)` is a correct equality test (the argument at line 10 preceding `IH`) but Coq does not know how `list_eq_OK` uses this argument, since its body is opaque.

The issue seems unfixable without changing Coq in order to use a more modular check for termination, for example based on sized types [1]. We propose a less ambitious but more practical approach here, that consists in putting the transparent terms that the termination checker is going to inspect outside of the main proof bodies so that they can be kept opaque.

The intuition is to "reify" the property the termination checker wants to enforce. It can be phrased as "x is a subterm of t and has the same type". More in general we model "x is a subterm of t with property P".

3 The idea: put unary parametricity translation to good use

Given an inductive type `T` we name `is_T` an inductive predicate describing the type of the inhabitants of `T`. This is the one for natural numbers:

```
Inductive is_nat : nat → U :=
| is_O : is_nat 0
| is_S n (pn : is_nat n) : is_nat (S n).
```

The one for a container such as `list` is more interesting:

```
Inductive is_list A (is_A : A → U) : list A → U :=
| is_nil : is_list A is_A nil
| is_cons a (pa : is_A a) l (pl : is_list A is_A l) : is_list A is_A (a :: l).
```

Remark that all the elements of the list validate `is_A`.

When a type `T` is defined in terms of another type `C`, typically a container, the `is_C` predicate shows up inside `is_T`. For example:

```
1  Inductive is_rtree A (is_A : A → U) : rtree A → U :=
2  | is_Leaf a (pa : is_A a) : is_rtree A is_A (Leaf A a)
3  | is_Node l (pl : is_list (rtree A) (is_rtree A is_A) l) : is_rtree A is_A (Node A l).
```

Note how line 3 expresses the fact that all elements in the list `l` validate `(is_rtree A is_A)`.

Our intuition is that these predicates reify the notion of being of a certain type, structurally. What we typically write `(t : T)` can now be also phrased as `(is_T t)` as one would do in a framework other than type theory, such as a mono-sorted logic.

It turns out that the inductive predicate `is_T` corresponds to the unary parametricity translation [22] of the type `T`. Keller and Lasson in [8] give us an algorithm to synthesize these predicates automatically. What we look for now is a way to synthesize a reasoning principle for a term `t` when `(is_T t)` holds.

3.1 Stronger induction principles for containers

Let's have a look at the standard induction principle of lists.

```
Lemma list_ind A (P : list A → U) :
   P nil →
   (∀a l, P l → P (a :: l)) →
  ∀l : list A, P l.
```

This principle is parametric on `A`: no knowledge on any term of type `A` such as `a` is ever available. We want to synthesize a more powerful principle that lets us choose an invariant for the subterms of type `A` (the differences are underlined):

```
1 Lemma list_induction A (is_A: A → U) (P: list A → U):
2    P nil →
3    (∀a (pa : is_A a) l, P l → P (a :: l)) →
4   ∀l, is_list A is_A l → P l.
```

Note the extra premise `(is_list A is_A l)`: The implementation of this induction principle goes by recursion on the term of this type and finds as an argument of the `is_cons` constructor the proof evidence `(pa : is_A a)` it feeds to the second premise (line 3). Intuitively all terms of type `(list A)` validate the property `P`, while all terms of type `A` validate the property `is_A`.

More in general to each type we attach a property. For parameters we let the user choose (we take another parameter, `is_A` here). For the type being analysed, `list A` here, we take the usual induction predicate `P`. For terms of other types we use their unary parametricity translation. Take for example the induction principle for `rtree`.

```
1 Lemma rtree_induction A is_A (P : rtree A → U) :
2    (∀a, is_A a → P (Leaf A a)) →
3    (∀l, is_list (rtree A) P l → P (Node A l)) →
4   ∀t, is_rtree A is_A t → P t.
```

Line 3 uses `is_list` to attach a property to `l`, and given that `l` has type `(list (rtree A))` the property for the type parameter `(rtree A)` is exactly `P`. Note that this induction principle gives us access to `P`, the property one is proving, on the subtrees contained in `l`.

3.1.1 Synthesizing stronger induction principles

We postpone a detailed description of the synthesis to section 5.4, here we just sketch how to build the type on the induction principle.

It turns out that the types of the constructors of `is_T` give us a very good hint on the type of the induction principle. The type of the first premise

```
   (∀a, is_A a → P (Leaf A a)) →
```

is exactly the type of the `is_Leaf` constructor

```
| is_Leaf a (pa : is_A a) : is_rtree A is_A (Leaf A n)
```

where `(is_rtree A is_A)` is replaced by `P`. The same holds for the other premise: its type can be trivially obtained from the type of `is_Node`.

Our intuition is that the inductive predicate `is_T` provides the same information that typing provides. Induction principles give `P` on (smaller) terms of the same type, that would be terms for which `is_T` holds. Given their inductive nature, `is_T` predicates are able to propagate the desired property inside parametric containers.

3.2 Isolating the syntactic termination check problem

As one expects, it is possible to prove that `is_T` holds for terms of type `T`.

```
Definition nat_is_nat : ∀n : nat, is_nat n :=
  fix rec n : is_nat n :=
    match n as i return (is_nat i) with
    | 0 => is_O
    | S p => is_S p (rec p)
    end.
```

For containers `(T A)` we can prove `(is_T A is_A)` when `is_A` is trivial.

```
Definition list_is_list : ∀A (is_A : A → U), (∀a, is_A a) → ∀l, is_list A is_A l.

Definition rtree_is_rtree : ∀A (is_A : A → U), (∀a, is_A a) → ∀t, is_rtree A is_A t.
```

These facts are then to be used in order to satisfy the premise of our induction principles.

Going back to our goal, we can build correctness proofs of equality tests in two steps. For example, for natural numbers we can generate two lemmas:

```
1  Lemma nat_eq_correct : ∀n, is_nat n → eq_axiom nat nat_eq n :=
2    nat_induction (eq_axiom nat nat_eq) PO PS.
3
4  Lemma nat_eq_OK n : eq_axiom nat nat_eq n :=
5    nat_eq_correct n (nat_is_nat n).
```

where `PO` and `PS` (line 2) stand for the two proof terms corresponding to the base case and the inductive step of the proof. We omit them here for brevity.

For containers such as `(list A)` we can link the pieces in a similar way (at line 3 we omit the proofs for `nil` and `cons` as before).

```
1  Lemma list_eq_correct A A_eq : ∀l, is_list A (eq_axiom A A_eq) l →
2      eq_axiom list A (list_eq A A_eq) l :=
3    list_induction A (eq_axiom A A_eq) (eq_axiom (list A) (list_eq A A_eq)) Pnil Pcons.
4
5  Lemma list_eq_OK A A_eq (HA : ∀a, eq_axiom A A_eq a) l :
6      eq_axiom (list A) (list_eq A A_eq) l :=
7    list_eq_correct A A_eq l (list_is_list A (eq_axiom A A_eq) HA l).
```

It is interesting to look at a data type that uses a container such as `rtree`: the induction hypothesis `Pl` given by `rtree_induction` perfectly fits the premise of `list_eq_correct` (line 7).

```
1  Lemma rtree_eq_correct A A_eq : ∀t, is_tree A (eq_axiom A A_eq) t →
2      eq_axiom (rtree A) (rtree_eq A A_eq)
3  :=
4    rtree_induction A (eq_axiom A A_eq) (eq_axiom (rtree A) (rtree_eq Afa))
5      PLeaf
6      (fun l (Pl : is_list (rtree A) (eq_axiom (rtree A) (rtree_eq A A_eq)) l) =>
7       .. list_eq_correct (rtree A) (rtree_eq A A_eq) l Pl ..).
8
9  Lemma rtree_eq_OK A A_eq (HA : ∀a, eq_axiom A A_eq a) t :
10     eq_axiom (rtree A) (rtree_eq A A_eq) t :=
11   rtree_eq_correct A A_eq t (rtree_is_rtree A (eq_axiom A A_eq) HA t).
```

Type checking the terms above does not require any term to be transparent. Actually they are applicative terms, there is no apparently recursive function involved.

Still there is no magic, we just swept the problem under the rug. In order to type check the proof of `rtree_is_rtree` Coq needs to look at the proof term of `list_is_list`:

```
Definition rtree_is_rtree A is_A (His_A : ∀a, is_A a) :=
  fix IH t {struct t} : is_rtree A is_A t :=
    match t with
    | Leaf a => is_Leaf A is_A a (His_A a)
    | Node l => is_Node A is_A l (list_is_list (rtree A) (is_rtree A) IH l)
    end.
```

As we explained in section 2 Coq would reject this term if the body of `list_is_list` was opaque.

Even if we cannot make the problem disappear (without changing the way Coq checks termination), we claim we confined the termination checking issue to the world of reified type information. The transparent proofs of theorems such as `T_is_T` are separate from the other, more relevant, proofs that can hence remain opaque as desired.

4 Elpi: an extension language for Coq

Elpi [6] is a dialect of λProlog [13], a higher order logic programming language. Elpi can be used as an extension language for Coq [19] in order to develop new commands in a programming language that has native support for bound variables.

Coq terms are represented in $\lambda-$tree syntax style [12] (sometimes also called Higher Order Abstract Syntax) reusing the binders of the programming language to represent the ones of Coq. For example, the term `(fun x => fact x)` is represented as `(lam (λx, app["fact",x]))`. We say that `app` and `lam` are object level term constructors standing for iterated (n-ary) application and unary lambda abstraction; `"fact"` is a constant and `x` is a variable bound by `λx,` that is the binder of the programming language. [2]

Programs are organized in clauses that represent both a data base of known facts and a set of rules to derive new facts out of known ones. For example one could use a relation named `eq-db` to link a type to its equality test.

```
eq-db "nat" "nat_eq".
eq-db (app["list", B]) (app["list_eq", B, B_eq]) :- eq-db B B_eq.
```

The first clause is a fact stating that `nat_eq` is the equality test for type `nat`. The second clause is an inference one and reads: the equality test for `(list B)` is `(list_eq B B_eq)` *if* `B_eq` is the equality test for `B`.

The `eq-db` data base can be queried for an equality test for, say, `(list nat)` by writing the goal `(eq-db (app["list", "nat"]) F)` where `F` is a variable to be filled in. By chaining the two clauses Elpi answers `(F = app["list_eq", "nat", "nat_eq"])` that reads back in the Coq syntax as `(list_eq nat nat_eq)`, the desired equality test for `(list nat)`.

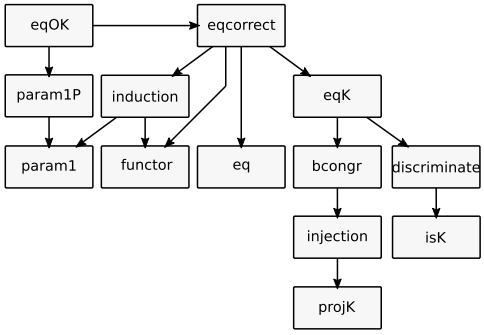
It is worth pointing out that in λProlog the set of clauses is dynamic: a program is allowed to add clauses inside a specific scope (typically the one of a binder) and the runtime collects them when the scope ends. As we will see, this feature is useful when a derivation takes place under an hypothetical context, e.g. when one assumes a parameter `A` and an equality test `A_eq`. No other feature of the Elpi language is relevant to this paper.

Finally, the integration of Elpi in Coq exposes to the extension language primitives to access the logical environment, e.g. to read an inductive data type declaration; to declare a new inductive type; to define a new constant; etc.

[2] Here we simplify a little the embedding and use strings to represent named terms, omitting their nodes: For example `nat`, an inductive type, is actually written `(indt "Coq.Init.Datatypes.nat")`, while `fact`, a defined constant, is written `(const "Coq.Arith.Factorial.fact")`.

5 Anatomy of the derivation

The structure of the derivation is depicted in the following diagram. Each box represents a component deriving a complete term. An arrow from component A to component B tells that the terms generated by B are used by the terms generated by A. The interfaces between these components are indeed types: one can replace the work done by each component with a few hand written terms, if necessary.



The *eq* component is in charge of synthesizing the program performing the equality test. The correctness proof generated by *eqcorrect* goes by induction on the first term of the two being compared and then goes on in a different branch for each constructor K. The property being proved by induction is expressed using `eq_axiom` that, as we will detail in section 5.6 is equivalent to a double implication. The *bcongr* component proves that the property is preserved by equal contexts, that is when the two terms are built using the same constructor. When they are not the program must return false and the equality be false as well: this is shown by *eqK*, that performs the case split on the second term. The no confusion property of constructor is key to this contextual reasoning. *projK* and *isK* generate utility functions that are then used by *injection* and *discriminate* to prove that constructors are injective and different. As we sketched in the previous sections the unary parametricity translation plays a key role in expressing the induction principle. The inductive predicate `is_T` for an inductive type `T` is generated by *param1* while *param1P* shows that terms of type `T` validate `is_T`. *functor* shows that `is_T` is a functor when `T` has parameters. This property is both used to synthesize induction principles and also to combine the pieces together in the correctness proof. The *eqOK* component hides the `is_T` relation from the theorems proved by *eqcorrect* by using the lemmas `T_is_T` proved by *param1P*.

5.1 Equality test

Synthesizing the equality test for a type `T` proceeds as follows. First the test takes in input each type parameter `A` together with an equality test `A_eq`. Then the recursive function takes in input two terms of type `T` and inspects both via pattern matching. Outside the diagonal, where constructors are different, it says `false`. On the diagonal it composes the calls on the arguments of the constructors using boolean conjunction. The code called to compare two arguments depends on their type: If it is `T` then it is a recursive call; if it is a type parameter `A` then we use `A_eq`; if it is another type it uses the corresponding equality test.

Let us take for example the equality test for rose trees:

```
1 Definition rtree_eq A (A_eq : A → A → bool) :=
2   fix rec (t1 t2 : rtree A) {struct t1} : bool :=
3     match t1, t2 with
4     | Leaf a, Leaf b => A_eq a b
5     | Node l, Node s => list_eq (rtree A) rec l s
6     | _, _ => false
7     end.
```

Line 5 calls `list_eq` since the type of `l` and `s` is `(list (rtree A))` and it passes to it `rec` since the type parameter of `list` is `(rtree A)`.

Here is an excerpt of Elpi code used to synthesize the body of the branches:

```
eq-db "A" "A_eq".
eq-db (app["rtree","A"]) "rec".
eq-db (app["list", B]) (app["list_eq", B, B_eq]) :- eq-db B B_eq.
```

The first clause says that `A_eq` is the equality test for type `A`, and is used to build the branch at line 4. The third clause, chained with the second one, combines `list_eq` with `rec` building the branch at line 5. The first two clauses are present only during the derivation of the body of the fixpoint, under the context formed by the type parameter `A`, its equality test `A_eq`, and the recursive call `rec` itself. Once the derivation is complete both clauses are removed from the data base and the following one is permanently added.

```
eq-db (app["rtree", B]) (app["rtree_eq", B, B_eq]) :- eq-db B_eq.
```

5.2 Parametricity

The *param1* component is able to generate the unary parametricity translation of types and terms following [8]. We already gave many examples in section 3. The *param1P* component synthesizes proofs that terms of type `T` validate `is_T` by a trivial structural recursion: constructor `K` is mapped to `is_K`. When `T` is a container we assume the triviality of the property on the type parameter. For example:

```
Definition rtree_is_rtree A (is_A : A → U) : (∀x, is_A x) → ∀t, is_rtree A is_A t.
```

5.3 Functoriality

The *functor* component implements a double service. For non-indexed containers it synthesizes a simple map:

```
Definition list_map A B : (A → B) → list A → list B.
```

The derivation becomes more interesting when the container has indexes, e.g. when the container is a `is_T` inductive predicate. On indexed data types the derivation avoids to map the indexes and consequently all type variables occurring in the types of the indexes. For example, mapping the `is_list` inductive predicate gives:

```
Lemma is_list_funct A P Q : (∀a, P a → Q a) → ∀l, is_list A P l → is_list A Q l.
```

This property corresponds to the functoriality of `is_list` over the property about the type parameter. Note that parameters of arity one, such as `P`, are mapped point wise.

As we did for the `eq-db` data base of equality tests, we can store these maps as clauses and use the data base later on in the *induction* and *eqcorrect* derivations. Here is an excerpt of Elpi code for this data base, that we call `funct-db`:

```
funct-db (app["is_list",A,P]) (app["is_list",A,Q]) (app["is_list_funct",A,P,Q,F]) :-
  funct-db P Q F.
```

Note that the terms involved are "point free", i.e. the first two arguments are terms of arity one, while the third term is of arity two. The identity is written as follows:

```
funct-db P P (lam (λa, lam (λp, p))).
```

This means that when one has a term `a` and a term `(p : P a)`, in order to obtain a term `(q : Q a)` he can query `funct-db` by asking Elpi to fill in `M` in `(funct-db "P" "Q" M)`. If the answer is `(M = f)` then the desired term is obtained by passing `a` and `p` to `f`, that is `(f a p : Q a)`.

5.4 Induction

In order to derive the induction principle for type `T` we first derive its unary parametricity translation `is_T`. The `is_T` inductive predicate has one constructor `is_K` for each constructor `K` of the type `T`. The type of `is_K` relates to the type of `K` in the following way. For each argument `(a : A)` of `K`, `is_K` takes two arguments: `(a : A)` and `(pa : is_A a)`. Finally the type of `(is_K a1 pa1 .. an pan)` is `(is_T (K a1 .. an))`.

The induction principle is synthesized by following these steps:

1. take in input each parameter `A1 is_A .. An is_A` of `is_T`.
2. take in input a predicate `(P : T A1 .. An → U)`.
3. for each constructor `is_K` of type
   `(∀A1 is_A .. An is_A, ∀a1 pa1 .. am pam, is_T A1 is_A .. An is_A (K a1 .. am))`
   take in input an assumption `HK` of type `(∀a1 pa1 .. am pam, P (K a1 .. am))`.
4. take in input `(t : T A1 .. An)`.
5. take in input `(x : is_T A1 is_A .. An is_A t)`.
6. perform recursion on `x` and a case split. Then in each branch
   a. bind all arguments of `is_K`, namely
      `(a1 : A1) (pa1 : is_A1 a1) .. (an : An) (pan : is_An an)`
   b. obtain `qai` by *mapping* the corresponding `pai` (as in `funct-db`, see below).
   c. return `(HK a1 qa1 .. an qan)`

Lets take for example the induction principle for rose trees:

```
1  Definition rtree_induction A is_A P
2     (HLeaf : ∀a, is_A a → P (Leaf A a))
3     (HNode : ∀l, is_list (rtree A) P l → P (Node A l)) :
4   ∀t, is_rtree A is_A t → P t
5  :=
6   fix IH (t: rtree A) (x: is_rtree A is_A t) {struct x}: P t :=
7     match x with
8     | is_Leaf a pa => HLeaf a pa
9     | is_Node l pl => (* pl: is_list (rtree A) (is_rtree A is_A) l *)
10       HNode l (is_list_funct (rtree A) (is_rtree A is_A) P IH l pl)
11    end.
```

Note how, intuitively, the type of `HLeaf` can be obtained from the type of `is_Leaf` by replacing `(is_rtree A is_A)` with `P`.

Finally let us see how the second argument to `HNode` is synthesized. We take advantage of the fact that Elpi is a logic programming language and we query the data base `funct-db` as follows. First we temporarily register the fact that `IH` maps `(is_rtree A is_A)` to `P` obtaining, among others, the following clauses.

```
funct-db (app["is_rtree", "A", "is_A"]) "P" "IH".
funct-db (app["is_list",A,P]) (app["is_list",A,Q]) (app["is_list_funct",A,P,Q,F]) :-
  funct-db P Q F.
```

Then we query `funct-db` as follows:

```
funct-db (app["is_list", app["rtree","A"], app["is_rtree","A","is_A"]])
         (app["is_list", app["rtree","A"], "P"])
         Q.
```

The answer `(Q=app["is_list_funct",app["rtree","A"],app["is_rtree","A","is_A"],"P","IH"])` is exactly the second term we need to pass to `HNode` (once applied to `l` and `P1`, line 10 above).

It is worth pointing out that, for the term to be accepted by the termination checker the map over `is_list` must be transparent.

To sum up the unary parametricity translation gives us the type of the induction principle, up to a trivial substitution. The functoriality property of the inductive predicates obtained by parametricity gives us a way to prove the branches.

5.5 No confusion property

In order to prove that an equality test is correct one has to show the so called "no confusion" property, that is that constructors are injective and disjoint (see for example [11]).

The simplest form of the property of being disjoint is expressed on `bool`:

```
Lemma bool_discr : true = false → ∀T : U, T.
```

This lemma is proved by hand once and for all. What the *isK* component synthesizes is a per-constructor test to be used in order to reduce a discrimination problem on type `T` to a discrimination problem on `bool`. For the rose tree data type *isK* generates:

```
Definition is_Node A (t : rtree A) := match t with Node _ => true | _ => false end.
Definition is_Leaf A (t : rtree A) := match t with Leaf _ => true | _ => false end.
```

The *discriminate* components uses one more trivial fact, `eq_f` [3], in order to assemble these tests together with `bool_discr`.

```
Lemma eq_f T1 T2 (f : T1 → T2) : ∀a b, a = b → f a = f b.
```

From a term `H` of type `(Node l = Leaf a)` the *discriminate* procedure synthesizes:

```
(bool_discr (eq_f (rtree A) (rtree A) (is_Node A) H)) : ∀T : U, T
```

Note that the type of the term `(eq_f .. H)` is `(is_Node A (Node l) = is_Node A (Leaf a))` that is convertible to `(true = false)`, the premise of `bool_discr`.

In order to prove the injectivity of constructors the *projK* component synthesizes a projector for each argument of each constructor. For the `cons` constructor of `list` we get:

```
Definition get_cons1 A (d1 : A) (d2 : list A) (l : list A) : A :=
  match l with nil => d1 | x :: _ => x end.

Definition get_cons2 A (d1 : A) (d2 : list A) (l : list A) : list A :=
  match l with nil => d2 | _ :: xs => xs end.
```

[3] `eq_f` is called `f_equal` in the Coq standard library.

Each projector takes in input default values for each and every argument of the constructor. It is designed to be used by the *injection* procedure as follows. Given a term `H` of type `(x :: xs = y :: ys)` it synthesizes:

```
(eq_f (list A) A        (get_cons1 A x xs) (x :: xs) (y :: ys) H) : x = y
(eq_f (list A) (list A) (get_cons2 A x xs) (x :: xs) (y :: ys) H) : xs = ys
```

These terms are easy to build given that the type of `H` contains the default values to be passed to the projectors. Note that the type of the second term is actually:

```
get_cons2 A x xs (x :: xs) = get_cons2 A x xs (y :: ys)
```

that is convertible to the desired type `(xs = ys)`.

5.6 Congruence

In the definition of `eq_axiom` we use the `reflect` predicate [10]. It is a sort of if-and-only-if specialized to link a proposition and a boolean test. It is defined as follows:

```
Inductive reflect (P : U) : bool → U :=
| ReflectT (p : P) : reflect P true
| ReflectF (np : P → False) : reflect P false.
```

In our case the shape of `P` is always an equation between two terms of an inductive type, i.e. constructors. When the same constructor occurs in both sides, as in `(k x1.. xn = k y1.. y2)`, the equality test discards `k` and proceeds on each `(xi = yi)`. The *bcongr* component synthesizes lemmas helping to prove the correctness of this step. For example:

```
Lemma list_bcongr_cons A :
   ∀(x y : A) b, reflect (x = y) b →
   ∀(xs ys : list A) c, reflect (xs = ys) c →
  reflect (x :: xs = y :: ys) (b && c)

Lemma rtree_bcongr_Leaf A (x y : A) b :
  reflect (x = y) b → reflect (Leaf A x = Leaf A y) b

Lemma rtree_bcongr_Node A (l1 l2 : list (rtree A)) b :
  reflect (l1 = l2) b → reflect (Node A l1 = Node A l2) b
```

Note that these lemmas are not related to the equality test specific to the inductive type. Indeed they deal with the `reflect` predicate, but not with the `eq_axiom` predicate that we use every time we talk about equality tests.

The derivation goes as follows: if any of the premises is false, then the result is proved by `ReflectF` and the injectivity of constructors. If all premises are `ReflectT` their argument, an equation, can be used to rewrite the conclusion.

```
1  Lemma list_bcongr_cons A
2      (x y : A) b (hb : reflect (x = y) b)
3      (xs ys : list A) c (hc : reflect (xs = ys) c) :
4    reflect (x :: xs = y :: ys) (b && c) :=
5  match hb, hc with
6  | ReflectT eq_refl, ReflectT eq_refl => ReflectT eq_refl
7  | ReflectF (e : x = y → False), _ =>
8      ReflectF (fun H : x :: xs = y :: ys =>
9                  e (eq_f (list A) A (get_cons1 A x xs) (x :: xs) (y :: ys) H))
10 | _, ReflectF e =>
11     ReflectF .. (e (eq_f .. (get_cons2 ..) ..) ..) ..
12 end.
```

The elimination of `hb` and `hc` substitutes `b` and `c` by either `true` or `false`. In the branch at line 6 the boolean expression is hence `(true && true)` while the proposition is `(x :: xs = x :: xs)` given that the two equations `(x = y)` and `(xs = ys)` were eliminated as well.

The argument of `e` at line 9 is the term generated by the *injection* component. The branch at line 11, covering the case where the heads are equal but the tails different, is very close to lines 8 and 9 but for the fact that the projector for the second argument of `cons` is used, instead of the projection for the first one.

There are other ways one could have expressed these lemmas, for example by not mentioning the `cons` constructor explicitly but rather an abstract function `k` known to be injective on the first and second argument. Even if we find this presentation more appealing on paper, in practice we found no advantage and we hence opted for the current approach.

bcongr gives us lemmas to propagate equality and inequality only under the same constructor. *eqK* complements this work by proving `eq_axiom` also when the constructors differ.

Recall that the induction principle does a case split on one term, the first one of the two being compared. *eqK* generates a lemma for each constructor, to be used in the corresponding branch of the induction, that performs the case split on the second term being compared. This is the lemma generated for `Node`:

```
Lemma rtree_eq_axiom_Node A (A_eq : A → A → bool) l1 :
    eq_axiom (list (rtree A)) (list_eq (rtree A) (rtree_eq A A_eq)) l1 →
  eq_axiom (rtree A) (rtree_eq A A_eq) (Node A l1)
:=
  fun H (t2 : rtree A) =>
  match t2 with
  | Leaf n =>
    ReflectF (fun abs : Node A l1 = Leaf A n =>
      bool_discr (eq_f (rtree A) bool (is_Node A) (Node A l1) (Leaf A n) abs) False)
  | Node l2 =>
      rtree_bcongr_Node A l1 l2 (list_eq (rtree A) (rtree_eq A A_eq) l1 l2) (H l2)
end.
```

Note that the code for the first branch is what *discriminate* synthesizes; while the code in the second branch is what *bcongr* generates.

5.7 Correctness

The *eqcorrect* component combines the induction principle generated by *induction* with the case split on the second term provided by *eqK*.

Let's recall the type of the correctness lemma for `list_eq`, of the induction principle and then let's analyse the proof of `rtree_eq_correct`:

```
Lemma list_eq_correct A (fa : A → A → bool) l,
    is_list A (eq_axiom A fa) l →
  eq_axiom (list A) (list_eq A fa) l.

Definition rtree_induction A is_A P
    (HLeaf : ∀y, is_A y → P (Leaf A y))
    (HNode : ∀l, is_list (rtree A) P l → P (Node A l)) :
  ∀t, is_rtree A is_A t → P t.

Lemma rtree_eq_axiom_Node A (f : A → A → bool) l1 :
    eq_axiom (list (rtree A)) (list_eq (rtree A) (rtree_eq A f)) l1 →
  eq_axiom (rtree A) (rtree_eq A f) (Node A l1).
```

The proof is a rather straightforward application of the induction principle to the property

```
eq_axiom (rtree A) (rtree_eq A fa)
```

Each branch is then proved by the corresponding lemma generated by *eqK* with only one caveat: one may need to adapt the induction hypothesis, `Pl` here, in order to make it fit the premise of the lemma generated by *eqK*. In this specific case the "adaptor" is `list_eq_correct`.

```
Lemma rtree_eq_correct A (fa : A → A → bool) :=
  rtree_induction A (eq_axiom A fa)
(*P*)     (eq_axiom (rtree A) (rtree_eq A fa))
(*HLeaf*) (rtree_eq_axiom_Leaf A fa)
(*HNode*) (fun l (Pl : is_list (rtree a) (eq_axiom (rtree a) (rtree_eq a fa)) l) =>
           rtree_eq_axiom_Node A fa l (list_eq_correct (rtree a) (rtree_eq a fa) l Pl)).
```

Logic programming provides a natural way to synthesize the adaptor. We load in the data base all the correctness proofs synthesized so far, as follows:

```
funct-db (app["is_list", A, is_A])
         (app["eq_axiom", app["list", A], app["list_eq", A, A_eq]]) R :-
  R = (app["list_eq_correct", A, A_eq]),
  funct-db is_A (app["eq_axiom", A, A_eq]).
```

This clause simply gives an operational reading to the type of `list_eq_correct`: the conclusion is true if the premise is. The only cleverness is to separate the premise in two parts, being a `(list A)` with property `is_A` and have `is_A` be a sufficient condition to prove that `A_eq` is correct. In this way clauses compose better: Search peels off just one type constructor at a time. Indeed we extend the `funct-db` predicate, instead of building a new one just for correctness lemmas, because functoriality lemmas are sometimes needed in addition to the correctness ones. Take for example this simple data type of a histogram.

```
Inductive histogram := Columns (bars : list nat).

Lemma histogram_induction (P : histogram → Type) :
    (∀l, is_list nat is_nat l → P (Columns l)) →
  ∀h, is_histogram h → P h.
```

Now look at the lemma synthesized by *eqK* for the `Columns` constructor.

```
Lemma histogram_eq_axiom_Columns l :
    eq_axiom (list nat) (list_eq nat nat_eq) l →
  ∀h, eq_axiom_at histogram histogram_eq (Columns l) h.
```

```
Lemma histogram_eq_correct h : eq_axiom histogram histogram_eq h :=
  histogram_induction
    (eq_axiom histogram histogram_eq)
    (fun l (Pl : is_list nat is_nat l) =>
       histogram_eq_axiom_Columns
         l (list_eq_correct nat nat_eq
              l (is_list_funct nat is_nat (eq_axiom nat nat_eq) nat_eq_correct l Pl))).
```

Note that the type of `Pl` is `(is_list nat is_nat)` and that it needs to be adapted to match `(is_list nat (eq_axiom nat nat_eq))`. The correctness lemma for `nat_eq`, namely `nat_eq_correct` of type `(∀n, is_nat n → eq_axiom nat nat_eq n)`, cannot be used directly but must undergo the `is_list_funct` functor.

5.8 eqOK

The last derivation hides the `is_T` predicate to the final user by combining the output of *eqcorrect* and *param1P*.

```
Lemma list_eq_correct A A_eq :
  ∀l, is_list A (eq_axiom A A_eq) l → eq_axiom (list A) (list_eq A A_eq) l.

Lemma list_eq_OK A A_eq A_eq_OK l : eq_axiom (list A) (list_eq A A_eq) l :=
  list_eq_correct A A_eq l (list_is_list A (eq_axiom A A_eq) A_eq_OK).
```

Both lemmas are needed. The former composes well and is needed if one defines a type using lists as a container. The latter is what the user needs in order to work with lists.

5.9 Assessment

The code is quite compact thanks to the fact that the programming language is very high level and that its programming paradigm is a good fit for this application.

On the average each components is about 200 lines of code. Simpler derivations like *projK*, *isK* or even *param1P* are under 100 lines.

Debugging this kind of code did not pose particular difficulties. The typical error results in the generated term being ill-typed. In that case the Coq type checker could be used to identify the culprit. Given how small the derivations are, it was simple to identify the lines generating the offending subterm.

The time required to design and develop the entire procedure amounts to approximatively six months, but spanned over more than one and a half year: most of the time has been spent improving the integration of Elpi in Coq in response to the experience gathered on this work. At the time of writing the Elpi integration in Coq does not support mutual inductive types, universe polymorphic definitions and primitive projections.

All derivations support polynomial types. Some derivations also support index data, e.g. *eq* is able to synthesize an equality test for vectors. Most of the derivations for contextual reasoning, such as *eqK* and *bcongr* do not support indexes.

6 Related work

Systems similar to Coq [20], e.g. Matita [2], Lean [5] and Isabelle [14] all generate induction principles automatically, with the exception of Agda [15], and some of them also the no confusion properties.

To our knowledge Isabelle is the only system that generates sensible induction principles and proved equality tests when containers are involved. As described in [4] the (co)datatype package is built on top of Bounded Natural Functors [21], a notion that makes the construction of (co)datatypes in Higher Order Logic compositional. Our starting point is very different since Coq, and type theory in general, internalizes the definitional mechanism for (co)datatypes. As a consequence a package like the one described in this paper cannot change it but only work around its eventual limitations. In particular the way Coq checks recursive functions for termination is a fixed, syntactic, non modular, criteria for which some alternatives have been studied (see for example [3, 16]) but never implemented. The non modular criteria applies to induction principles as well, since they are proved using recursion. It is a strength of the construction described in this paper to recover some modularity and hence be able to synthesize mechanically most of what [4] is able to synthesize.

Most Interactive Theorem Provers come with simple forms of Prolog-like automation, usually in the form of Type Classes. The user typically resorts to that in order to perform some of the inductive reasoning one needs in order to synthesize code in a type directed way. To our knowledge no ready-to-use package to synthesize equality tests and their proofs was written this way.

Some systems, notably Lean, come with a whole round meta programming framework. Still, to our knowledge, the primary application is the development of proof commands, not program/proof synthesis, in spite of the stunning similarity.

Coq provides two mechanisms strictly related to this work. The `Scheme Equality` command generates for a type `T` the code for the equality test (`T_eqb`) and a proof that equality is decidable on `T`. The proof internally uses the equality test, but its type does not:

```
T_eq_dec : ∀x y : T, {x = y} + {x <> y}
```

By unfolding the proof term, that is transparent, it should be possible to recover the fact that `T_eqb` is a correct equality test. Data types defined using containers are not supported. The `decide equality` tactic requires the user to start a lemma with a statement as the one depicted above. The tactic only performs one (case split) step and has to be iterated by hand. It does not remember which equalities were proved decidable before, it is up to the user to eventually share code. The proof term generated is, in a type theoretic sense, a program even if its code mixes the comparison test with its correctness proof. This proof is fully transparent, and inlines all the contextual reasoning steps such as injection and discrimination. As a result the term is very large and computationally heavy when run within Coq.

In the programming language world derivation is much more developed. The dominant approach is to provide some meta programming facilities, e.g. by providing a syntax to the declaration of types and then use the programming language itself to write derivations [17] that run at compile time as compiler plugins. Our approach is similar in a sense, since we work at the meta level on the syntax of types (and terms), but it is also very different since we pick a different programming language for meta programming. In particular we choose a very high level one that makes our derivations very concise and hides uninteresting details such as the representation of bound variables. The derivation described in the paper is the result of many failed attempts and we believe that the high level nature of the programming language we chose played an important role in the exploratory phase.

The link between the unary parametricity translation, also called predicate lifting, and induction principles was independently remarked by Kaposi and Kovács in [7].

7 Conclusion

We described a technique to derive stronger induction principles for Coq data types built using containers. We use the unary parametricity translation of a data type in order to fuel its induction principle, to thread an invariant on the contained when used as a container and finally to confine the modularity problems stemming from the termination check implemented in Coq. Finally we provide a Coq package deriving correct equality tests for polynomial inductive data types.

It is work in progress to extend the derivation to inductive types with decidable indexes. Preliminary work hints that indexes of base types such as `nat` pose no problem. On the contrary when indexes mention containers, that admit a decidable equality only if their contained does, the *param1P* component gets substantially more complex. In particular some notions of Homotopy Type Theory come in to play. For example the notion of being provable on the entire domain such as `(∀a : A, P a)` → `(∀t : T A, is_T A P t)` seems to require to be strengthened using the notion of contractibility (that is, the property should hold and its proof be unique), in order for the construction to compose well.

We also look forward to let the user tune the derivation process by annotating the type declarations. For example the user may want to skip certain arguments when generating the equality test, such as the integer describing the length of a sub vector in the `cons` constructor. The resulting equality test surely requires some user intervention in order to be proved correct, but it features a better computational complexity.

Finally, adding other derivations to the package seems appealing. For example the interface next to `eqType` in the hierarchy used in the Mathematical Component library is the one of countable types, i.e. types in bijection with natural numbers. The interface requires, roughly, a serialization function to another countable type, a tedious task that could be made automatic.

We are grateful to Maxime Denes and Cyril Cohen for the many discussions shedding light on the subject. We thank Cyril Cohen for writing the code of *param2* (binary parametricity translation), out of which *param1* was easily obtained. We also thank Damien Rouhling, Laurent Théry and Laurence Rideau for proofreading the paper. Finally we are indebted to Luc Chabassier for working on an early prototype of Elpi on the synthesis of equality tests: an experiment that convinced the author it was actually doable.

References

1 Andreas Abel. Type-based termination of generic programs. *Sci. Comput. Program.*, 74(8):550–567, 2009. doi:10.1016/j.scico.2008.01.004.

2 Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The Matita Interactive Theorem Prover. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, pages 64–69, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

3 Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. CIC^: Type-Based Termination of Recursive Definitions in the Calculus of Inductive Constructions. In *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia, November 13-17, 2006, Proceedings*, pages 257–271, 2006. doi:10.1007/11916277_18.

4 Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Truly Modular (Co)datatypes for Isabelle/HOL. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, pages 93–110, Cham, 2014. Springer International Publishing.

5 Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing.

6 Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast, Embeddable, λProlog Interpreter. In *Proceedings of LPAR*, Suva, Fiji, November 2015. URL: https://hal.inria.fr/hal-01176856.

7 Ambrus Kaposi and András Kovács. A Syntax for Higher Inductive-Inductive Types. In *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*, pages 20:1–20:18, 2018. doi:10.4230/LIPIcs.FSCD.2018.20.

8 Chantal Keller and Marc Lasson. Parametricity in an Impredicative Sort. In Patrick Cégielski and Arnaud Durand, editors, *CSL - 26th International Workshop/21st Annual Conference of the EACSL - 2012*, volume 16 of *CSL*, pages 381–395, Fontainebleau, France, September 2012. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CSL.2012.399.

9 Assia Mahboubi and Enrico Tassi. Canonical Structures for the Working Coq User. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, pages 19–34, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

10 Assia Mahboubi and Enrico Tassi. *Mathematical Components*. draft, v1-183-gb37ad7, 2018.

11 Conor McBride, Healfdene Goguen, and James McKinna. A Few Constructions on Constructors. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *Types for Proofs and Programs*, pages 186–200, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

12 Dale Miller. Abstract Syntax for Variable Binders: An Overview. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz

Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic — CL 2000*, pages 239–253, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

13 Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012. doi:10.1017/CBO9781139021326.

14 Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

15 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

16 Jorge Luis Sacchini. *On type-based termination and dependent pattern matching in the calculus of inductive constructions*. Theses, École Nationale Supérieure des Mines de Paris, June 2011. URL: https://pastel.archives-ouvertes.fr/pastel-00622429.

17 Tim Sheard and Simon Peyton Jones. Template Meta-programming for Haskell. *SIGPLAN Not.*, 37(12):60–75, December 2002. doi:10.1145/636517.636528.

18 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.

19 Enrico Tassi. Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect). CoqPL, January 2018. URL: https://hal.inria.fr/hal-01637063.

20 The Coq Development Team. The Coq Proof Assistant, version 8.8.0, April 2018. doi:10.5281/zenodo.1219885.

21 Dmitry Traytel, Andrei Popescu, and Jasmin C. Blanchette. Foundational, Compositional (Co)Datatypes for Higher-Order Logic: Category Theory Applied to Theorem Proving. In *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*, LICS '12, pages 596–605, Washington, DC, USA, 2012. IEEE Computer Society. doi:10.1109/LICS.2012.75.

22 Philip Wadler. Theorems for Free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 347–359, New York, NY, USA, 1989. ACM. doi:10.1145/99370.99404.

Complete Non-Orders and Fixed Points

Akihisa Yamada
National Institute of Informatics, Tokyo, Japan
akihisayamada@nii.ac.jp

Jérémy Dubut
National Institute of Informatics, Tokyo, Japan
Japanese-French Laboratory for Informatics, Tokyo, Japan
dubut@nii.ac.jp

Abstract

In this paper, we develop an Isabelle/HOL library of order-theoretic concepts, such as various completeness conditions and fixed-point theorems. We keep our formalization as general as possible: we reprove several well-known results about complete orders, often without any property of ordering, thus complete non-orders. In particular, we generalize the Knaster–Tarski theorem so that we ensure the existence of a quasi-fixed point of monotone maps over complete non-orders, and show that the set of quasi-fixed points is complete under a mild condition – attractivity – which is implied by either antisymmetry or transitivity. This result generalizes and strengthens a result by Stauti and Maaden. Finally, we recover Kleene's fixed-point theorem for omega-complete non-orders, again using attractivity to prove that Kleene's fixed points are least quasi-fixed points.

2012 ACM Subject Classification Theory of computation → Interactive proof systems

Keywords and phrases Order Theory, Lattice Theory, Fixed-Points, Isabelle/HOL

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.30

Related Version Akihisa Yamada and Jérémy Dubut. Complete Non-Orders and Fixed Points. *Archive of Formal Proofs*, 2019. (`http://isa-afp.org/entries/Complete_Non_Orders.html`, Formal proof development).

Funding The authors are supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST; J. Dubut is also supported by Grant-in-aid No. 19K20215, JSPS.

1 Introduction

The main driving force towards mechanizing mathematics using proof assistants has been the reliability they offer, exemplified prominently by [10], [12], [14], etc. In this work, we utilize another aspect of proof assistants: they are also engineering tools for developing mathematical theories. In particular, we choose Isabelle/JEdit [22], a *very* smart environment for developing theories in Isabelle/HOL [17]. There, the proofs we write are checked "as you type", so that one can easily refine proofs or even theorem statements by just changing a part of it and see if Isabelle complains or not. Sledgehammer [7] can often automatically fill relatively small gaps in proofs so that we can concentrate on more important aspects. Isabelle's counterexample finders [3, 6] should also be highly appreciated, considering the amount of time one would spend trying in vain to prove a false claim.

In this paper, we formalize order-theoretic concepts and results in Isabelle/HOL. Here we adopt an *as-general-as-possible* approach: most results concerning order-theoretic completeness and fixed-point theorems are proved without assuming the underlying relations to be orders (non-orders). In particular, we provide the following:

- Various completeness results that generalize known theorems in order theory: Actually most relationships and duality of completeness conditions are proved without *any* properties of the underlying relations.

- Existence of fixed points: We show that a relation-preserving mapping $f : A \to A$ over a complete non-order $\langle A, \sqsubseteq \rangle$ admits a *quasi-fixed point* $f(x) \sim x$, meaning $x \sqsubseteq f(x) \wedge f(x) \sqsubseteq x$. Clearly if $\sqsubseteq$ is antisymmetric then this implies the existence of fixed points $f(x) = x$.
- Completeness of the set of fixed points: We further show that if $\sqsubseteq$ satisfies a mild condition, which we call *attractivity* and which is implied by either transitivity or antisymmetry, then the set of quasi-fixed points is complete. Furthermore, we also show that if $\sqsubseteq$ is antisymmetric, then the set of *strict* fixed points $f(x) = x$ is complete.
- Kleene-style fixed-point theorems: For an ω-complete non-order $\langle A, \sqsubseteq \rangle$ with a bottom element $\bot \in A$ (not necessarily unique) and for every ω-continuous map $f : A \to A$, a supremum exists for the set $\{f^n(\bot) \mid n \in \mathbb{N}\}$, and it is a quasi-fixed point. If $\sqsubseteq$ is attractive, then the quasi-fixed points obtained this way are precisely the least quasi-fixed points.

We remark that all these results would have required much more effort than we spent (if possible at all), if we were not with the aforementioned smart assistance by Isabelle. Our workflow was often the following: first we formalize existing proofs, try relaxing assumptions, see where proof breaks, and at some point ask for a counterexample.

The formalization is available in the Archive of Formal Proofs.

Related Work

Many attempts have been made to generalize the notion of completeness for lattices, conducted in different directions: by relaxing the notion of order itself, removing transitivity (pseudo-orders [19]); by relaxing the notion of lattice, considering minimal upper bounds instead of least upper bounds (χ-posets [15]); by relaxing the notion of completeness, requiring the existence of least upper bounds for restricted classes of subsets (e.g., directed complete and ω-complete, see [8] for a textbook). Considering those generalizations, it was natural to prove new versions of classical fixed-point theorems for maps preserving those structures, e.g., existence of least fixed points for monotone maps on (weak chain) complete pseudo-orders [5, 20], construction of least fixed points for ω-continuous functions for ω-complete lattices [16], (weak chain) completeness of the set of fixed points for monotone functions on (weak chain) complete pseudo-orders [18].

Concerning Isabelle formalization, one can easily find several formalizations of complete partial orders or lattices in Isabelle's standard library. They are, however, defined on partial orders, either in form of classes or locales, and thus not directly reusable for non-orders. Nevertheless we tried to make our formalization compatible with the existing ones, and various correspondences are ensured in the Isabelle source.

2 Preliminaries

This work is based on Isabelle 2019. In Isabelle/HOL, R :: 'a ⇒ 'a ⇒ bool means a binary predicate R, by which we represent a binary relation $R \subseteq A \times A$. Here A is the universe of the type variable 'a, in Isabelle's syntax, UNIV :: 'a set. Type annotations ":: _" are omitted unless they are necessary. We call the pair $\langle A, \sqsubseteq \rangle$ of a set A and a binary relation ($\sqsubseteq$) over A a *related set*. One could also call it a *graph* or an *abstract reduction system*, but then some terminology like "complete" become incompatible.

To make our library *as general as possible*, we avoid using the order symbol $\leq$, which is fixed by the class mechanism of Isabelle/HOL. Instead we make the relation of concern explicit as an argument, sometimes called the *dictionary-passing* style [11]. On one hand

this design choice adds a notational burden, but on the other hand it allows instantiating obtained results to arbitrary relations over a type, for which the class mechanism fixes one ordering. In the formalization we also import our results into the class hierarchy.

A map $f : I \to A$ over related sets from $\langle I, \preceq \rangle$ to $\langle A, \sqsubseteq \rangle$ is *relation preserving*, or *monotone*, if $i \preceq j$ implies $f(i) \sqsubseteq f(j)$. For this property there already exists a definition in the standard Isabelle library:

```
monotone (⪯) (⊑) f ⟷ (∀ i j. i ⪯ j ⟶ f i ⊑ f j)
```

Hereafter, in our Isabelle code, we use symbols (⊑) denoting a variable of type 'a ⇒ 'a ⇒ bool, and (⪯) denoting a variable of type 'i ⇒ 'i ⇒ bool. More precisely, statements and definitions using these symbols are made in a context such as

```
context fixes less_eq :: "'a ⇒ 'a ⇒ bool" (infix "⊑" 50)
```

For clarity, we present definitions, e.g., of predicates for being upper/lower bounds and greatest/least elements, as

```
definition "bound (⊑) X b ≡ ∀x ∈ X. x ⊑ b"
definition "extreme (⊑) X e ≡ e ∈ X ∧ (∀x ∈ X. x ⊑ e)"
```

making the relation (⊑) of concern as an explicit parameter. Note that we chose such constant names that do not suggest which side is greater or lower. The least upper bounds (suprema) and greatest lower bounds (infima) are thus uniformly defined as follows.

```
abbreviation "extreme_bound (⊑) X ≡ extreme (⊒) {b. bound (⊑) X b}"
```

Hereafter, we write (⊒) for $(\sqsubseteq)^-$, which is also an abbreviation:

```
abbreviation "(⊑)⁻ x y ≡ y ⊑ x"
```

We can already prove some useful lemmas. For instance, if $f : I \to A$ is relation preserving and $C \subseteq I$ has a greatest element $e \in C$, then $f(e)$ is a supremum of the image $f(C)$. Note here that no assumption is imposed on the relations $\preceq$ and $\sqsubseteq$.

```
lemma monotone_extreme_imp_extreme_bound:
  assumes "monotone (⪯) (⊑) f" and "extreme (⪯) C e"
  shows "extreme_bound (⊑) (f ` C) (f e)"
```

2.1 Locale Hierarchy of Relations

We now define basic properties of binary relations, in form of *locales* [13, 2]. Isabelle's locale mechanism allows us to conveniently manage notations, assumptions and facts. For instance, we introduce the following locale to fix a relation parameter and use infix notation.

```
locale less_eq_syntax = fixes less_eq :: "'a ⇒ 'a ⇒ bool" (infix "⊑" 50)
```

The most important feature of locales is that we can give assumptions on parameters. For instance, we define a locale for reflexive relations as follows.

```
locale reflexive = less_eq_syntax + assumes refl[iff]: "x ⊑ x"
```

This declaration defines a new predicate "reflexive", with the following defining equation:

```
theorem reflexive_def: "reflexive (⊑) ≡ ∀x. x ⊑ x"
```

One may doubt that such a simple assumption deserves a locale not just the definition. Nevertheless, we have some useful lemmas already, for instance:

```
lemma (in reflexive) extreme_singleton[simp]: "extreme (⊑) {a} b ⟷ a = b"
lemma (in reflexive) extreme_bound_singleton[iff]: "extreme_bound (⊑) {a} a"
```

Similarly we define transitivity and antisymmetry:

```
locale transitive = less_eq_syntax + assumes trans[trans]: "x ⊑ y ⟹ y ⊑ z ⟹ x ⊑ z"
locale antisymmetric = less_eq_syntax +
  assumes antisym[dest]: "a ⊑ b ⟹ b ⊑ a ⟹ a = b"
```

It is straightforward to have locales that combine the above assumptions. Some famous combinations are *quasi-orders* for reflexive and transitive relations and *partial orders* for antisymmetric quasi-order.

```
locale quasi_order = reflexive + transitive
locale partial_order = quasi_order + antisymmetric
```

Less known, but still a convenient assumption is being a *pseudo-order*, coined by Skala [19] for reflexive and antisymmetric relations. There, the supremum of a singleton set $\{x\}$ uniquely exists – x itself.

```
locale pseudo_order = reflexive + antisymmetric

lemma (in pseudo_order) extreme_bound_singleton_eq[simp]:
  "extreme_bound (⊑) {x} y ⟷ x = y" by auto
```

It is clear that a partial order is also a pseudo-order, which is stated by the following *sublocale* declaration. Afterwards facts proved in pseudo_order will be automatically available in partial_order.

```
sublocale partial_order ⊆ pseudo_order..
```

Although these combinations are sufficient for the rest of this paper, we also present all locales combining these basic properties and their relationships in Fig. 1.

3 Completeness of Non-Orders

Here we formalize various order-theoretic completeness conditions in Isabelle. Order-theoretic completeness demands certain subsets of elements to admit suprema or infima. The strongest completeness requires that any subset of elements has suprema and infima.

```
locale complete = less_eq_syntax + assumes "Ex (extreme_bound (⊑) X)"
```

The above assumption only requires suprema (if the right-hand side of $\sqsubseteq$ is seen greater) but not infima, in Isabelle, "Ex (extreme_bound (⊒) X)". This is a well-known consequence in complete lattices, and luckily the proof does not rely on any property of orders. Hence we can declare the following sublocale:

```
sublocale complete ⊆ dual: complete "(⊒)"
proof
  fix X :: "'a set"
  obtain s where "extreme_bound (⊑) {b. bound (⊒) X b}" s using complete by auto
  then show "Ex (extreme_bound (⊒) X)" by (intro exI[of _ s] extreme_boundI, auto)
qed
```

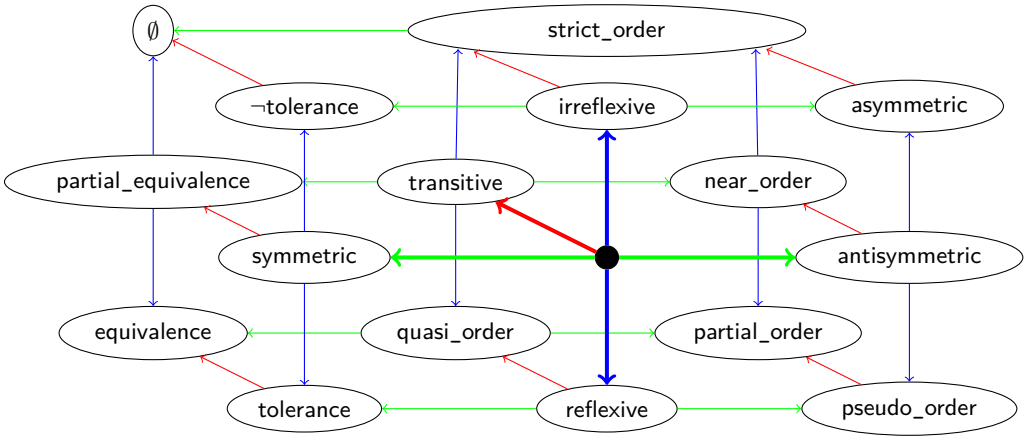


Figure 1 Combinations of basic properties. The black dot around the center represents arbitrary binary relations, and the five outgoing arrows indicate atomic assumptions. We do not present the combination of reflexive and irreflexive, which is empty, and one of symmetric and antisymmetric, which is a subset of equality. Node "¬tolerance" indicates the negated relation is tolerance, and "∅" is the empty relation.

Afterwards, a theorem named xxx proved in locale complete will be available in its dual form as dual.xxx.

Let us mention another strong completeness condition: every nonempty subset of elements has a supremum. This condition is called *semicompleteness*, cf. [4, Chapter 6].

locale semicomplete = less_eq_syntax +
  assumes "X ≠ {} ⟹ Ex (extreme_bound (⊑) X)"

However, semicompleteness fails to be self-dual. Instead, duality holds for a slightly weaker, but highly important completeness condition, *conditional completeness* or *Dedekind completeness*, asserting that any nonempty bounded set has a supremum.

locale conditionally_complete = less_eq_syntax +
  assumes "Ex (bound (⊑) X) ⟹ X ≠ {} ⟹ Ex (extreme_bound (⊑) X)"

sublocale conditionally_complete ⊆ dual: conditionally_complete "(⊒)"

Let us also mention a very weak form of completeness. A related set $\langle A, \sqsubseteq \rangle$ is called *bounded* if there is a "top" element $\top \in A$, a greatest element in A. Note that there might be multiple tops if $(\sqsubseteq)$ is not antisymmetric.

locale bounded = less_eq_syntax + **assumes** "∃t. ∀x. x ⊑ t"

This notion can be also seen as a completeness condition, since it is equivalent to saying that the universe has a supremum.

lemma bounded_iff_UNIV_complete: "bounded (⊑) ⟷ Ex (extreme_bound (⊑) UNIV)"

Since a top element is a bound of any subset of elements, a conditionally complete relation is semicomplete if (and only if) it is bounded.

proposition semicomplete_iff_conditionally_complete_bounded:
  shows "semicomplete (⊑) ⟷ conditionally_complete (⊑) ∧ bounded (⊑)"

The dual notion of bounded is called *pointed*. There, a least element is called a "bottom" element, and serves as a supremum of the emptyset. The dual form of the above proposition, together with the duality of conditional completeness means that, (⊒) is semicomplete if and only if (⊑) is pointed conditionally complete. The latter means that every bounded set, including the empty set, has a supremum – the notion known as "bounded complete".

proposition bounded_complete_iff_dual_semicomplete:
"bounded_complete (⊑) ⟷ semicomplete (⊒)"

3.1 Lattice-Like Completeness

One of the most well-studied notion of completeness would be the semilattice condition: every pair of elements x and y has a supremum $x \sqcup y$ (not necessarily unique if the underlying relation is not antisymmetric).

locale pair_complete = less_eq_syntax + **assumes** "Ex (extreme_bound (⊑) {x,y})"

It is well known that in a semilattice, i.e., a pair-complete partial order, every finite nonempty subset of elements has a supremum. We prove the result assuming transitivity, but only that.

locale finite_complete = less_eq_syntax +
assumes "finite X ⟹ X ≠ {} ⟹ Ex (extreme_bound (⊑) X)"

locale trans_semilattice = transitive + pair_complete

sublocale trans_semilattice ⊆ finite_complete

Proof. The proof is an easy induction on the finite set X. Only a care is taken for the case where X is singleton $\{x\}$; then x may fail to be a supremum of itself, as we do not have reflexivity. Instead we find a supremum via that of the pair of x and x. ◀

3.2 Directed Completeness

Directed completeness is an important notion in domain theory [1], asserting that every nonempty directed set has a supremum. Here, a set X is *directed* if any pair of two elements in X has a bound in X.

definition "directed (⊑) X ≡ ∀x ∈ X. ∀y ∈ X. ∃z ∈ X. x ⊑ z ∧ y ⊑ z"

locale directed_complete = less_eq_syntax +
assumes "directed (⊑) X ⟹ X ≠ {} ⟹ Ex (extreme_bound (⊑) X)"

The image of a relation-preserving map preserves directed sets.

lemma monotone_directed_image:
assumes "monotone (≼) (⊑) f" **and** "directed (≼) D" **shows** "directed (⊑) (f ‘ D)"

Gierz et al. [9] showed that a directed complete partial order is semicomplete if and only if it is also a semilattice. We generalize the claim so that the underlying relation is only transitive.

proposition (**in** transitive) semicomplete_iff_directed_complete_pair_complete:
shows "semicomplete (⊑) ⟷ directed_complete (⊑) ∧ pair_complete (⊑)"

Proof. The $\longrightarrow$ direction is trivial. For the other direction, consider a nonempty set X. We collect all suprema of every nonempty finite subset Y of X into a set S:

$$S = \{x.\ \exists Y \subseteq X.\ \text{finite } Y \wedge Y \neq \{\} \wedge \text{extreme\_bound } (\sqsubseteq)\ Y\ x\}$$

Then S is nonempty since there exists $x \in X$ and a supremum for $\{x\}$ is in S. Next we show that S is directed as follows. Any $y, z \in S$ are suprema of corresponding finite sets $Y \subseteq X$ and $Z \subseteq X$. Since $Y \cup Z$ is finite we get a supremum w of $Y \cup Z$ in S. It is easy to show that w is an upper bound of y and z.

Since $(\sqsubseteq)$ is directed complete, we obtain a supremum s for S. Then s is a supremum of X; here we only show that s is a bound of X. For any $x \in X$ we have a supremum x' of $\{x\}$ in S, and thus we have $x' \sqsubseteq s$. As $x \sqsubseteq x'$ by transitivity we conclude $x \sqsubseteq s$. ◀

The last argument in the above proof requires transitivity, but if we had reflexivity then x itself is a supremum of $\{x\}$ (see lemma extreme_bound_singleton) and so $x \sqsubseteq s$ would be immediate. Thus we can replace transitivity by reflexivity, but then pair-completeness does not imply finite completeness. We obtain the following result.

proposition (**in** reflexive) semicomplete_iff_directed_complete_finite_complete:
shows "semicomplete $(\sqsubseteq)$ $\longleftrightarrow$ directed_complete $(\sqsubseteq)$ $\wedge$ finite_complete $(\sqsubseteq)$"

We also tried to strengthen the above result by replacing finite completeness by pair completeness, but at the time of writing, the question is left open. We remark that, at least, Nitpick did not find a counterexample.

4 Knaster–Tarski-Style Fixed-Point Theorems

Given a monotone map $f : A \to A$ on a complete lattice $\langle A, \sqsubseteq \rangle$, the Knaster–Tarski theorem [21] states that

1. f has a fixed point in A, and
2. the set of fixed points forms a complete lattice.

Stauti and Maaden [20] generalized statement (1) where $\langle A, \sqsubseteq \rangle$ is a complete *trellis* – a complete pseudo-order – relaxing transitivity. They also proved a restricted version of (2), namely there exists a least (and by duality a greatest) fixed point in A.

In the following Section 4.1 we further generalize claim (1) so that any complete relation admits a *quasi-fixed point* $f(x) \sim x$, that is, $f(x) \sqsubseteq x$ and $x \sqsubseteq f(x)$. Quasi-fixed points are fixed points for antisymmetric relations; hence the Stauti–Maaden theorem is further generalized by relaxing reflexivity.

In Section 4.2 we also generalize claim (2) so that only a mild condition, which we call *attractivity*, is assumed. In this attractive setting quasi-fixed points are complete. Since attractivity is implied by either of transitivity or antisymmetry, in particular fixed points are complete in complete trellis, thus completing Stauti and Maaden's result.

In Section 4.3 we further generalize the result, proving that antisymmetry is sufficient for *strict* fixed points $f(x) = x$ to be complete.

4.1 Existence of Quasi-Fixed Points

First, we generalize the existence of fixed points so that nothing besides completeness is assumed on the relation. Fortunately, Quickcheck [3] quickly refutes the existence of *strict* fixed point $f(x) = x$ for an arbitrary complete relation.

▶ **Example 1** (by Quickcheck). Let $A = \{\mathsf{a}_1, \mathsf{a}_2\}$, $(\sqsubseteq) = A \times A$, $f(\mathsf{a}_1) = \mathsf{a}_2$, and $f(\mathsf{a}_2) = \mathsf{a}_1$. Trivially f is monotone but $f(x) \neq x$ for either $x \in A$.

Hence, we instead show the existence of a quasi-fixed point $f(x) \sim x$. For reusability of proofs for the completeness results later on, we start with a stronger statement, namely: there exists a quasi-fixed point in any set of elements that is closed under f and complete for $(\sqsubseteq)$. Completeness restricted to a subset of elements is formalized as follows:

```
definition "complete_in S ≡ ∀X ⊆ S. Ex (extreme_bound_in S X)"
```

where predicate extreme_bound_in indicates the least elements among the bounds restricted to a given subset.

```
abbreviation "extreme_bound_in S X ≡ extreme (⊒) {b ∈ S. bound (⊑) X b}"
```

For convenience we construct a proof within the following context.

```
context
  fixes f and S
  assumes "monotone (⊑) (⊑) f" and "f ‘ S ⊆ S" and "complete_in (⊑) S"
```

Inspired by Stauti and Maaden [20], we start the proof by considering the set of subsets of S that are closed under f and themselves "complete":

```
definition AA where "AA ≡
  {A. A ⊆ S ∧ f ‘ A ⊆ A ∧ (∀B ⊆ A. ∀b. extreme_bound_in (⊑) S B b ⟶ b ∈ A)}"
```

Note here that by a "complete" subset $A \subseteq S$ we mean that *any* suprema with respect to S are in A, since suprema are not necessarily unique. We denote the intersection of all those subsets by C, and show that C contains a quasi-fixed point.

```
definition C where "C ≡ ⋂ AA"

lemma quasi_fixed_point_in_C: "∃c ∈ C. f c ∼ c"
```

Proof. We prove that any supremum c of C in S, which exists due to the completeness of S, is a quasi-fixed point of f. First, observe that $\mathsf{C} \in \mathsf{AA}$. Indeed:

- $\mathsf{C} \subseteq S$: since S is closed under f and complete, $S \in \mathsf{AA}$.
- $f(\mathsf{C}) \subseteq \mathsf{C}$: for every $A \in \mathsf{AA}$, we have $f(\mathsf{C}) \subseteq f(\mathsf{A}) \subseteq \mathsf{A}$. So $f(\mathsf{C}) \subseteq (\bigcap \mathsf{AA}) = \mathsf{C}$.
- completeness: given $B \subseteq \mathsf{C}$ and its supremum b in S, we prove $b \in \mathsf{C}$, that is, $b \in A'$ for every $A' \in \mathsf{AA}$. Indeed, we have $B \subseteq \mathsf{C} \subseteq A'$ and the definition of AA ensures $b \in A'$.

This implies that $c \in \mathsf{C}$. Moreover, since $f(\mathsf{C}) \subseteq \mathsf{C}$, we have $f(c) \in \mathsf{C}$, and since c is a supremum of C, we get $f(c) \sqsubseteq c$. It remains to prove the converse orientation $c \sqsubseteq f(c)$. To this end we consider the following set D:

```
define D where "D ≡ {x ∈ C. x ⊑ f c}"
```

We conclude by proving that $\mathsf{D} \in \mathsf{AA}$, since this implies $\mathsf{C} \subseteq \mathsf{D}$ and in particular $c \in \mathsf{D}$, which means $c \sqsubseteq f(c)$.

- $\mathsf{D} \subseteq S$: because $\mathsf{D} \subseteq \mathsf{C} \subseteq S$.
- $f(\mathsf{D}) \subseteq \mathsf{D}$: Let $d \in \mathsf{D}$. So $d \in \mathsf{C}$, and since c is a supremum of C, we have $d \sqsubseteq c$. With the monotonicity of f we get $f(d) \sqsubseteq f(c)$ and thus $f(d) \in \mathsf{D}$.
- completeness: Given $E \subseteq \mathsf{D}$ and its supremum b in S, we prove that $b \in \mathsf{D}$. Since $E \subseteq \mathsf{D}$, $f(c)$ is a bound of E, and as b is a least of such, $b \sqsubseteq f(c)$, that is $b \in \mathsf{D}$. ◀

By taking $S = \mathsf{UNIV}$ in the above lemma, we obtain:

```
theorem (in complete) monotone_imp_ex_quasi_fixed_point:
  assumes "monotone (⊑) (⊑) f" shows "∃s. f s ∼ s"
```

It is easy to see that this result indicates the existence of a strict fixed point if the relation $\sqsubseteq$ is antisymmetric, recovering statement (1) in the context of Stauti and Maaden [20], but without requiring reflexivity.

```
locale complete_antisymmetric = complete + antisymmetric
```

```
corollary (in complete_antisymmetric) monotone_imp_ex_fixed_point:
  assumes "monotone (⊑) (⊑) f" shows "∃s. f s = s"
```

4.2 Completeness of Quasi-Fixed Points

Next, we tacle the completeness of quasi-fixed points, generalizing statement (2). It was a surprise to us that, this time Nitpick [6] found a counterexample for this claim.

▶ **Example 2** (by Nitpick). We claimed (**in** complete) **assumes** "monotone (⊑) (⊑) f" **shows** "complete_in (⊑) {s. f s ∼ s}" and typed **nitpick**. In seconds it found a counterexample:

```
f = (λx. _) (a₁ := a₃, a₂ := a₃, a₃ := a₃, a₄ := a₁)
    (⊑) =
      (λx. _)
      (a₁ := (λx. _) (a₁ := False, a₂ := True, a₃ := True, a₄ := True),
       a₂ := (λx. _) (a₁ := True, a₂ := True, a₃ := True, a₄ := True),
       a₃ := (λx. _) (a₁ := True, a₂ := False, a₃ := True, a₄ := False),
       a₄ := (λx. _) (a₁ := True, a₂ := True, a₃ := True, a₄ := False))
```

Below we depict the relation $\sqsubseteq$ (left) and the mapping f (right).

On the left, arrow $a_i \to a_j$ means $a_i \sqsubseteq a_j$, and arrow $a_i \leftrightarrow a_j$ means $a_i \sim a_j$. On the right, an arrow $a_i \dashrightarrow a_j$ means $f(a_i) = a_j$. In this example, indeed $\sqsubseteq$ is complete and f is monotone. The quasi-fixed points are a_1, a_3, a_4; however, none of them are least, because $a_1 \not\sqsubseteq a_1$, $a_3 \not\sqsubseteq a_4$ and $a_4 \not\sqsubseteq a_4$.

After analysing the counterexample and existing proofs for lattices and trellises, we found a mild requirement on the relation $\sqsubseteq$, that we call *(semi)attractivity*:

```
locale semiattractive = less_eq_syntax +
  assumes attract: "x ⊑ y ⟹ y ⊑ x ⟹ x ⊑ z ⟹ y ⊑ z"
```

```
locale attractive = semiattractive + dual: semiattractive "(⊒)"
```

The intuition of this assumption is dipicted in Fig. 2. Attractivity is so mild that it is implied by either of antisymmetry and transitivity:

```
sublocale transitive ⊆ attractive by (unfold_locales, auto dest: trans)
sublocale antisymmetric ⊆ attractive by (unfold_locales, auto)
```

Figure 2 Attractivity: If two elements are similar, then arrows coming to one of them is also "attracted" to the other.

Assuming attractivity and completeness, we prove that the set of quasi-fixed points of a relation-preserving map f are complete. We start with a lemma saying that any complete subset S closed under f has a least quasi-fixed point:

```
lemma ex_extreme_quasi_fixed_point:
  assumes "monotone (⊑) (⊑) f" and "f ' S ⊆ S" and "complete_in (⊑) S"
    and attract: "∀q x. f q ∼ q ⟶ x ⊑ f q ⟶ x ⊑ q"
  shows "Ex (extreme (⊒) {q ∈ S. f q ∼ q})"
end
```

Proof. We start by defining the set of lower bounds of the quasi-fixed points in S.

```
define A where "A ≡ {a ∈ S. ∀s ∈ S. f s ∼ s ⟶ a ⊑ s}"
```

Let us first show that A ∈ AA, using the notation from the previous section.

- A ⊆ S: By definition.
- f(A) ⊆ A: Let a ∈ A. For any quasi-fixed point $s \in S$, we have that $a \sqsubseteq s$ and by monotonicity, $f(a) \sqsubseteq f(s)$. Since $f(s) \sim s$, by attract we get $f(a) \sqsubseteq s$, and thus $f(a) \in$ A.
- Completeness: Given B ⊆ A, we show that any supremum b of B in S is in A. Since every quasi-fixed point s in S is a bound of A, s is a bound of B. As b is a least of such, we get $b \sqsubseteq s$ and thus $b \in$ A.

This implies C ⊆ A, and with lemma quasi_fixed_point_in_C we obtain a quasi-fixed point in C ⊆ A ⊆ S. This is a least one by the definition of A. ◀

Finally, we prove that the set of quasi-fixed points of f is complete.

```
locale complete_attractive = complete + attractive
```

```
theorem (in complete_attractive) monotone_imp_quasi_fixed_points_complete:
  assumes "monotone (⊑) (⊑) f" shows "complete_in (⊑) {s. f s ∼ s}"
```

Proof. Given a subset A of quasi-fixed points, we prove that A has a supremum *inside* the set of quasi-fixed points. Define S the set of bounds of A.

```
define S where "S ≡ {s. ∀a ∈ A. a ⊑ s}"
```

We prove that S satisfies the assumptions of ex_extreme_quasi_fixed_point:

- f(S) ⊆ S: Let $s \in$ S. By the definition of S, for any $a \in A$ we have $a \sqsubseteq s$, and with monotonicity $f(a) \sqsubseteq f(s)$. Then by dual.attract with $f(a) \sim a$, we get $a \sqsubseteq f(s)$, and thus $f(s) \in$ S.
- Completeness: Due to the duality of completeness, it suffices to prove that every subset B of S has an infimum in S. As the universe is complete, B has an infimum b in UNIV. By the definition of S, every $a \in A$ is a lower bound of S and so of B. As b is a greatest of such, we get $a \sqsubseteq b$, concluding $b \in$ S.

Consequently, by ex_extreme_quasi_fixed_point, we find a least quasi-fixed point q in S. We conclude the proof by showing that q is a least bound of A, restricted to the set of quasi-fixed points:

- q is a quasi-fixed point: by construction.
- q is a bound of A: by construction, q is in S.
- q is least: Let p be another quasi-fixed point which is also a bound of A. Then p is a quasi-fixed point in S, and by construction of q, $q \sqsubseteq p$. ◀

The second result of Stauti and Maaden [20] states that, for a monotone map in a complete trellis, there exists a least fixed point. We have already obtained a stronger result: the set of fixed points are complete in complete trellises, since quasi-fixed points are precisely fixed points in pseudo-orders. Nevertheless, holding the as-general-as-possible manifesto in mind, we further generalize the result to show that antisymmetry alone is sufficient for the set of fixed points to be complete.

4.3 Completeness of Fixed Points in Antisymmetry

Now we prove that the set of strict fixed points is complete, only assuming antisymmetry. Observe first that this is not an immediate consequence of the completeness of quasi-fixed points, since when reflexivity is not available, there can be more fixed points than quasi-fixed points. So we have to show that there is no fixed points below the least quasi-fixed point we have found.

The proof relies on the following technical lemma, stating that given two sets A and B of strict fixed points, such that every element of A is below every element of B, there is a quasi-fixed point in-between.

```
lemma qfp_interpolant:
  assumes "complete (⊑)" and "monotone (⊑) (⊑) f"
    and "∀a ∈ A. ∀b ∈ B. a ⊑ b"
    and "∀a ∈ A. f a = a"
    and "∀b ∈ B. f b = b"
  shows "∃t. (f t ∼ t) ∧ (∀a ∈ A. a ⊑ t) ∧ (∀b ∈ B. t ⊑ b)"
```

Proof. We first define the set T of elements in between A and B:

```
define T where "T ≡ {t. (∀a ∈ A. a ⊑ t) ∧ (∀b ∈ B. t ⊑ b)}"
```

It is enough to prove that T satisfies the assumptions of lemma quasi_fixed_point_in_C:

- $f(\mathsf{T}) \subseteq \mathsf{T}$: Let $t \in \mathsf{T}$. Then for every $a \in A$, $a \sqsubseteq t$ and by monotonicity $f(a) \sqsubseteq f(t)$. Since a is a fixed point, we have $a = f(a) \sqsubseteq f(t)$. Similarly, we have $f(t) \sqsubseteq b$ for every $b \in B$, and thus $f(t) \in \mathsf{T}$.
- completeness: Let $C \subseteq \mathsf{T}$ and let us prove that C has a supremum in T. By the completeness of $(\sqsubseteq)$, we find a supremum c of $C \cup A$ in UNIV. Let us prove that this is a supremum of C in T:
  - $c \in \mathsf{T}$: By construction, c is a bound of A. Since $C \subseteq \mathsf{T}$, every $b \in B$ is a bound of C, and as c is least of such, $c \sqsubseteq b$. Consequently, $c \in \mathsf{T}$.
  - c is a bound of C: by construction.
  - c is least: Let $d \in \mathsf{T}$ be another bound of C. By the definition of T, d is also a bound of A, and so of $C \cup A$. As c is least of such, we conclude $c \sqsubseteq d$. ◀

From this lemma, we deduce that the set of strict fixed points is complete.

theorem (**in** complete_antisymmetric) monotone_imp_fixed_points_complete:
assumes mono: "monotone (⊑) (⊑) f" **shows** "complete_in (⊑) {s. f s = s}"

Proof. Let A be a subset of strict fixed points. Similarly to the proof of attract_imp_qfp_complete, define the set S of bounds of A. This set S still satisfies the assumptions of ex_extreme_quasi_fixed_point, so it has a least *quasi*-fixed point q. We prove that this is a supremum of A with respect to the set of (strict) fixed points.

- q is a fixed point: by antisymmetry and the fact that q is a quasi-fixed point.
- q is a bound of A: because $q \in$ S.
- q is least: Let p be a fixed point and at the same time a bound of A. Let $B = \{q, p\}$. Then A and B satisfy the assumption of monotone_imp_interpolant_quasi_fixed_point. So there is a quasi-fixed point t between A and B. In particular, $t \sqsubseteq q$ and $t \sqsubseteq p$. Since t is a bound of A, we know $t \in$ S. Since q is a least quasi-fixed point in S, we get $q \sqsubseteq t$. With $t \sqsubseteq q$ and antisymmetry we get $q = t$, and since $t \sqsubseteq p$, we conclude $q \sqsubseteq p$. ◀

5 Kleene-Style Fixed-Point Theorems

Kleene's fixed-point theorem states that, for a pointed directed complete partial order $\langle A, \sqsubseteq\rangle$ and a Scott-continous map $f : A \to A$, the supremum of $\{f^n(\bot) \mid n \in \mathbb{N}\}$ exists in A and is a least fixed point. Mashburn [16] generalized the result so that $\langle A, \sqsubseteq\rangle$ is a ω-complete partial order and f is ω-continuous.

In this section we further generalize the result and show that for ω-complete relation $\langle A, \sqsubseteq\rangle$ and for every bottom element $\bot \in A$, the set $\{f^n(\bot) \mid n \in \mathbb{N}\}$ has suprema (not necessarily unique, of course) and, they are quasi-fixed points. Moreover, if $(\sqsubseteq)$ is attractive, then the suprema are precisely the least quasi-fixed points.

5.1 Scott Continuity, ω-Completeness, ω-Continuity

A related set $\langle A, \sqsubseteq\rangle$ is *ω-complete* if every ω-chain – a countable set in which any two elements are related – has a supremum. In order to characterize ω-chains in Isabelle (without going into ordinals), we model an ω-chain as the range of a relation-preserving map $c : \mathbb{N} \to A$.

locale omega_complete = less_eq_syntax +
assumes "⋀ c :: nat ⇒ 'a. monotone (≤) (⊑) c ⟹ Ex (extreme_bound (⊑) (range c))"

A map $f : A \to A$ is *Scott-continuous* with respect to $(\sqsubseteq) \subseteq A \times A$ if for every directed subset $D \subseteq A$ with a supremum s, $f(s)$ is a supremum of the image $f(D)$.

definition "scott_continuous f ≡
∀D s. directed (⊑) D ⟶ extreme_bound (⊑) D s ⟶ extreme_bound (⊑) (f ‘ D) (f s)"

The notion of *ω-continuity* relaxes Scott-continuity by considering only ω-chain as D.

definition "omega_continuous f ≡ ∀c :: nat ⇒ 'a. ∀s.
monotone (≤) (⊑) c ⟶
extreme_bound (⊑) (range c) s ⟶ extreme_bound (⊑) (f ‘ range c) (f s)"

As $\langle \mathbb{N}, \leq\rangle$ is total, and thus directed, we can easily verify that Scott-continuity implies ω-continuity using the fact that the image of a monotone map over a directed set is directed.

lemma scott_continous_imp_omega_continous:
assumes "scott_continuous f" **shows** "omega_continuous f"

For the later development we also prove that every ω-continuous function is *nearly* monotone, in the sense that it preserves relation $x \sqsubseteq y$ when x and y are reflexive elements. Note that near monotonicity coincides with monotonicity if the underlying relation is reflexive.

```
lemma omega_continous_imp_mono_refl:
  assumes "omega_continuous f" and "x ⊑ y" and "x ⊑ x" and "y ⊑ y"
  shows "f x ⊑ f y"
```

Proof. The proof consists in observing that under the assumptions, function c :: nat ⇒ 'a defined by "c i ≡ if i = 0 then x else y" is monotone. Furthermore, y is a supremum of the image of c, i.e., $\{x, y\}$, so ω-continuity ensures that $f(y)$ is a supremum of $\{f(x), f(y)\}$, which in particular means that $f(x) \sqsubseteq f(y)$. ◀

5.2 Kleene's Fixed-Point Theorem

The first part of Kleene's theorem demands to prove that the set $\{f^n(\bot) \mid n \in \mathbb{N}\}$ has a supremum and that all such are quasi-fixed points. We prove this claim without assuming anything on the relation $\sqsubseteq$ besides ω-completeness and one bottom element.

```
context
  fixes f and bot ("⊥")
  assumes "omega_complete (⊑)" and "omega_continuous (⊑) f" and "∀x. ⊥ ⊑ x"
begin
```

Just for convenience we abbreviate the set $\{f^n(\bot) \mid n \in \mathbb{N}\}$ as Fn in Isabelle:

```
abbreviation(input) fn where "fn n ≡ (f ^^ n) ⊥"
abbreviation(input) "Fn ≡ range fn"

theorem kleene_quasi_fixed_point:
  shows "∃p. extreme_bound (⊑) Fn p" and "extreme_bound (⊑) Fn p ⟹ f p ∼ p"
```

Proof. First note that fn is a relation-preserving map from $\langle \mathbb{N}, \leq \rangle$ to $\langle A, \sqsubseteq \rangle$: this is reduced to $f^n(\bot) \sqsubseteq f^{n+k}(\bot)$ for any n and k, which is easily proved by induction on n. Thus Fn = range fn is an ω-chain, and ω-completness gives a supremum, say p, for Fn. Now let us prove that p is a quasi-fixed point.

Since p is a supremum of Fn, the ω-continuity of f ensures that $f(p)$ is a supremum of $f(\mathsf{Fn})$. As p is a bound of Fn, it is also a bound of $f(\mathsf{Fn})$ due to the definition of Fn. Consequently, $f(p) \sqsubseteq p$.

It remains to show the other orientation $p \sqsubseteq f(p)$. Since p is least in the bounds of Fn, it suffices to show that $f(p)$ is a bound of Fn, that is, $f^n(\bot) \sqsubseteq f(p)$ for every n. We prove this by induction on n. The base case is by the assumption of $\bot$. For inductive case, assume $f^n(\bot) \sqsubseteq p$. By the "near" monotonicity we conclude $f^{n+1}(\bot) \sqsubseteq f(p)$, but to this end we need $f^n(\bot) \sqsubseteq f^n(\bot)$ for every n, which would be trivial if we had reflexivity. Instead we prove this fact by induction on n, also using omega_continous_imp_mono_refl. ◀

Now the first part of Kleene's theorem is reproved without any order assumption: for an ω-complete set $\langle A, \sqsubseteq \rangle$ with a bottom element $\bot$ and ω-continuous map $f : A \to A$, there exists a supremum for $\{f^n(\bot) \mid n \in \mathbb{N}\}$ and it is a quasi-fixed point.

Kleene's theorem also states that the quasi-fixed point found this way is a least one. Hence naturally we consider proving this claim for arbitrary relations, but again Nitpick saved us this hopeless effort.

▶ **Example 3** (by Nitpick). Our conjecture is now "extreme_bound (⊑) Fn q ⟹ extreme (⊒) {s. f s ∼ s} q". Following is a counterexample found by Nitpick:

```
⊥ = a1
  f = (λx. _) (a1 := a3, a2 := a1, a3 := a3)
  (⊑) =
    (λx. _)
    (a1 := (λx. _) (a1 := True, a2 := True, a3 := True),
     a2 := (λx. _) (a1 := True, a2 := False, a3 := True),
     a3 := (λx. _) (a1 := True, a2 := False, a3 := True))
  q = a3
```

In this example, indeed a_1 is a bottom element, $\sqsubseteq$ is (ω-)complete, and f is ω-continuous. The set of quasi-fixed points is $\{a_1, a_2, a_3\}$, and a_3 is an extreme bound of $\{f^n(\bot) \mid n \in \mathbb{N}\} = \{a_1, a_3\}$. However, a_3 is not a least quasi-fixed point because $a_3 \not\sqsubseteq a_2$.

Now again, attractivity turns out to be the key. We prove that the set of suprema of Fn coincides with the set of least quasi-fixed points, if the underlying relation is attractive.

```
corollary (in attractive) kleene_fixed_point_dual_extreme:
  shows "extreme_bound (⊑) Fn = extreme (⊒) {s. f s ∼ s}"
```

Proof. Let q be a supremum of Fn. By kleene_quasi_fixed_point, we already know that this is a quasi-fixed point. So to prove that q is a least quasi-fixed point, it is enough to show that any other quasi-fixed point s is a bound of $\mathsf{Fn} = \{f^n(\bot) \mid n \in \mathbb{N}\}$. This is done by induction on n. The base case $\bot \sqsubseteq s$ is trivial by assumption. For the inductive case, assuming $f^n(\bot) \sqsubseteq s$ we get $f^{n+1}(\bot) \sqsubseteq f(s)$ by the same argument as in the previous proof. Since $f(s) \sim s$, attractivity concludes $f^{n+1}(\bot) \sqsubseteq s$.

Conversely, consider a least quasi-fixed point s. We show that s is a supremum of Fn. Since s is a quasi-fixed point, and as we have just proved above, s is a bound of Fn. It remains to prove that s is least in bounds of Fn.

By kleene_quasi_fixed_point, Fn has a supremum, say k, and is a quasi-fixed point. As s is a least quasi-fixed point, we have $s \sqsubseteq k$. On the other hand, as s is a bound of Fn and k is a least of such, we see $k \sqsubseteq s$. Consequently, $s \sim k$.

Now let x be a bound of Fn. We know $k \sqsubseteq x$, and with $s \sim k$, we conclude $s \sqsubseteq x$ due to attractivity. ◀

6 Conclusion

In this paper, we developed an Isabelle/HOL formalization for order-theoretic concepts such as various completeness conditions and fixed-point theorems. We adopt an as-general-as-possible approach, so that many results previously known only for partial orders or pseudo-orders are generalized. In particular the generalizations of the Knaster–Tarski theorem and Kleene's fixed-point theorems would deserve some attention. These achievement become reachable to us largely due to the great assistance by the smart Isabelle 2018 environment.

For future work, it is tempting to further formalize and hopefully generalize other results about completeness and fixed points, which are listed as related work in the introduction. We also plan to extend the library with convergence arguments, which were actually our original motivation for formalizing these order-theoretic concepts.

References

1. Samson Abramsky and Achim Jung. *Domain Theory*. Number III in Handbook of Logic in Computer Science. Oxford University Press, 1994.
2. Clemens Ballarin. Interpretation of Locales in Isabelle: Theories and Proof Contexts. In Jonathan M. Borwein and William M. Farmer, editors, *Proceedings of the 5th International Conference on Mathematical Knowledge Management (MKM 2006)*, volume 4108 of *LNCS*, pages 31–43. Springer Berlin Heidelberg, 2006. doi:10.1007/11812289_4.
3. Stefan Berghofer and Tobias Nipkow. Random Testing in Isabelle/HOL. In *Proceedings of the 2nd International Conference on Software Engineering and Formal Methods (SEFM 2004)*, pages 230–239. IEEE Computer Society, 2004. doi:10.1109/SEFM.2004.36.
4. George M. Bergman. *An Invitation to General Algebra and Universal Constructions*. Springer International Publishing, Cham, 2015. doi:10.1007/978-3-319-11478-1.
5. S. Parameshwara Bhatta. Weak chain-completeness and fixed point property for pseudo-ordered sets. *Czechoslovak Mathematical Journal*, 55(2):365–369, 2005.
6. Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder. In Matt Kaufmann and Lawrence C. Paulson, editors, *Proceedings of the 1st International Conference on Interactive Theorem Proving (ITP 2010)*, volume 6172 of *LNCS*, pages 131–146. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-14052-5_11.
7. Sascha Böhme and Tobias Nipkow. Sledgehammer: Judgement Day. In *Proceedings of the 5th International Joint Conference on Automated Reasoning (IJCAR 2010)*, volume 6173 of *LNCS*, pages 107–121. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-14203-1_9.
8. B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002. doi:10.1017/CBO9780511809088.
9. G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. W. Mislove, and D. S. Scott. *Continuous Lattices and Domains*. Cambridge University Press, 2003. doi:10.1017/CBO9780511542725.
10. Georges Gonthier. Formal proof – the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.
11. Florian Haftmann and Tobias Nipkow. A code generator framework for Isabelle/HOL. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics: Emerging Trends*, pages 128–143. Department of Computer Science, University of Kaiserslautern, 2007.
12. Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Hoang Le Truong, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, et al. A formal proof of the Kepler conjecture. *Forum of Mathematics, Pi*, 5:e2, 2017. doi:10.1017/fmp.2017.1.
13. Florian Kammüller. Modular Reasoning in Isabelle. In David McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction (CADE-17)*, volume 1831 of *LNCS*, pages 99–114. Springer Berlin Heidelberg, 2000. doi:10.1007/10721959_7.
14. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Wiwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP 2009)*, pages 207–220. ACM, 2009. doi:10.1145/1629575.1629596.
15. K. Leutola and J. Nieminen. Posets and generalized lattices. *Algebra Universalis*, 16(1):344–354, 1983.
16. J. D. Mashburn. The least fixed point property for omega-chain continuous functions. *Houston Journal of Mathematics*, 9(2):231–244, 1983.

17 T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. doi:10.1007/3-540-45949-9.

18 S. Parameshwara Bhatta and Shiju George. Some fixed point theorems for pseudo ordered sets. *Algebra and Discrete Mathematics*, 11(1):17–22, 2011.

19 H.L. Skala. Trellis theory. *Algebra Univ.*, 1:218–233, 1971. doi:10.1007/BF02944982.

20 Abdelkader Stouti and Abdelhakim Maaden. Fixed points and common fixed points theorems in pseudo-ordered sets. *Proyecciones*, 32(4):409–418, 2013. doi:10.4067/S0716-09172013000400008.

21 Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

22 Makarius Wenzel. Isabelle/jEdit – a prover IDE within the PIDE framework. In *Proceedings of the 5th Conferences on Intelligent Computer Mathematics (CICM 2012)*, volume 7362 of *LNCS*, pages 468–471. Springer Berlin Heidelberg, 2012. doi:10.1007/978-3-642-31374-5_38.

Verified Decision Procedures for Modal Logics

Minchao Wu
Research School of Computer Science, Australian National University, Australia

Rajeev Goré
Research School of Computer Science, Australian National University, Australia

Abstract

We describe a formalization of modal tableaux with histories for the modal logics K, KT and S4 in Lean. We describe how we formalized the static and transitional rules, the non-trivial termination and the correctness of loop-checks. The formalized tableaux are essentially executable decision procedures with soundness and completeness proved. Termination is also proved in order to define them as functions in Lean. All of these decision procedures return a concrete Kripke model in cases where the input set of formulas is satisfiable, and a proof constructed via the tableau rules witnessing unsatisfiability otherwise. We also describe an extensible formalization of backjumping and its verified implementation for the modal logic K. As far as we know, these are the first verified decision procedures for these modal logics.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Type theory; Software and its engineering → Formal methods

Keywords and phrases Formal Methods, Interactive Theorem Proving, Modal Logic, Lean

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.31

Supplement Material The formalization is available at `https://github.com/minchaowu/ModalTab`

1 Introduction

Propositional modal logics have proved useful for reasoning about knowledge and belief [24], verifying digital circuits [8], and knowledge representation and reasoning [1].

The main reason for their success is that they provide just the right amount of extra expressive power, somewhere between propositional and first order logic, while retaining almost universal decidability. Modal description logics [3] in particular are extremely expressive, many with decision procedures that are EXPTIME-complete and beyond.

There are many efficient implementations of various modal and description logics, but the desire for efficiency leads to numerous non-trivial optimizations which make the theoretical soundness and completeness harder to prove. Consequently, most of these implementations are buggy and require constant maintenance to iron out these bugs. For efficiency, these provers also do not provide concrete evidence, such as proofs or countermodels, for their answers. As such, these implementations cannot be used in safety-critical applications.

Naive decision procedures for modal logics sometimes proceed by constructing the, possibly exponential sized, set of all maximal consistent subsets of a given set Γ and then attempting to construct a model directly by comparing when two such subsets can be related by the semantic binary relation. They are analogous to the construction of a canonical model when using a Henkin-style completeness proof for a Hilbert calculus for modal logic. If Γ is finite then so is its set of maximal consistent sets, so termination is usually easy [19], and it suffices to prove soundness and completeness constructively [11]. But they are not practical because their first task requires a possibly unnecessary exponential operation. More refined "on the fly" tableau procedures [29] only construct the set of subsets in the worst-case, and as is well-known, real-world examples rarely contain such worst-case examples.

Our work follows this "refined" approach. We break new ground for producing verified and optimized implementations for modal description logics by handling the basic modal logics K, KT and S4. For K, we also give a verified implementation of backjumping [3]. The logic K allows us to set the scene and incorporate backjumping. The logic KT allows us to showcase how to handle axiomatic extensions. The termination argument for the logic S4 requires detecting "ancestor loops" in tableau branches. Loop-checking is also required to handle knowledge bases (global assumptions), which encode real-world problems [4].

By utilizing the constructive nature and strong type system of Lean [9], we implemented verified decision procedures based on tableaux with histories, which are variants of sequent calculi with histories given by Heuerding, Seyfried, and Zimmermann [16] (HSZ henceforth). However, our formalization does not mimic the proofs given in HSZ. Although the verified decision procedures are not competitive against state-of-the-art provers, they provide promising evidence that efficient verified provers for expressive modal description logics are plausible.

The verified decision procedures are functions defined in Lean. They could be executed using `#reduce` but this will take too long to compute, meaning they cannot be used directly in a Lean proof. Instead, we use `#eval` to execute these functions on the virtual machine provided by Lean, and thus obtain our experimental results.

Related Work

Formalizations of theories, decision procedures and SAT solvers for classical propositional logic and first order logic have been well studied. Many recent verified provers also come with verified optimizations [7] [26]. These formalizations usually adopt modern variants of the resolution method as their primary calculus in order to achieve efficiency. On the other hand, there are also verified decision procedures based on tableau methods [21] [2] [17]. However, the target logics of these projects are either classical propositional logic [21] or basic description logics [2], without loop-checks, and without verified optimizations. For example, Hidalgo et al. [17] verified a decision procedure for description logic ALC, but their satisfiability was then defined with respect to empty global assumptions, so loop-checks are not required. Our formalization also extends their work. Other work related to modal and temporal logics includes Paulien [10], Bentzen [6] and Yuasa *et al.*[30]. Paulien [10] gives a comprehensive account of embedding modal logics in Coq. Bentzen [6] gives a formalization of Henkin-style completeness proof of modal logics in Lean. Yuasa et al. [30] use an external decision procedure for the μ-calculus to help verify the Deutsch-Schorr-Waite marking algorithm in Agda, but leave the decision procedure itself trusted. There are also formalizations of temporal logics developed by Schimpf et al. [25], Jantsch and Norrish [18], Esparza et al. [12], targeting verification related to model checking problems including verified model checkers, and translation between temporal logics and automata.

2 Modal logic preliminaries

Our verified decision procedures are all implemented with lists. However, for readability, we use usual mathematical notation for sets in the following when there is no confusion.

2.1 Syntax and semantics of K, KT and S4

▶ **Definition 2.1.** *The syntax of formulas in this paper is given by the following grammar.*

$$\mathbb{N} ::= 0 \mid S\ \mathbb{N}$$
$$\varphi ::= \mathbb{N} \mid \neg\mathbb{N} \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \Box\varphi \mid \Diamond\varphi$$

▶ **Definition 2.2.** *The length l of a formula φ is the number of logical connectives including $\Box$ and $\Diamond$ occurring in φ. The length l of a set Γ of formulas is $\sum_{\varphi\in\Gamma} l(\varphi)$. The closure cl of a formula φ is the set of all the subformulas of φ.*

To obtain a neat formalization we only consider formulas in negation normal form as defined in Definition 2.1. However, it is easy to establish a translation between the full language and the negation normal form, preserving the correctness of the decision procedures.

▶ **Definition 2.3** (Kripke models). *A Kripke model is a triple (S, R, V) where S is a set of states, and $R \subseteq S \times S$ and $V \subseteq \mathbb{N} \times S$ are two binary relations. R is called a reachability relation, and V is called a valuation function.*

▶ **Definition 2.4.** *A KT model is a Kripke model whose reachability relation is reflexive. An S4 model is a KT model whose reachability relation is transitive.*

▶ **Definition 2.5** (forcing). *For a Kripke model $M = (S, R, V)$, the forcing relation $\Vdash$ between a state $s \in S$ and a formula φ is:*

$$
\begin{array}{ll}
(M,s) \Vdash n & \text{if } V(n,s) \\
(M,s) \Vdash \neg n & \text{if } (M,s) \nVdash n \\
(M,s) \Vdash \varphi \wedge \psi & \text{if } (M,s) \Vdash \varphi \text{ and } (M,s) \Vdash \psi \\
(M,s) \Vdash \varphi \vee \psi & \text{if } (M,s) \Vdash \varphi \text{ or } (M,s) \Vdash \psi \\
(M,s) \Vdash \Box\varphi & \text{if for all } t \in S, R(s,t) \text{ implies } (M,t) \Vdash \varphi \\
(M,s) \Vdash \Diamond\varphi & \text{if there exists } t \in S, R(s,t) \text{ and } (M,t) \Vdash \varphi
\end{array}
$$

▶ **Definition 2.6** (satisfiability). *Let M be a Kripke model. A state $s \in M$ satisfies a set Γ of formulas, written $(M, s) \vDash \Gamma$, if for all $\varphi \in \Gamma$, $(M, s) \Vdash \varphi$. A set Γ of formulas is satisfiable if there is a Kripke model containing a state that satisfies Γ, and is unsatisfiable otherwise.*

We write $s \Vdash \varphi$ and $s \vDash \Gamma$ if the model M is clear from the context. Kripke models are formalized as a Lean structure equipped with two relations, parameterized by a carrier type `states`. Also note that by definition, an empty model never satisfies a set Γ of formulas. When Γ is proved to be unsatisfiable, then it is also not satisfied in any non-empty model.

```
structure kripke (states : Type) :=
(val : ℕ → states → Prop)
(rel : states → states → Prop)

def sat {st} (k : kripke st) (s) (Γ : list nnf) : Prop :=
∀ φ ∈ Γ, force k s φ

def unsatisfiable (Γ : list nnf) : Prop :=
∀ (st) (k : kripke st) s, ¬ sat k s Γ
```

2.2 Tableaux for K, KT and S4

The tableau $\mathrm{K}^{\mathcal{T}}$ for modal logic K is the calculus defined as in Figure 1. We call the upper part of a rule the upper sequent (lower sequent resp.). Rule (K) is called the transition rule [13]. The computational behaviour of the transition rule has a backtracking flavor. If the lower sequent is unsatisfiable, then so is the upper sequent. If the lower sequent is satisfiable,

$$(id)\ \frac{n, \neg n, \Gamma}{\text{unsatisfiable}} \qquad (\wedge)\ \frac{\varphi \wedge \psi, \Gamma}{\varphi, \psi, \Gamma} \qquad (\vee)\ \frac{\varphi \vee \psi, \Gamma}{\varphi, \Gamma \quad \psi, \Gamma} \qquad (K)\ \frac{\Diamond\varphi, \Box\Sigma, \Gamma}{\varphi, \Sigma}$$

Figure 1 Tableau $\mathrm{K}^{\mathcal{T}}$.

$$(K)\ \frac{\Diamond\Delta, \Box\Sigma, \Gamma}{\varphi_0, \Sigma \quad \varphi_1, \Sigma \quad \ldots \quad \varphi_n, \Sigma}$$

where $\Delta = \{\varphi_0, \ldots, \varphi_n\} \neq \varnothing$ and Γ is a set of literals not containing a pair $n, \neg n$

Figure 2 Equivalent form of the transition rule.

then the decision procedure backtracks and tries another $\Diamond$-formula. If all of them are satisfiable, then so is the upper sequent. In our formalization, the transition rule should be understood as its variant as shown in Figure 2, which encodes such a computational behavior in the form of a rule. This applies to all the transition rules of the tableaux given in this paper. Rules with the semantics captured by our dotted line are also known as AND-nodes, indicating that such rules have a semantic interpretation that is "dual" to that of the $\vee$ rule, and the resulting calculi are also called AND-OR tableaux [14]. We abuse notation by using the same rule names but distinguish them by dotted lines.

▶ **Theorem 2.7** (invertibility). *For the (K) rule (above) and the $(\wedge)$ rule, all the lower sequents are satisfiable if and only if the upper sequent is satisfiable. One of the lower sequents of the $(\vee)$ rule is satisfiable if and only if the upper sequent is satisfiable.*

▶ **Theorem 2.8** (termination). *Let Γ_u be the upper sequent and Γ_l a lower sequent of a non-id rule in $K^{\mathcal{T}}$. Then $l(\Gamma_l) < l(\Gamma_u)$.*

The tableau $\mathrm{KT}^{\mathcal{T}}$ for modal logic KT is obtained by adding the (T) rule to $\mathrm{K}^{\mathcal{T}}$. The tableau $\mathrm{S4}^{\mathcal{T}}$ for modal logic S4 is $\mathrm{KT}^{\mathcal{T}}$ with the transition rule replaced by the rule $(S4)$:

$$(T)\ \frac{\Box\varphi, \Gamma}{\varphi, \Box\varphi, \Gamma} \qquad (S4)\ \frac{\Diamond\varphi, \Box\Sigma, \Gamma}{\varphi, \Box\Sigma}$$

3 Formalization

We now describe verified decision procedures for K, KT and S4. The one for K introduces the basic tools we developed for formalizing modal tableaux, and serves as an overview of the verified algorithms. The one for KT introduces tableaux with histories to handle non-trivial termination, and focuses on their correctness. The one for S4 combines the techniques for K and KT to deal with the correctness of loop-checks.

Each decision procedure is implemented as a computable function in Lean, and is proved to be sound, complete and terminating. They can be evaluated by Lean's `#eval` command.

3.1 Formalizing modal tableaux – K

The invertibility Theorem 2.7 guarantees that each rule of $\mathrm{K}^{\mathcal{T}}$ properly propagates the status of a sequent. Thus a natural way to write a decision procedure f for K is to call f recursively on the lower sequents of each rule and propagate the status upwards [29]. Eventually, the root sequent, which is the goal, will have its status updated. In a theorem prover supporting a strong type theory, the status can be a complex witness such as a proof or a model.

In cases where a formula φ is satisfiable, instead of returning a proof of the statement that φ is satisfiable, which is an existential sentence, we can return a Kripke model as a concrete object that satisfies φ. For the purpose of formalization, such a model should be easy to construct and easy to check. Defining a Kripke structure by specifying all its fields from scratch can be tedious, especially when one wants to extract information from a sequence of Kripke models and re-arrange them by manipulating their R and V to construct a new model. This happens when dealing with the transition rule. To achieve a better solution, we describe a uniform way of constructing Kripke models using tree models with interpretation functions, and let the decision procedure return such a model as a witness when a lower sequent is satisfiable.

A tree model is defined as an inductive type `model` with a first argument of type `list nat`, intuitively representing the propositional variables true in a state, and a recursive argument `list model`, intuitively representing the states reachable from that state. The base case is when the second list is empty and is not encoded explicitly. Interpretation functions `mval` and `mrel` are defined as follows to capture this intuition.

```
inductive model
| cons : list ℕ → list model → model

def mval : ℕ → model → bool
| p (cons v r) := p ∈ v

def mrel : model → model → bool
| (cons v r) m := m ∈ r
```

Note that although such a type is called model, as can be seen from the type of interpretation functions, `model` is supposed to be used as the type of a state. However, such a state contains essentially all the information about a Kripke model constructed so far. It is always possible to recover the model from within a state via the interpretation functions. We define such a recovery `builder`, whose type is exactly just `Kripke model`.

```
def builder : kripke model :=
{val := λ n s, mval n s, rel := λ s₁ s₂, mrel s₁ s₂} -- λ for coercion
```

This mechanism allows us to construct without too much effort a provably correct model of the upper sequent Γ of a transition rule from the models returned by its lower sequents. For example, if l is the list of tree models returned by the recursive calls on the lower sequents of the transition rule, then the tree model s of the upper sequent is simply $s := cons\ v\ l$ where v is a list of natural numbers definable from the upper sequent itself. For non-transition rules, the tree model remains the same. Then we prove `sat builder s` Γ. The return type of the decision procedure f is as follows, where the `//` notation denotes subtypes:

```
inductive node (Γ : list nnf) : Type
| closed : unsatisfiable Γ → node
| open_  : {s // sat builder s Γ} → node
```

Since the return value of calling f on the lower sequents of the transition rule is potentially a list of tree models satisfying each lower sequent, a predicate called `batch_sat` is defined to relate the lower sequents and their models. The function `unmodal` takes a sequent Γ and produces its lower sequent according to the transition rule.

```
inductive batch_sat : list model → list (list nnf) → Prop
| bs_nil : batch_sat [] []
| bs_cons (m Γ l₁ l₂) : sat builder m Γ → batch_sat l₁ l₂ →
                        batch_sat (m::l₁) (Γ::l₂)

-- unbox and undia take a list of formulas, and
-- get rid of the outermost box or diamond of each formula respectively
def unmodal (Γ : list nnf) : list (list nnf) :=
list.map (λ d, d :: (unbox Γ)) (undia Γ)
```

The verification of the transition rule is illustrated next. The type `modal_applicable` expresses the extra conditions of the transition rule: Γ contains only literals, contains no contradictions, and contains at least one $\Diamond$-formula. Moreover, `unmodal_sat_of_sat` not only encodes that if the upper sequent is satisfiable then so is every lower sequent, but also that it holds for any list Δ of formulas that contains all the $\Box$- and $\Diamond$-formulas in Γ.

```
theorem sat_of_batch_sat : Π l Γ (h : modal_applicable Γ),
batch_sat l (unmodal Γ) → sat builder (cons h.v l) Γ

theorem unmodal_sat_of_sat (Γ : list nnf) : ∀ (i : list nnf),
i ∈ unmodal Γ → (∀ {st : Type} (k : kripke st) s Δ
(h₁ : ∀ φ, box φ ∈ Γ → box φ ∈ Δ)
(h₂ : ∀ φ, dia φ ∈ Γ → dia φ ∈ Δ),  sat k s Δ → ∃ s', sat k s' i)
```

The termination of the algorithm is not difficult to formalize, because each lower sequent of each rule contains fewer logical connectives. A termination argument is then given by the length of a sequent. However, the transition rule requires some implementational attention. It is worth noting that a map-like function is needed to execute f on the list of lower sequents. Given a term such as `list.map f l` occurring within the definition of f, Lean does not know automatically that the computation terminates because f is not applied to any arguments. Secondly, the transition rule needs early termination in order to make the algorithm efficient. As soon as one of the lower sequents turns out to be unsatisfiable, the computation should terminate because the upper sequent must be unsatisfiable. We define a dedicated function as follows to achieve early termination and help Lean prove termination.

```
-- psum is the sum type extended to Sort in Lean.
def tmap {p : list nnf → Prop} (f : Π Γ, p Γ → node Γ):
Π Γ : list (list nnf), (∀ i∈Γ, p i) →
psum {i // i∈Γ ∧ unsatisfiable i} {x // batch_sat x Γ}
```

The dependent function f in the argument is an abstraction of the decision procedure f itself with a proof h saying that the input of it satisfies the property p. This p is supposed to be the termination of the transition rule whose proof is given as follows. Since the termination proof is found in the local context where the decision procedure f is being called, Lean knows that this recursive call terminates.

```
def unmodal_size (Γ : list nnf) : ∀ (i : list nnf),
i ∈ unmodal Γ → (node_size i < node_size Γ)
```

Since the return type contains either a proof that the goal is unsatisfiable, or a Kripke model which provably satisfies the goal, soundness and completeness are immediately given. They can also be proved explicitly as follows.

```
def tableau : Π Γ : list nnf, node Γ := ...
using_well_founded {rel_tac := λ _ _, `[exact ⟨_, measure_wf node_size⟩]}

def is_sat (Γ : list nnf) : bool :=
match tableau Γ with
| closed _ := ff
| open_ _  := tt
end

theorem correctness (Γ : list nnf) :
is_sat Γ = tt ↔ ∃ (st : Type) (k : kripke st) s, sat k s Γ
```

3.2 Tableaux with histories – KT

As we can see from the (T) rule of $\mathrm{KT}^{\mathcal{T}}$, the termination of proof search in KT becomes non-trivial. In HSZ, a sequent calculus with histories was proposed to handle termination. Soundness and completeness of such a sequent calculus was then proved by establishing a translation between the original calculus and the one with histories. We use a tableau system with histories based on the sequent calculus with histories, and give a direct semantic proof of soundness and completeness with the corresponding formalization. We use a different termination argument, as we found that the measure given by HSZ does not always decrease.

▶ **Definition 3.1.** *Tableau $KT^{\mathcal{TH}}$ is defined as in Figure 3 where the vertical bar | separates the history Σ, which is a formula-set, from the formula-set Γ carried by each sequent:*

$$(id)\ \frac{\Sigma \mid n, \neg n, \Gamma}{\text{unsatisfiable}} \qquad (\wedge)\ \frac{\Sigma \mid \varphi \wedge \psi, \Gamma}{\Sigma \mid \varphi, \psi, \Gamma} \qquad (\vee)\ \frac{\Sigma \mid \varphi \vee \psi, \Gamma}{\Sigma \mid \varphi, \Gamma \quad \Sigma \mid \psi, \Gamma}$$

$$(T)\ \frac{\Sigma \mid \Box\varphi, \Gamma}{\Box\varphi, \Sigma \mid \varphi, \Gamma} \qquad (K)\ \frac{\Box\Sigma \mid \Diamond\varphi, \Gamma}{\varnothing \mid \varphi, \Sigma}$$

where Γ in (K) is a set of literals not containing a contradiction

Figure 3 Tableau $\mathrm{KT}^{\mathcal{TH}}$.

▶ **Definition 3.2.** *A sequent $\Sigma \mid \Gamma$ is satisfiable if $\Sigma \cup \Gamma$ is satisfiable.*

The procedure to decide whether Γ is satisfiable in KT is similar to the one designed for K. Starting with the goal $\varnothing \mid \Gamma$ as a root sequent, apply rules repeatedly until a contradiction is found or no rule is applicable, whence a KT proof or model can be constructed.

However, the correctness proof now becomes different. The first thing to notice is that $\mathrm{KT}^{\mathcal{TH}}$ does not have the strict subformula property as does $\mathrm{K}^{\mathcal{T}}$. The lower sequent of the (T) rule contains more logical connectives than the upper sequent. Consequently, the termination argument that worked for K does not work for KT. Secondly, although the transition rule in $\mathrm{KT}^{\mathcal{TH}}$ is very similar to that of $\mathrm{K}^{\mathcal{T}}$, the change of semantics from K to KT means its invertibility is not immediately obvious. We prove termination by defining a measure on sequents and showing that such a measure decreases under a well-founded relation every time we apply a rule.

▶ **Definition 3.3.** *Let* Γ *be a set of formulas. The degree of* Γ *is the maximal number of modal operators occurring in any formula* $\varphi \in \Gamma$.

▶ **Definition 3.4.** *Let* $\Sigma \mid \Gamma$ *be a sequent in* $KT^{\mathcal{TH}}$. *The size of* $\Sigma \mid \Gamma$ *is defined as a pair*

$$size(\Sigma \mid \Gamma) := (degree(\Sigma \cup \Gamma), l(\Gamma))$$

▶ **Theorem 3.5.** *Let* $\Sigma \mid \Gamma$ *be the upper sequent and* $\Sigma' \mid \Gamma'$ *a lower sequent of a rule in* $KT^{\mathcal{TH}}$. *Let* $<_{lex}$ *be the lexicographic order on* $\mathbb{N} \times \mathbb{N}$. *Then*

$$size(\Sigma' \mid \Gamma') <_{lex} size(\Sigma \mid \Gamma)$$

Proof. For the (T) rule, $degree(\Sigma \mid \Gamma)$ remains unchanged and $l(\Gamma)$ decreases. For the transition rule, $degree(\Sigma \mid \Gamma)$ decreases. For propositional rules, there are two possibilities: either $degree(\Sigma \mid \Gamma)$ decreases or $degree(\Sigma \mid \Gamma)$ remains unchanged and $l(\Gamma)$ decreases. In either case $size(\Sigma \mid \Gamma)$ decreases. ◀

The invertibility of the transition rule is key to the correctness of the decision procedure. We prove this by establishing a semantic relationship between Σ and Γ of a sequent, using the tree model and interpretation functions mechanism developed in the previous section.

▶ **Definition 3.6.** *A sequent* $\Sigma \mid \Gamma$ *is called reflexive if for every* $\Box\varphi \in \Sigma$, *if a tree model* $m := cons\ v\ l$ *satisfies the following two conditions:*

1. $m \vDash \Gamma$, *and*
2. *for every* $s \in l$, *for every* $\Box\psi \in \Sigma$, $s \Vdash \psi$.

then $m \Vdash \varphi$.

▶ **Theorem 3.7.** *Let* $\Sigma \mid \Gamma$ *be a sequent generated by* $KT^{\mathcal{TH}}$. *Then*

1. Σ *contains only* $\Box$*-formulas.*
2. $\Sigma \mid \Gamma$ *is reflexive.*

A paper proof of Theorem 3.7 could proceed by induction on the construction of sequents. In our formalization, we encode Theorem 3.7 as a property into the definition of a sequent so a sequent cannot be constructed without proving it obeys Theorem 3.7. This avoids the extra work of defining an inductive type representing $\mathrm{KT}^{\mathcal{TH}}$, carrying out an explicit induction and relating such a type to the decision procedure. The final definition of a sequent of $\mathrm{KT}^{\mathcal{TH}}$ is:

```
structure seqt : Type :=
(main : list nnf)
(hdld : list nnf)
-- srefl main hdld says that sequent hdld | main satisfies theorem 3.7(2)
(pmain : srefl main hdld)
-- box_only says there are only boxed formulas in hdld
(phdld : box_only hdld)
```

As an example, we show the construction of a sequent in the implementation. The `and_child` function takes a sequent Γ, assumes that an $\wedge$-formula is found in the main part, and returns a new sequent which is the lower sequent of the $\wedge$-rule with Theorem 3.7 proved. Henceforth, we refer to this way of proving properties of sequents as "downward propagation". The function Γ`.main` returns the `main` field of the sequent Γ.

```
def and_child {φ ψ} (Γ : seqt) (h : nnf.and φ ψ ∈ Γ.main) : seqt :=
⟨φ :: ψ :: Γ.main.erase (and φ ψ), Γ.hdld,
 begin
   intros k s γ hsat hin hall,
   by_cases heq : γ = and φ ψ,
   { rw heq, split, apply hsat, simp, apply hsat, simp },
   { apply Γ.pmain _ hin hall,
     apply sat_and_of_sat_split _ _ _ _ _ h hsat }
 end, Γ.phdld⟩
```

▶ **Theorem 3.8** (invertibility). *All the lower sequents of the transition rule are satisfiable if and only if the upper sequent is satisfiable.*

Proof. ($\Rightarrow$) By Theorem 3.7. ($\Leftarrow$) By KT semantics. ◀

Note that applying Theorem 3.7 by itself gives us satisfiability with respect to tree models. However, in the end of the formalization, an immediate corollary is that a sequent is satisfiable if and only if it is satisfied by a tree model. Thus Theorem 3.8 does not claim too much.

3.3 Loop checks – S4

The transitivity constraint in the semantics of S4 poses more difficulties on both the termination and correctness of the decision procedure. HSZ introduced a sequent calculus with histories for S4 and proved its soundness and completeness via a translation connecting two intermediate calculi. We give a tableau calculus that enhances HSZ's calculus, and give a formalization of soundness and completeness without establishing translations. We again use a slightly different termination argument as the measure from HSZ could become negative in some cases. This does not necessarily mean that HSZ's argument is incorrect, because a negative lower bound might still be given. However, we were not able to find it in the paper.

▶ **Definition 3.9.** *Tableau $S4^{\mathcal{TH}}$ is defined as in Figure 4. The condition $\varphi \notin H$ in the transition rule is called a "loop-check". H, S and A are called a history, a signature, and ancestors of the sequent respectively. A signature is a pair of formula and list of formulas, but can be empty. The ancestors is a list of non-empty signatures. For each rule that is not id or S4, the χ in its upper sequent $A \,||\, S \,||\, H \,||\, \Sigma \mid \chi, \Gamma$ is called a principal formula.*

▶ **Definition 3.10.** *A sequent $A \,||\, S \,||\, H \,||\, \Sigma \mid \Gamma$ is satisfiable if $\Sigma \cup \Gamma$ is satisfiable. Given a sequent s, we refer to its fields by the projection notation (e.g., $s.\Gamma$).*

3.3.1 Downward propagation

The transition rule of $S4^{\mathcal{TH}}$ prevents us from using the termination argument for $KT^{\mathcal{TH}}$, because the degree of $\Sigma \mid \Gamma$ can remain unchanged, and the length of Γ can increase. However, $S4^{\mathcal{TH}}$ has some nice properties that are helpful to design a termination argument.

▶ **Theorem 3.11.** *Let $A \,||\, S \,||\, H \,||\, \Sigma \mid \Gamma$ be a sequent generated by $S4^{\mathcal{TH}}$ from root $A' \,||\, S' \,||\, H' \,||\, \Sigma' \mid \Gamma'$. Then*

1. *Σ contains no duplicate elements. H contains no duplicate elements.*
2. *Σ and H are sublist permutations of $cl(\Gamma')$.*
3. *$\Gamma \subseteq cl(\Gamma')$.*

$$(id)\ \frac{A \mid\mid S \mid\mid H \mid\mid \Sigma \mid n, \neg n, \Gamma}{\text{unsatisfiable}}$$

$$(\wedge)\ \frac{A \mid\mid S \mid\mid H \mid\mid \Sigma \mid \varphi \wedge \psi, \Gamma}{A \mid\mid \varepsilon \mid\mid H \mid\mid \Sigma \mid \varphi, \psi, \Gamma} \qquad (\vee)\ \frac{A \mid\mid S \mid\mid H \mid\mid \Sigma \mid \varphi \vee \psi, \Gamma}{A \mid\mid \varepsilon \mid\mid H \mid\mid \Sigma \mid \varphi, \Gamma \quad A \mid\mid \varepsilon \mid\mid H \mid\mid \Sigma \mid \psi, \Gamma}$$

$$(\Box, \text{new})\ \frac{A \mid\mid S \mid\mid H \mid\mid \Sigma \mid \Box\varphi, \Gamma}{A \mid\mid \varepsilon \mid\mid \varnothing \mid\mid \Box\varphi, \Sigma \mid \varphi, \Gamma}\ (\Box\varphi \notin \Sigma)$$

$$(\Box, \text{dup})\ \frac{A \mid\mid S \mid\mid H \mid\mid \Sigma \mid \Box\varphi, \Gamma}{A \mid\mid \varepsilon \mid\mid H \mid\mid \Sigma \mid \varphi, \Gamma}\ (\Box\varphi \in \Sigma)$$

$$(S4)\ \frac{A \mid\mid S \mid\mid H \mid\mid \Sigma \mid \Diamond\varphi, \Gamma}{(\varphi, \Sigma), A \mid\mid (\varphi, \Sigma) \mid\mid \varphi, H \mid\mid \Sigma \mid \varphi, \Sigma}\ (\varphi \notin H)$$

where Γ in (S4) is a set of literals not containing a contradictory pair

Figure 4 Tableau $\mathrm{S4}^{\mathcal{TH}}$.

A paper proof of Theorem 3.11 could use induction on the $\mathrm{S4}^{\mathcal{TH}}$ rules using these three properties simultaneously. In the formalization, this is handled by a downward propagation as in Section 3.2.

▶ **Definition 3.12.** *Let $A \mid\mid S \mid\mid H \mid\mid \Sigma \mid \Gamma$ be a sequent generated by $S4^{\mathcal{TH}}$ from root $A' \mid\mid S' \mid\mid H' \mid\mid \Sigma' \mid \Gamma'$. We use $\circ$ for function composition, l for the length function and cl for the closure function in Definition 2.2. The size of $A \mid\mid S \mid\mid H \mid\mid \Sigma \mid \Gamma$ is a triple*

$$size(A \mid\mid S \mid\mid H \mid\mid \Sigma \mid \Gamma) := (l \circ cl(\Gamma') - l(\Sigma), l \circ cl(\Gamma') - l(H), l(\Gamma))$$

By Theorem 3.11, *size* is a well-defined function from sequents to $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$.

▶ **Theorem 3.13.** *Let u be the upper sequent and l the lower sequent of a non-id rule in $S4^{\mathcal{TH}}$. Let $<_{lex}$ be the lexicographic order on $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$. Then*

$$size(l) <_{lex} size(u)$$

In addition to Theorem 3.13, which suffices to prove termination, $\mathrm{S4}^{\mathcal{TH}}$ has more properties that help prove soundness and completeness. These properties do not need to be proved by referring to each other, but we gather them together as they are all properties about sequents, which can be handled by a downward propagation.

▶ **Theorem 3.14.** *Let $A \mid\mid S \mid\mid H \mid\mid \Sigma \mid \Gamma$ be a sequent generated by $S4^{\mathcal{TH}}$. Then*

1. *Σ contains only $\Box$-formulas.*
2. *For every $\varphi \in H$, $(\varphi, \Sigma) \in A$.*
3. *If $S \neq \varepsilon$ then $fst(S) \in \Gamma$ and $snd(S) \subseteq \Gamma$.*

As for a sequent of $\mathrm{KT}^{\mathcal{TH}}$, a sequent of $\mathrm{S4}^{\mathcal{TH}}$ can now be defined formally as follows.

```
structure sseqt : Type :=
(goal : list nnf)
(a : list psig) -- psig is the signature, which is of form (d, b)
(s : sig) -- sig := option psig
(h b m: list nnf)
(ndh : list.nodup h) -- nodup says there is no duplicate elements
```

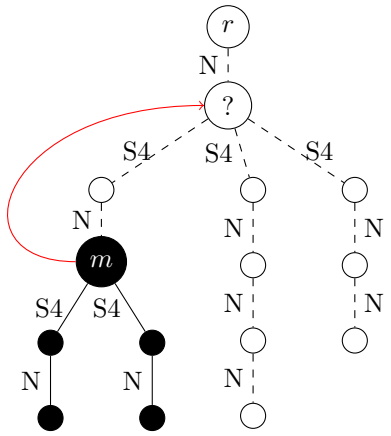


Figure 5 Edges labeled with S4 are an application of transition rule, N an application of non-transition rule. The red edge indicates that a loop-check is triggered at node m and a request is made. Black nodes are nodes with tree models constructed, and white nodes do not have a tree structure yet and their statuses are unknown to m. The node labeled r is the root.

```
(ndb : list.nodup b)
(sph : h <+~ closure goal) -- <+~ denotes sublist permutation
(spb : b <+~ closure goal)
(sbm : m ⊆ closure goal)
(ha : ∀ φ ∈ h, (⟨φ, b⟩ : psig) ∈ a)
(hb : box_only b)
-- dsig takes a signature (d, b) and a proof h, and returns d
(ps₁ : Π (h : s ≠ none), dsig s h ∈ m)
-- bsig takes a signature (d, b) and a proof h, and returns b
(ps₂ : Π (h : s ≠ none), bsig s h ⊆ m)
```

3.3.2 Upward propagation

Before diving into correctness, we give an informal view of the problems caused by S4. One essential difference between $\mathrm{S4}^{\mathcal{TH}}$ and $\mathrm{KT}^{\mathcal{TH}}$ is that when there are no rules applicable to a sequent l, in $\mathrm{KT}^{\mathcal{TH}}$ a provably correct model for l can immediately be constructed, but in $\mathrm{S4}^{\mathcal{TH}}$ this is not true. In $\mathrm{S4}^{\mathcal{TH}}$, there are two cases where no rules are applicable. The first case is that Γ contains only literals in the current sequent $A \mid\mid S \mid\mid H \mid\mid \Sigma \mid \Gamma$. In this case a tree model m which is a singleton can be constructed, and with some effort we might be able to prove $m \vDash \Sigma \cup \Gamma$.

The second case is that Γ contains not only literals, but also a list of $\Diamond$-formulas $\Diamond D$ such that $D \subseteq H$. This happens when loop-checks are triggered to prevent further computation. The intuition behind this termination is that if the $\Diamond$-formula to be handled occurs in the history, then it must have been handled before. Then a reachability relation is supposed to be established between the potential state that satisfies the current sequent and the potential state that satisfies the resulting sequent of the previous $(S4)$-rule application.

There are three levels of difficulty towards the construction of a provably correct model at this stage. The first is that the current sequent l needs to know where the previous handling happened and what the resulting sequent r was. The second is that even if it knows what r was, a tree model m_l for l cannot be constructed because r is above l and does not

have a tree structure yet. Moreover, m_l is a subtree of m_r and its construction should not refer to that of m_r. The third is that even if we give up the idea of trees and manage to construct a model where a state s_l is supposed to satisfy l (s_r for r resp.), to prove that s_l satisfies l, we need to show that all the $\Box$-formulas in l, when unboxed, are satisfied by s_r. However, whether this is true has not been determined because there could be unexplored branches of r. It is also worth noting that the overall status of r depends on the status of l, which is being determined. This non-well-founded behaviour of S4 is illustrated by Figure 5. Similar difficulties also occur when dealing with other more expressive modal logics such as propositional dynamic logic [29].

We proceed as follows to handle this non-well-foundedness and give a formalization of the correctness of $\mathrm{S4}^{\mathcal{TH}}$.

1. When no rule is applicable to a sequent $l = A \mid\mid S \mid\mid H \mid\mid \Sigma \mid \Gamma$ and Γ contains diamonds, a tree model m is constructed. The tree model comes with three additional pieces of data: the sequent l called *id*, a list of formulas called *htk*, and a list of signatures called *request*. Intuitively, *htk* contains formulas true within m, and *request* contains backward edges representing loops. A *request* can be defined using only H and Σ, without referring to a sequent occurring "above" l, but l does not know whether theses requests can be fulfilled.
2. The model m is then propagated to the upper sequents in the same way it is done for K and KT. For the transition rule, the constructor is applied to obtain a new tree. For the non-transition rules, the tree structure remains the same. The construction of *htk* and *request* are described below.
3. The correctness of m is left open at the time it is constructed, instead, a set P of properties of m is proved. These properties exploit the data contained in l and m, and are preserved by upward propagation. In other words, for each rule of $\mathrm{S4}^{\mathcal{TH}}$, if there is a tree model of the lower sequent with P proved, then a tree model of the upper sequent can also be constructed with P proved.
4. We show that if the root sequent $r = \varnothing \mid\mid \varepsilon \mid\mid \varnothing \mid\mid \varnothing \mid \Gamma$ has a tree model m_r with P proved, then interpretation functions can be defined on a type induced by m_r to construct an S4 model m. It can be proved from P that $m \vDash \Gamma$.

▶ **Definition 3.15.** *Lists htk and request are defined recursively as follows :*

1. *If no rules can be applied to a sequent $A \mid\mid S \mid\mid H \mid\mid \Sigma \mid \Gamma$, or it is the upper sequent of a transition rule, then*

   $$htk = \Gamma$$

   $$request = \{(\varphi, \Sigma) : \varphi \in H\}$$

2. *If htk_l and $request_l$ are defined for the a sequent of a non-transition rule R, then*

   $$htk_u = \{\varphi\} \cup htk_l$$

   $$request_u = request_l$$

   where φ is the principal formula of R.

▶ **Definition 3.16.** *A list l of formulas is called pre-hintikka if the following hold:*

1. *For every propositional variable p, if $p \in l$ then $\neg p \notin l$.*
2. *If $\varphi \wedge \psi \in l$, then $\varphi \in l$ and $\psi \in l$.*
3. *If $\varphi \vee \psi \in l$, then $\varphi \in l$ or $\psi \in l$.*
4. *If $\Box\varphi \in l$, then $\varphi \in l$.*

▶ **Theorem 3.17.** *Let m be a tree model and $A \parallel S \parallel H \parallel \Sigma \mid \Gamma$ its id. Then $m.htk$ is pre-hintikka and $\Gamma \subseteq m.htk$.*

As for sequents, Theorem 3.17 can be part of the definition of a tree model. We put *id*, *htk*, and Theorem 3.17 into a single package called `info`, as they are the information about the state being constructed. The final definition of an S4 tree model is as follows:

```
structure info : Type :=
(id : sseqt)
(htk : list nnf)
(hhtk : pre_hintikka htk)
(mhtk : Γ.m ⊆ htk)

inductive tmodel
| cons : info → list tmodel → list psig → tmodel
```

In order to describe the properties P in step 3, we need one more definition.

▶ **Definition 3.18.** *Let $m := cons\ i\ l\ r$ be a tree model. A tree model s is a child of m if $s \in l$. The descendant relation is the transitive closure of the child relation.*

The following non-trivial properties are the key to the correctness proof. We refer to the *id*, *htk* and *request* of a tree model m by the projections $m.id$, $m.htk$ and $m.request$ respectively.

▶ **Theorem 3.19.** *Let m be a tree model constructed in the way described above. Then*

1. *If s is a child of m and $\varphi \in m.id.\Sigma$, then $\varphi \in s.htk$.*
2. *If s is a child of m and $\Box\varphi \in m.htk$, then $\Box\varphi \in s.htk$.*
3. *If $\Diamond\varphi \in m.htk$, then either there exists a Δ such that $(\varphi, \Delta) \in m.request$ or there exists a child s of m such that $\varphi \in s.htk$.*
4. *If $(\gamma, \Delta) \in m.request$ and $\Box\varphi \in m.htk \cup m.id.\Sigma$, then $\Box\varphi \in \Delta$.*
5. *$m.request \subseteq m.id.A$.*

Property 2 requires property 1 as a lemma, and property 5 needs property 2 from Theorem 3.14. We omit the proof of Theorem 3.19, but give a proof sketch of the following substantial theorem, which illustrates that well-founded reasoning is achieved eventually.

▶ **Theorem 3.20** (fulfillment). *Let m be a tree model constructed in the way described above and s be a descendant of m. For every $r \in s.request$, either $r \in m.id.A$, or there exists a descendant d of m such that $r = d.id.S$.*

Proof. By induction on the construction of m. In the base case, there is no descendant of m. If m is constructed by non-transition rules, the theorem holds trivially because the tree structure, $m.id.A$ and *request* remain the same. Suppose m is constructed by the transition rule. Let s be a descendant of m. Then s is either a child of m or there exists a child c of m such that s is a descendant of c. In the first case, we proceed by cases on whether r is the head of $s.id.A$: if so, then s itself is the witness of a qualified descendant, else $r \in m.id.A$. In the second case, we apply the inductive hypothesis and proceed by cases once more. ◀

The fulfilment theorem tells us that every request is eventually fulfilled by the tree model constructed at the root. This is because the root sequent has an empty *ancestors* A.

▶ **Theorem 3.21** (global invariant). *Let m be a tree model constructed in the way described above and s a descendant of m. s satisfies the conclusions of Theorem 3.19 and Theorem 3.20.*

Theorem 3.21 is not superfluous for the formalization. Since Theorem 3.19 and Theorem 3.20 are not part of the definition of a tree model, each model knows that it itself satisfies Theorem 3.19 and Theorem 3.20, but has no information about its descendant once the construction is completed. The formalization of Theorem 3.21 is no harder than a proof by intuition – it is simply a property of m whose proof is immediately given by Theorem 3.19 and Theorem 3.20 during the construction of m, and can be kept by the return type.

For convenience, we now define a subtype `rmodel` that combines a tree model and the invariants `ptmodel`, as the invariants are frequently referred to in the following proofs, especially in proving semantic facts about the reachability relation. The evaluation function `val` and reachability relation `reach` are defined on `rmodel` as follows. Note that `reach` is the reflexive transitive closure of the relation `reach_step`. A state s reaches a state t by one step, if t is a child of s, or the *signature* of t is in the *request* of s.

```
def rmodel : Type := {m : tmodel // ptmodel m}

inductive reach_step : rmodel → rmodel → Prop
| fwd (s : rmodel) (i l ba h) : s.1 ∈ l → reach_step ⟨(cons i l ba), h⟩  s
| bwd (s : rmodel) (i l ba h) : (∃ rq ∈ ba, some rq = msig s.1) →
                                reach_step ⟨(cons i l ba), h⟩ s

def reach (s₁ s₂ : rmodel) := rtc reach_step s₁ s₂
```

Given a tree model m, the carrier set M of the final S4 model induced by m is all the `rmodel`s whose `tmodel` part is either m or a descendant of m. The final S4 model ready to be proved correct is as follows.

```
def builder (m : tmodel) : S4 {x : rmodel // x.1 = m ∨ desc x.1 m} :=
{val := λ v s, var v ∈ htk s.1.1,
 rel := λ s₁ s₂, reach s₁ s₂, -- λ for coercion
 refl := λ s, refl_reach s,
 trans := λ a b c, trans_reach a b c}
```

When there is no confusion, we refer to the *htk* of an element s in M as $s.htk$.

▶ **Theorem 3.22.** *Let m be the tree model returned by the decision procedure called on the root sequent $r = \varnothing \,||\, \varepsilon \,||\, \varnothing \,||\, \varnothing \mid \Gamma$ with Theorem 3.19, Theorem 3.20 and Theorem 3.21 proved. Then for every state s in the induced model M, if $\varphi \in s.htk$ then $s \Vdash \varphi$. In particular, when viewed as an element of M, $m \vDash \Gamma$.*

Proof. By induction on the construction of formulas. This one makes use of everything proved so far, especially the invariants. $m \vDash \Gamma$ because $\Gamma \subseteq m.htk$ by Theorem 3.17. ◀

4 Backjumping

The verified decision procedures described above can be equipped with provably correct optimizations as well. We now describe how backjumping [3] as an optimization can be integrated to gain exponential speedups.

Backjumping reduces search space by preventing recursive calls on the right branch of an $(\vee)$ rule when its status can already be determined by analyzing the information propagated from the left branch. If the left branch is open, then there is no need for backjumping as we don't have to explore the right branch. Backjumping is triggered only on closed branches. On

the other hand, all the significant changes we made to verify $\mathrm{KT}^{\mathcal{TH}}$ and $\mathrm{S4}^{\mathcal{TH}}$ happen only on the construction of models, namely the open branches, and the closed branches remain almost the same. Due to this nature, the verification of backjumping does not interfere with the proofs in Section 3.2 and Section 3.3. Such a verification for K can be ported to KT and S4 without too many changes. We formalize backjumping for K as an example, and leave the extension to KT and S4 as future work.

We define a notion of responsibility for each of the rules of $\mathrm{K}^{\mathcal{T}}$. Each sequent is assigned a list of formulas, called a marking set, representing the formulas responsible for contradictions. The construction of a marking set is an upward propagation.

▶ **Definition 4.1** (responsibility). *A marking set M is recursively defined on closed branches as follows. The φ and ψ refer to their corresponding occurrences in $K^{\mathcal{T}}$ defined in Figure 1.*

1. *For the id rule, $M = \{p, \neg p\}$.*
2. *Let M_l be the marking set of the lower sequent of the $\wedge$-rule.*

$$M = \begin{cases} \{\varphi \wedge \psi\} \cup M_l & \text{if } \varphi \in M_l \text{ or } \psi \in M_l \\ M_l & \text{otherwise} \end{cases}$$

3. *Let M_l/M_r be the marking sets of the left/right lower sequent of the $\vee$-rule respectively.*

$$M = \begin{cases} M_l \cup M_r \cup \{\varphi \vee \psi\} & \text{if } \varphi \in M_l \text{ or } \psi \in M_r \\ M_l \cup M_r & \text{otherwise} \end{cases}$$

4. *Let l be the first unsatisfiable lower sequent of the transition rule, with a marking set M_l:*

$$M = \Diamond(l.head) \cup \Box(l.tail \cap M_l)$$

The idea of backjumping is that if the left principal formula (i.e., φ) of the ($\vee$) rule is not in the marking set of the left lower sequent, then the upper sequent is unsatisfiable. We strengthen this claim and prove the following:

▶ **Theorem 4.2** (marking). *For each sequent Γ, if a sublist Δ of Γ contains nothing in the marking set if defined, then $\Gamma - \Delta$ is unsatisfiable.*

Formally, Theorem 4.2 is defined as:

```
def pmark (Γ m : list nnf) :=
∀ Δ, (∀ δ ∈ Δ, δ ∉ m) → Δ <+ Γ → unsatisfiable (list.diff Γ Δ)
```

The motivation of Theorem 4.2 is that we want to add a new rule BJ standing for backjumping into $\mathrm{K}^{\mathcal{T}}$. The upper sequent is immediately unsatisfiable because Γ is unsatisfiable by Theorem 4.2. Also note that a marking set is defined if and only if the sequent is unsatisfiable.

$$(BJ)\ \frac{\varphi \vee \psi; \Gamma}{\varphi; \Gamma} \quad \text{if } \varphi \notin M_l$$

In terms of the formalization of S4, Theorem 4.2 is an invariant. It is proved during the upward propagation along with the construction of the marking set, but this time everything happens in the closed branches. The formalization of Theorem 4.2 is difficult, mainly due to the reasoning about list difference. We omit the proof here as it should be conceptually clear how it can be proved by induction. One thing to note is that a marking set of BJ also needs to be defined and proved to respect Theorem 4.2 because BJ now takes part in the computation. This can be achieved by using case 3 of Definition 4.1 assuming M_r is empty. The corresponding proof of Theorem 4.2 is then straightforward.

Table 1 Results on the LWB benchmark for K (left) and S4 (right).

Subclass	K	K (backjumping)	FaCT++
branch_n	3	5	10
branch_p	1	3	10
d4_n	5	5	21
d4_p	6	7	21
dum_n	18	18	21
dum_p	9	17	21
grz_n	21	21	21
grz_p	6	7	21
lin_n	3	4	21
lin_p	6	7	21
path_n	10	10	21
path_p	2	12	21
ph_n	3	3	13
ph_p	2	3	7
poly_n	20	20	21
poly_p	19	21	21
t4p_n	7	7	21
t4p_p	7	12	21

Subclass	S4	FaCT++
45_n	0	21
45_p	1	21
branch_n	3	6
branch_p	6	7
grz_n	21	21
grz_p	0	21
ipc_n	3	10
ipc_p	3	10
md_n	3	10
md_p	4	4
path_n	2	15
path_p	1	21
ph_n	3	7
ph_p	2	6
s5_n	2	21
s5_p	21	21
t4p_n	0	21
t4p_p	0	21

5 Evaluation

We evaluated the performance of the verified decision procedures for K and S4 against FaCT++ [28], using the Logic Work Bench (LWB) benchmarks [5]. FaCT++ is a state-of-the-art reasoner for modal description logics. The LWB benchmarks are widely used for measuring the performance of modal reasoners [20] [15]. There are 18 subclasses of problems in the benchmark. Each subclass contains 21 problems, and each problem is harder to solve than the previous ones within the same subclass. We perform the tests on an Intel 2.20 GHz CPU with 2GB of memory. The time limit for each problem in a subclass is set to be 100 seconds and Table 1 shows the most difficult problem solved from each subclass by each prover within this limit. Thus the first row of the left hand table shows that our verified provers K and K (with backjumping) could solve 3 and 5 problems, respectively, while FaCT++ could solve 10, with each problem taking at most 100 seconds.

It is not surprising that FaCT++ outperforms the verified decision procedures on almost every problem. Figure 6 displays a fragment of a typical profile of the verified S4 decision procedure called on a problem. It can be seen that nearly 50% to 75% of the time is spent on `dec_eq_nnf`, which is called heavily in list operations such as `list.erase`. This suggests that future improvements of efficiency can include using better data structures such as arrays or hash tables instead of lists, as well as implementing other algorithmic optimizations such as unit propagation, semantic branching [28] and better ordering heuristics [27]. On the other hand, we see from Table 1 that the decision procedure for K with backjumping dominates the vanilla one in performance and backjumping is never worse. In particular, on the subclasses `dum_p` and `path_p`, there is a huge boost given by backjumping.

6 Conclusion and future work

We have presented verified decision procedures for three basic modal logics and shown how to handle loop-checking and backjumping. All of these decision procedures are executable, and are proved to be sound, complete and terminating. Backjumping has been implemented and

```
#eval execution took 15.6s
    15552ms    99.9%   tableau
    ...
    10703ms    68.8%   dec_eq_nnf
     9491ms    61.0%   list.decidable_mem
    ...
```

Figure 6 A fragment of a typical profile.

verified for K, and can be ported to KT and S4. All of these decision procedures return a concrete Kripke model when the input set of formulas is satisfiable, and a proof constructed via the tableau rules witnessing unsatisfiability otherwise. In fact, although the following well-known theorem is not formalized, it is implied by the formalization:

▶ **Theorem 6.1** (finite model K, KT, S4)**.** *Every satisfiable formula is satisfiable in a finite model.*

It should be clear that each satisfiable formula is witnessed by a tree model, which is a finite object. It can also be seen that the valuation functions and reachability relations constructed by the decision procedures for K and KT are computable. In the case of S4, this is a bit subtle because the transitive closure relation is not necessarily computable. However, since the descendants of a tree model form a finite set, by checking their *request*s and *signature*s one by one, we do have a way to compute reachability. We leave it to future work to have an explicit formalization of this for completeness.

Tableaux with histories offer us convenient tools for formalizing correctness of decision procedure for modal logics, but they also introduce some inefficiency. Comparing to a sequent of $\mathrm{S4}^{\mathcal{T}}$, a sequent of $\mathrm{S4}^{\mathcal{TH}}$ contains more information and takes time to construct. The extra information helps with verification but slows down the implementation. Therefore, future work also includes finding a balance point between the ease of formalization and computational efficiency of these decision procedures, and of course, porting backjumping to them would be an external boost.

One last thing to notice is that the expressiveness of S4 allows us to apply the verified decision procedures to more than modal logics. Since S4 is topologically complete [22], a translation between Kripke semantics and topological semantics can be established. It can be shown that a formula φ has a topological model if and only if it has an S4 model. We have also done half of this translation in our formalization. Another translation called the Gödel-McKinsey-Tarski translation is given in McKinsey and Tarski [23]. It is a translation between propositional intuitionistic logic and modal logic S4, and preserves theoremhood. Consequently, if the translation is formalized, then a verified decision procedure for S4 also gives us a verified decision procedure for intuitionistic propositional logic. The formalization of S4 opens the possibility of promising cross-field applications, and we leave the implementation of these as future work.

References

1 Samson Abramsky. Domain Theory and the Logic of Observable Properties. *CoRR*, abs/1112.0347, 2011. `arXiv:1112.0347`.

2 José-Antonio Alonso, Joaquín Borrego-Díaz, María-José Hidalgo, Francisco-Jesus Martín-Mateos, and José-Luis Ruiz-Reina. A Formally Verified Prover for the ALC Description Logic. In *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs'07, pages 135–150, Berlin, Heidelberg, 2007. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=1792233.1792244`.

3 Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, New York, NY, USA, 2003.

4 Franz Baader and Bernhard Hollunder. A terminological knowledge representation system with complete inference algorithms. In Harold Boley and Michael M. Richter, editors, *Processing Declarative Knowledge*, pages 67–86, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

5 Peter Balsiger, Alain Heuerding, and Stefan Schwendimann. A Benchmark Method for the Propositional Modal Logics K, KT, S4. *Journal of Automated Reasoning*, 24(3):297–317, April 2000. doi:10.1023/A:1006249507577.

6 Bruno Bentzen. A Henkin-style completeness proof for the modal logic S5, 2019. URL: https://github.com/bbentzen/mpl.

7 Jasmin Christian Blanchette, Mathias Fleury, Peter Lammich, and Christoph Weidenbach. A Verified SAT Solver Framework with Learn, Forget, Restart, and Incrementality. *Journal of Automated Reasoning*, 61(1):333–365, June 2018. doi:10.1007/s10817-018-9455-7.

8 Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

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, *Automated Deduction - CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing.

10 Paulien de Wind. Modal Logic in Coq. Master's thesis, Vrije Universiteit Amsterdam, 2001.

11 Christian Doczkal and Joachim Bard. Completeness and Decidability of Converse PDL in the Constructive Type Theory of Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 42–52, New York, NY, USA, 2018. ACM. doi:10.1145/3167088.

12 Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A fully verified executable LTL model checker. In *International Conference on Computer Aided Verification*, pages 463–478. Springer, 2013.

13 Rajeev Goré. Tableau Methods for Modal and Temporal Logics. In Marcello D'Agostino, Dov M. Gabbay, Reiner Hähnle, and Joachim Posegga, editors, *Handbook of Tableau Methods*, pages 297–396. Springer Netherlands, Dordrecht, 1999. doi:10.1007/978-94-017-1754-0_6.

14 Rajeev Goré. AND-OR tableaux for fixpoint logics with converse: LTL, CTL, PDL and CPDL. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning*, pages 26–45, Cham, 2014. Springer International Publishing.

15 Rajeev Goré, Kerry Olesen, and Jimmy Thomson. Implementing Tableau Calculi Using BDDs: BDDTab System Description. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning*, pages 337–343, Cham, 2014. Springer International Publishing.

16 Alain Heuerding, Michael Seyfried, and Heinrich Zimmermann. Efficient loop-check for backward proof search in some non-classical propositional logics. In P. Miglioli, U. Moscato, D. Mundici, and M. Ornaghi, editors, *Theorem Proving with Analytic Tableaux and Related Methods*, pages 210–225, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

17 M. J. Hidalgo-Doblado, J. A. Alonso-Jiménez, J. Borrego-Díaz, F. J. Martín-Mateos, and J. L. Ruiz-Reina. Formally Verified Tableau-Based Reasoners for a Description Logic. *Journal of Automated Reasoning*, 52(3):331–360, March 2014. doi:10.1007/s10817-013-9291-8.

18 Simon Jantsch and Michael Norrish. Verifying the LTL to Büchi Automata Translation via Very Weak Alternating Automata. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving*, pages 306–323, Cham, 2018. Springer International Publishing.

19 Mark Kaminski, Thomas Schneider, and Gert Smolka. Correctness and Worst-Case Optimality of Pratt-Style Decision Procedures for Modal and Hybrid Logics. In Kai Brünnler and George Metcalfe, editors, *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 196–210, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

20 Mark Kaminski and Tobias Tebbi. InKreSAT: Modal reasoning via incremental reduction to SAT. In Maria Paola Bonacina, editor, *CADE-24*, volume 7898 of *LNCS*, pages 436–442. Springer, June 2013.

21 LudvikGalois. A verfied tableau prover for classical propositional logic, 2018. URL: `https://github.com/LudvikGalois/coq-CPL-NNF-tableau`.

22 J. C. C. McKinsey and Alfred Tarski. The Algebra of Topology. *Annals of Mathematics*, 45(1):141–191, 1944. URL: `http://www.jstor.org/stable/1969080`.

23 John CC McKinsey and Alfred Tarski. Some theorems about the sentential calculi of Lewis and Heyting. *The journal of symbolic logic*, 13(1):1–15, 1948.

24 John-Jules Ch Meyer and Wiebe Van Der Hoek. *Epistemic Logic for AI and Computer Science*. Cambridge University Press, New York, NY, USA, 1995.

25 Alexander Schimpf, Stephan Merz, and Jan-Georg Smaus. Construction of Büchi Automata for LTL Model Checking Verified in Isabelle/HOL. In *TPHOLs*, 2009.

26 Anders Schlichtkrull. Formalization of the Resolution Calculus for First-Order Logic. *Journal of Automated Reasoning*, 61(1):455–484, June 2018. `doi:10.1007/s10817-017-9447-z`.

27 Dmitry Tsarkov and Ian Horrocks. Ordering Heuristics for Description Logic Reasoning. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, IJCAI'05, pages 609–614, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc. URL: `http://dl.acm.org/citation.cfm?id=1642293.1642391`.

28 Dmitry Tsarkov and Ian Horrocks. FaCT++ description logic reasoner: System description. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning*, pages 292–297, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

29 Florian Widmann. *Tableaux-based Decision Procedures for Fixed Point Logics*. PhD thesis, Australian National University, 2010.

30 Yoshifumi Yuasa, Yoshinori Tanabe, Toshifusa Sekizawa, and Koichi Takahashi. Verification of the Deutsch-Schorr-Waite Marking Algorithm with Modal Logic. In *Proceedings of the 2Nd International Conference on Verified Software: Theories, Tools, Experiments*, VSTTE '08, pages 115–129, Berlin, Heidelberg, 2008. Springer-Verlag. `doi:10.1007/978-3-540-87873-5_12`.

Characteristic Formulae for Liveness Properties of Non-Terminating CakeML Programs

Johannes Åman Pohjola
Data61/CSIRO, Sydney, Australia
University of New South Wales, Sydney, Australia
johannes.amanpohjola@data61.csiro.au

Henrik Rostedt
Chalmers University of Technology, Gothenburg, Sweden

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

Abstract

There are useful programs that do not terminate, and yet standard Hoare logics are not able to prove liveness properties about non-terminating programs. This paper shows how a Hoare-like programming logic framework (characteristic formulae) can be extended to enable reasoning about the I/O behaviour of programs that do not terminate. The approach is inspired by transfinite induction rather than coinduction, and does not require non-terminating loops to be productive. This work has been developed in the HOL4 theorem prover and has been integrated into the ecosystem of proof tools surrounding the CakeML programming language.

2012 ACM Subject Classification Software and its engineering → Software verification; Theory of computation → Higher order logic; Theory of computation → Separation logic

Keywords and phrases Program verification, non-termination, liveness, Hoare logic

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.32

Supplement Material This work has been developed in HOL4; the sources are at `https://code.cakeml.org`

Funding *Johannes Åman Pohjola*: This work was sponsored in part by the U.S. Defense Advanced Research Projects Agency (DARPA). The views expressed are those of the authors and do not reflect the official policy or position of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government. Approved for Public Release, Distribution Unlimited.
Magnus O. Myreen: This work was supported by the Swedish Foundation for Strategic Research.

Acknowledgements We are grateful to Robert Sison and the anonymous reviewers for many constructive and insightful comments.

1 Introduction

Consider the following non-terminating ML program that prints the letter `y` forever.

```
fun yes() = (put_line "y"; yes());
val () = yes();
```

This program has the same behaviour as the default configuration of the Unix tool `yes`.

The `yes` program highlights a peculiar omission in Hoare-style programming logics to date: with only a few exceptions (Section 7), Hoare-like logics have only focused on reasoning about terminating programs or proving absence of bad behaviours. Few Hoare logics can state (let alone prove) that the `yes` program (1) will not terminate and (2) will produce a never-ending stream of `y` characters as output. Note that (2) is a liveness property.

One can argue that correctness is not important for toy examples such as `yes`. However, there are real-world programs that are both non-terminating and where correctness is important. Examples include embedded controllers, web servers, network filters and other software that is part of some device or infrastructure.

The fact that non-terminating behaviours are important is acknowledged in compiler verification where it is expected/common to prove that compilation preserves both terminating and non-terminating behaviours of the compiled programs; the CompCert [25] and CakeML [36] compilers are proved to preserve both types of behaviours.

In this paper we present how reasoning about total correctness of non-terminating programs can be integrated into and used in the context of a Hoare-like programming logic. Specifically, we describe how a proved-to-be-sound Hoare-style programming logic framework (characteristic formulae for CakeML) has been extended to enable reasoning about the I/O behaviour of non-terminating programs, thus enabling proofs of correctness properties such as (1) and (2).

For the `yes` program, our extended framework allows us to prove the following correctness theorem. The theorem is stated as a Hoare triple, where the precondition assumes that an I/O stream exists and nothing has been written to it, and the code is the application of function yes to an arbitrary argument *arg*. The postcondition is the interesting part here: we specify with POSTd that the program does not terminate. Furthermore, we assert that the trace of *io* produced by the diverging (i.e. non-terminating) execution is the infinite lazy list obtained by repeating the I/O event put_str_event "y\n" forever.

$$
\begin{array}{l}
\vdash \{|\mathsf{io\_events}\ []|\} \\
\quad \mathsf{yes} \cdot [\mathit{arg}] \\
\quad \{|\mathsf{POSTd}\ \mathit{io}.\ \mathit{io} = \mathsf{lrepeat}\ [\mathsf{put\_str\_event}\ \texttt{"y\textbackslash n"}]|\}
\end{array}
$$

In conventional Hoare logics, postconditions make a statement about final program states. However, non-terminating programs do not have final states, and the only interesting observation that can be made is what I/O they produce. Here the POSTd-postconditions make a statement about a possibly infinite trace of I/O events. One can think of this trace as the I/O events produced by an infinite execution of the program. As we will see later, formally POSTd-postconditions make a statement about *the least upper bound* of all I/O traces the program produces when allowed to run for different lengths of time.

Contributions

This paper makes the following contributions:

- It shows how a Hoare-like logic, characteristic formulae (CF) for CakeML, can be extended to enable correctness proofs for non-terminating programs. The approach is inspired by transfinite induction rather than coinduction, and does not require non-terminating loops to be productive. Our extension to CF enables reasoning about non-terminating programs in the same setting as terminating programs. We support postconditions with conditional (non-)termination, including conditions on external state such as the length or contents of input streams.
- Proofs of non-termination are, in two steps, turned into proofs about terminating functions. The first automatically step transforms the function under consideration into a repeat combinator applied to a terminating step function. The second step is to interactively prove that each execution of the (terminating) step function has behaviours that can

be composed to describe the infinite behaviour of the original function. Currently, this approach is limited to tail-recursive functions and does not consider mutual or higher-order recursion.[1]

- We demonstrate the use of CF on examples of non-terminating programs. The most complex example is a filter component for systems built on the verified seL4 microkernel. This filter component had an unwieldy and overly complicated proof before CF could be used, but now has a manageable proof that avoids reasoning directly at the level of the operational semantics.

2 Bird's-eye view of yes verification

We start with a high-level summary of the user experience when verifying the `yes` program. Subsequent sections explain the technical setup and several more interesting examples.

To prove the `yes` program, the user of the proof tools first applies a tactic that runs a verified source-to-source transformation on the recursive function `yes`. The transformation converts the goal we want (i.e. the theorem statement about `yes` from above) to a goal that talks about an application of repeat to a non-recursive function. Here `repeat` is defined as `fun repeat f x = repeat f (f x)`. This helps isolate the behaviour of each iteration of `yes`. The new goal statement is roughly:

$$\{|\mathsf{io\_events}\ []|\}$$
```
 repeat (fn () => put_line "y"; ()) ()
```
$$\{|\mathsf{POSTd}\ io.\ io = \mathsf{lrepeat}\ [\mathsf{put\_str\_event}\ \texttt{"y\textbackslash n"}]|\}$$

The tactic then invokes a general theorem that can reduce any such goal about `repeat` and POSTd into a goal about the terminating executions of its function argument. At this point, the proof goal splits into *two subgoals* and the user needs to instantiate three existentially quantified variables: *events*, *Hs* and *vs* (which will be explained below).

The first subgoal is a Hoare triple which asserts that each execution of the loop body respects *events*, *Hs* and *vs*. The Hoare triple is roughly the following. Think of *Hs i* as the precondition for the *i*th iteration of the loop. Here *events i* is a list of I/O events produced on the *i*th iteration; and *vs i* is a predicate that the argument given as input on iteration *i* satisfies. The POSTv-postcondition requires that the function returns normally and *vs* $(i+1)$ *retv* requires that the function produces the next argument.

$$\forall i.\ \{|Hs\ i * \mathsf{io\_events}\ []|\}$$
```
   (fn () => put_line "y"; ()) · [vs i]
```
$$\{|\mathsf{POSTv}\ retv.\ Hs\ (i+1) * \mathsf{io\_events}\ (events\ i) * \langle vs\ (i+1)\ retv\rangle|\}$$

The second subgoal requires the user to prove that the infinite concatenation lflatten of the list consisting of all I/O event lists, i.e. lgenlist *events* None, satisfies the desired postcondition:

$$\mathsf{lflatten}\ (\mathsf{lgenlist}\ events\ \mathsf{None}) = \mathsf{lrepeat}\ [\mathsf{put\_str\_event}\ \texttt{"y\textbackslash n"}]$$

Since each loop iteration behaves the same, we can instantiate *events*, *Hs*, *vs* with constant functions that return [put_str_event "y\n"], the empty heap predicate emp, and equality with the unit value (), respectively. The two subgoals are then easy to prove. Note that the proof of the first subgoal uses standard CF methods because it is about a terminating program.

[1] Note that this is not a significant restriction in practice, because most non-tail-recursive functions that have diverging semantics will actually terminate with an out-of-stack error message.

3 Background and new technical setup

3.1 Heaps and characteristic formulae

Characteristic formulae [5] (CF for short) is a technique for program verification that is based around a function that given a program p produces a predicate $\mathsf{cf}\ p$ called the *characteristic formula* of p. The idea is that in order to prove validity of the Hoare triple $\{\!|P|\!\}\ p\ \{\!|Q|\!\}$, it suffices to prove $\mathsf{cf}\ p\ P\ Q$. This helps because $\mathsf{cf}\ p$ is a higher-order logic formula and not a program; that is, we reduce reasoning about programs to shallowly embedded formulae in the meta-logic[2] that make no direct reference to the program source code. These formulae are typically much easier to reason about in a proof assistant. The present paper extends the work of Guéneau et al. [15] on characteristic formulae for CakeML, to which we refer for more background and details.

A *heap* is a set of *heap parts*; a heap part is either a memory cell $\mathsf{Mem}\ l\ v$, meaning that the value v is at memory location l, or an external resource $\mathsf{FFI\_part}\ st\ f\ ps\ e$, which describes a part of the world outside the CakeML runtime that can be affected by foreign function calls such as I/O operations. It records the state of the outside world (st), an oracle that models the effects of invoking foreign functions (f), the list of FFI calls it can handle (ps), and a list of the events (e) observed so far. We define the lifting of a boolean c to a heap predicate $\langle c \rangle$ as $\lambda\, s.\ s = \emptyset \wedge c$. Let $l \mapsto v$ denote the heap predicate $\lambda\, h.\ h = \{\ \mathsf{Mem}\ l\ v\ \}$.

The *result* of executing a program is modelled by an element of the datatype res:

$$\mathsf{res}\ =\ \mathsf{Val\ v}\ |\ \mathsf{Exn\ v}\ |\ \mathsf{FFIDiv\ string\ (word8\ list)\ (word8\ list)}\ |\ \mathsf{Div\ (io\_event\ llist)}$$

Programs can return a value (Val), raise an exception (Exn), invoke a foreign function that never returns control to CakeML (FFIDiv), or diverge (Div). When a program diverges, it exhibits a possibly infinite trace of I/O events, represented as a lazy list. Div and FFIDiv are where we extend previous work [15], which only considered values and exceptions. We will focus the presentation on the case of Div.

When we write a Hoare triple $\{\!|P|\!\}\ p\ \{\!|Q|\!\}$, the precondition P is a heap predicate, and the postcondition Q is a function from results to heap predicates. The following abbreviations are convenient for writing postconditions:

$$(\mathsf{POSTv}\ v.\ Q\ v) \stackrel{\mathsf{def}}{=} (\lambda\, res.\ \mathsf{case}\ res\ \mathsf{of}\ \mathsf{Val}\ v\ \Rightarrow\ Q\ v\ |\ \_\ \Rightarrow\ \langle \mathsf{F} \rangle)$$
$$(\mathsf{POSTd}\ io.\ Q\ io) \stackrel{\mathsf{def}}{=} (\lambda\, res.\ \mathsf{case}\ res\ \mathsf{of}\ \mathsf{Div}\ io\ \Rightarrow\ \langle Q\ io \rangle\ |\ \_\ \Rightarrow\ \langle \mathsf{F} \rangle)$$

For example, the Hoare triple $\{\!|\langle \mathsf{T} \rangle|\!\}\ p\ \{\!|\mathsf{POSTv}\ v.\ \langle \mathsf{int}\ 5\ v \rangle * l \mapsto v|\!\}$ is true if from every initial state the program p returns 5 and, moreover, the memory location l contains 5. Here $*$ denotes separating conjunction. Note that in POSTd, $Q\ io$ is a predicate and not a heap predicate; this reflects the fact that divergent programs have no final state.

The characteristic formula is generated by a straightforward recursion on the syntactic structure of the program. To give the flavour, we show the characteristic formula of a sequential composition $e_1\ ;\ e_2$:

$$\begin{array}{l}\mathsf{cf}\ (e_1\ ;\ e_2) \stackrel{\mathsf{def}}{=} \\ \quad \mathsf{local} \\ \quad\quad (\lambda\, H\ Q. \\ \quad\quad\quad \exists\, Q'. \\ \quad\quad\quad\quad (\mathsf{cf}\ e_1\ H\ Q' \wedge Q' \Rightarrow_{\neg v} Q) \wedge \\ \quad\quad\quad\quad \forall\, xv.\ \mathsf{cf}\ e_2\ (Q'\ (\mathsf{Val}\ xv))\ Q)\end{array}$$

[2] In our case, the meta-logic is higher-order logic.

local is, intuitively, the closure of a predicate under separating conjunction. This mimics the frame rule of separation logic, allowing us to disregard irrelevant parts of the heap in subproofs. The remainder of the formula says that there must be an intermediate postcondition Q' admitted by the characteristic formula of e_1 such that (1) Q' *res* implies Q *res* if *res* is not a value ($\Rightarrow_{\neg v}$), and (2) the characteristic formula of e_2 admits Q' (Val xv) as precondition and Q as postcondition for all values xv. To see how this formula copes with divergence, note that if Q' is a POSTd then conjunct (2) is vacuous because of the precondition Q' (Val xv) = $\langle$F$\rangle$. Thus, if e_1 diverges then cf (e_1 ; e_2) does not depend on e_2.

Perhaps surprisingly, virtually no changes to the definition of cf are needed to support divergence. This is for two reasons. First, CF for CakeML already supports reasoning about exceptions. Once obtained, the way a Div result propagates through a characteristic formula is exactly analogous to an exception that can never be handled; the reader may check this for the case when e_1 raises an exception in the above sequential composition. The second reason is that cf e does not unfold the definition of functions that are called within e. Instead, the cf of a function application falls back to a Hoare triple about the program:

$$\mathsf{cf}\ (f \cdot v) \stackrel{\text{def}}{=} \mathsf{local}\ (\lambda H\ Q.\ \{\!|H|\!\}\ f \cdot v\ \{\!|Q|\!\})$$

Thus there is no need to accommodate infinite recursion in e by, say, making cf clocked or corecursive. This design means that the CF logic has no native proof rule to deal with recursive calls, terminating or not. This aspect is handled entirely by the meta-logic, which, being higher-order, offers excellent support for induction.

In the above presentation, we have taken the liberty of abstracting away from certain details that are not germane to the issue at hand. In particular, the definition of cf in the formalisation is parameterised by a binding environment mapping variables to values. We will continue to ignore binding environments in the remainder of this presentation. This is because they are mainly a matter of plumbing: the CF user never sees or manipulates binding environments. Readers interested in the gory details may peruse [15] or the formalisation.

3.2 Semantics and soundness

In this section, we will define how we give meaning to Hoare triples, and prove that characteristic formulae are sound with respect to Hoare triples.

The semantics of CakeML is defined in the style of *functional big-step semantics* [30]. That is, the workhorse is a function evaluate which given an initial state and a program returns a final state and a result. It is structured much like an interpreter, but is not necessarily executable. Since evaluate is a function, we need to make sure it is terminating, and since we also wish to give semantics to non-terminating programs, evaluate is *clocked*: it is parameterised on a natural number *ck* that is decremented whenever evaluate consumes a function call, and if *ck* is 0, evaluate terminates with a special timeout result.[3] The top-level semantics of a program is defined in terms of evaluate by quantifying over possible clock values: a diverging program is one that times out for every *ck*. A terminating program is one where for some *ck*, evaluate terminates with a non-timeout result. This intuition is formalised in the definition of evaluate_to_heap:

[3] In the CakeML language, (recursive) function calls are the only language constructs that can lead to divergence. There are no while loops or similar constructs.

$$\begin{array}{l}\mathsf{evaluate\_to\_heap}\ st\ exp\ heap\ (\mathsf{Val}\ v) \stackrel{\mathsf{def}}{=} \\ \quad \exists\, ck\ st'.\ \mathsf{evaluate}\ ck\ st\ [exp] = (st', \mathsf{Rval}\ [v]) \wedge \mathsf{st2heap}\ st' = heap\end{array}$$

$$\begin{array}{l}\mathsf{evaluate\_to\_heap}\ st\ exp\ heap\ (\mathsf{Div}\ io) \stackrel{\mathsf{def}}{=} \\ \quad (\forall\, ck.\ \exists\, st'.\ \mathsf{evaluate}\ ck\ st\ [exp] = (st', \mathsf{Rerr}\ (\mathsf{Rabort}\ \mathsf{Rtimeout\_error}))) \wedge \\ \quad \mathsf{sup}\ \{\ io\ |\ \exists\, ck.\ io = \mathsf{fromList}\ (\mathsf{fst}\ (\mathsf{evaluate}\ ck\ st\ [exp])).\mathsf{ffi.io\_events}\ \}\ = io\end{array}$$

As a technical detail, evaluate_to_heap also mediates between evaluate's concrete notion of state, and the heap abstraction that our Hoare triples use, via st2heap. Two noteworthy things are happening in the divergence case. First we require that io is the supremum (ordered by prefix inclusion) of the I/O events that the program emits for every clock value. Thus, a Div result represents the limit behaviour of a program as time goes to infinity. Second, the value of *heap* is ignored, because a divergent execution has no final state.

We now have all the machinery required to define Hoare triples in terms of this semantics:

$$\begin{array}{l}\{\!|H|\!\}\ e\ \{\!|Q|\!\} \stackrel{\mathsf{def}}{=} \\ \quad \forall\, st\ h\_i\ h\_k. \\ \qquad \mathsf{split}\ (\mathsf{st2heap}\ st)\ (h\_i, h\_k) \Rightarrow \\ \qquad H\ h\_i \Rightarrow \\ \qquad \exists\, r\ h\_f\ h\_g\ heap.\ \mathsf{split3}\ heap\ (h\_f, h\_k, h\_g) \wedge Q\ r\ h\_f \wedge \mathsf{evaluate\_to\_heap}\ st\ e\ heap\ r\end{array}$$

In words, the Hoare triple $\{\!|H|\!\}\ e\ \{\!|Q|\!\}$ is true if, starting from any initial state which is the disjoint union (split) of a heap h_k and some heap that satisfies the precondition H, the result of evaluating e from this initial state is a heap *heap* and result r such that some subset of the *heap* which is disjoint from h_k satisfies $Q\ r$. Note that h_k recurs in both the pre- and postconditions – this, along with the disjoint unions before and after evaluation, are necessary to make local reasoning sound.

Soundness: the main result that validates the use of characteristic formulae for verification of CakeML programs is:

$$\vdash \mathsf{cf}\ e\ H\ Q \Rightarrow \{\!|H|\!\}\ e\ \{\!|Q|\!\}$$

This extends the soundness result of Guéneau et al. [15] to total-correctness Hoare triples about divergent programs. The main complications were the shifted clocks in the sampling of sup as used in the definition of evaluate_to_heap. The interesting cases to update for Div were `let`, `handle`, `andalso`/`orelse` and function application. Otherwise, the structure of the soundness proof needed some refactoring but did not fundamentally change.

4 Reasoning about divergent programs

When faced with programs that run forever, the traditional techniques for reasoning about loops no longer apply: induction fails because there is no base case, and the While rule of total correctness Hoare logic fails because there is no loop variant. Both of these approaches have the very important practical benefit that they are syntax-directed: they reduce reasoning about loops to reasoning about a single iteration of the loop body.

In this section, we develop the reasoning principles and tools necessary to support such syntax-directed proofs about divergent programs. In doing so, we have two conflicting goals that we must balance:

1. We want to support reasoning about silent loops that don't produce I/O.
2. The user should never see the semantic clock from Section 3.2 when proving a specification, and postconditions should describe no behaviour except the observable I/O behaviour.

The challenge is to meet both while avoiding unsoundness due to circularity. For example, the WHILE rule of Nakata and Uustalu [28] meets the first goal by sacrificing the second: their postconditions describe traces that include internal computation steps such as the evaluation of loop guards. Programming with corecursive functions in proof assistants or total programming languages [37] requires productivity, thus sacrificing the first goal.

Our solution is based on the insight that we can avoid circularity by exploiting the fact that in the evaluation semantics of Section 3.2, silent loops produce a clock tick every iteration. We can hide this tick from the user by considering programs encapsulated in a context that causes clock ticks to happen. Hence we consider programs of the form `repeat f x`, where

```
fun repeat f x = repeat f (f x);
```

We derive a reasoning principle, akin to transfinite induction, for proving POSTd specifications about such calls to `repeat`. This reasoning principle is syntax-directed in the sense that the premises talk only about the behaviour of the function argument `f`. In context, each invocation of `f` is interleaved with a recursive call to `repeat`, which produces the required clock tick.

The `repeat` form allows us to derive a sound and usable reasoning principle, but we do not want the straitjacket of having to write all our code in `repeat` form. Fortunately, there is no need. The key insight is that for every divergent tail-recursive function `f`, there exists a function `g` such that `f` and `repeat g` are semantically equivalent. For example, `yes` can be expressed in `repeat` form as follows:

```
fun yes() = repeat (fn x => put_line "y"; ()) ();
```

We implement and verify a program transformation that given a tail-recursive function produces a function in `repeat` form that satisfies the same POSTd specification. Thus, we can reduce reasoning about arbitrary divergent function calls to calls on `repeat` form, for which we have a sound reasoning principle. The reason we restrict attention to tail-recursive functions is that only functions consisting of tail-calls can truly diverge: any other program will eventually run out of stack space. Tail-calling programs, on the other hand, can run forever without exhausting the stack.

All these concerns are hidden from the user by a custom tactic that evaluates the program transformation in-logic and applies the transfinite induction principle to the resulting `repeat` program. The user may go about her business of verifying divergent programs without ever being exposed to the semantic clock, the `repeat` function, or the program transformation into `repeat` form.

In the remainder of this section, we will describe the induction principle and the program transformation in more detail.

4.1 An induction principle for divergence

Our reasoning principle for programs in `repeat` form is shown in Figure 1. In order to conclude that executing `repeat fv xv` from an initial state satisfying H results in a stream of I/O events satisfying Q using this rule, we must perform an argument by transfinite induction: we must discharge a base case, a successor case and a limit case.

The first conjunct limited_parts ns is a side condition which means, roughly, that ns is the list of all FFI calls that can occur in the program under consideration. Without this restriction, we could make only very limited predictions about the final I/O stream: since separation logic pre- and postconditions are local, we would have to account for the possibility that the frame includes others FFI_parts with other (possibly infinite) event streams that we have no information about.

$$
\begin{array}{l}
\vdash \mathsf{limited\_parts}\ ns \wedge \\
\quad (\exists\, Hs\ events\ vs\ ss\ u. \\
\qquad vs\ 0\ xv \wedge H \Rightarrow Hs\ 0 * \mathsf{one}\ (\mathsf{FFI\_part}\ (ss\ 0)\ u\ ns\ (events\ 0)) \wedge \\
\qquad (\forall\, i\ xv. \\
\qquad\quad vs\ i\ xv \Rightarrow \\
\qquad\quad \{\!|Hs\ i * \mathsf{one}\ (\mathsf{FFI\_part}\ (ss\ i)\ u\ ns\ [])|\!\} \\
\qquad\quad\ fv \cdot [xv] \\
\qquad\quad \{\!|\mathsf{POSTv}\ v'. \\
\qquad\quad\quad \langle vs\ (i+1)\ v'\rangle * Hs\ (i+1) * \\
\qquad\quad\quad \mathsf{one}\ (\mathsf{FFI\_part}\ (ss\ (i+1))\ u\ ns\ (events\ (i+1)))|\!\}) \wedge \\
\qquad Q\ (\mathsf{lflatten}\ (\mathsf{lgenlist}\ (\mathsf{fromList} \circ events)\ \mathsf{None}))) \Rightarrow \\
\{\!|H|\!\}\ \mathsf{repeat} \cdot [fv;\ xv]\ \{\!|\mathsf{POSTd}\ Q|\!\}
\end{array}
$$

Figure 1 Transfinite induction principle for proving POSTd specifications.

The base and successor cases require the user to exhibit a number of streams (represented as functions with domain num), where the i:th element of the streams describe the state after executing the function fv i times. Hs i is a heap predicate that holds after i iterations, $events$ i is the list of I/O produced by the i:th loop iteration, ss i the state of the FFI interface, and vs i a value predicate that $fv^i x$ satisfies.

In the base case, we must show that vs and Hs are true initially; this corresponds to the conjuncts $vs\ 0\ xv$ and $H \Rightarrow Hs\ 0$.

In the successor case, we must discharge a Hoare triple which intuitively states that doing one more iteration of fv respects the streams. Specifically, if we invoke fv with value and heap respectively satisfying vs i and Hs i, and with initial FFI state ss i, then fv terminates with a value and heap satisfying $vs\ (i+1)$ and $Hs\ (i+1)$, producing the I/O events $events\ (i+1)$ and reaching the FFI state ss i. Note that the event list starts out empty: this allows reasoning about each loop iteration that is independent of the I/O history from previous loop iterations.

In the limit case, we need to show that the least upper bound of $events$ – or in other words, the I/O events after infinitely many iterations of fv – satisfies Q. This upper bound has an explicit characterisation, namely the infinite concatenation of $events\ 0$, $events\ 1$, and so on, which is expressed by lflatten (lgenlist ... ...).

The intermediate heaps and values are not used in the limit case: the only relevant aspect is the I/O events. Since Q is a predicate on lazy lists, and since in HOL4 equality on lazy lists coincides with list bisimilarity, discharging this case tends to involve coinductive proofs via list bisimilarity.Hence our technique for verifying diverging programs uses a mix of transfinite induction and coinduction. The historically-minded reader may note that our limit case is similar to the admissibility side condition of Scott induction [34], where the predicate being proved must be closed under supremum.

4.2 Program transformation

To use the induction principles discussed in the previous section to verify a function f, we must first rewrite f into `repeat` form. That is, we must exhibit a function g such that if `repeat` g diverges, f diverges with the same result. In this section, we describe the program transformation we use to produce this g.

```
make_single_app fname allow_fname (e1 ; e2) ≝
 do
  e1' ← make_single_app fname F e1;
  e2' ← make_single_app fname allow_fname e2;
  Some (e1' ; e2')
 od

make_single_app fname allow_fname (f · x) ≝
 if Some f = fname then
  do assert allow_fname; make_single_app fname F x od
 else
  do
   x' ← make_single_app fname F x;
   if allow_fname then Some (then_tyerr (f · x'))
   else Some (f · x')
  od
```

Figure 2 Excerpts from the definition of the repeat program transformation.

We restrict attention to tail-recursive functions f which take a single input argument. The basic idea for how to produce g is simple: the body of g should be the body of f, but with every recursive call $f\ x$ replaced with x. To make the transformation sound, we need to muddy the basic idea with two minor complications. The first is to deal with shadowing carefully: if the function's name is shadowed by `let` bindings, occurrences of the function's name in this scope should obviously not be treated as recursive calls. Second, and more interestingly, what if f terminates? Consider this function:

```
fun condLoop x = if x = 0 then 0 else condLoop (x - 1);
```

If we were to naively rewrite its body as

```
fun condLoop' x = if x = 0 then 0 else x - 1;
```

we would lose soundness: it is easy to see that `repeat condLoop' 0` diverges but `condLoop 0` terminates. To avoid this problem, the transformation makes sure that whenever an expression is encountered in tail position that is *not* a tail call – like the expression `0` in the `if`-branch above – it is replaced with an expression that causes a runtime error.[4] With this modification, evaluation of `repeat condLoop' 0` gets stuck rather than diverges. This preserves soundness, because every Hoare triple is false for a program that gets stuck. It is true that this is not the same behaviour as `condLoop 0`, but that's fine: the transformation is only ever used to prove POSTd specifications, so the two programs only need to agree on divergent behaviour. Thus, to show that `condLoop (~1)` diverges it suffices to show that `repeat condLoop' (~1)` diverges: the `else` branch is always taken, so the runtime error never happens.

While the full definition is too big to show, Figure 2 shows representative extracts from make_single_app, the workhorse of the transformation. Given an expression e corresponding to the body of a function named *fname*, it produces the body of the transformed function.

[4] The expression we use is `ord 0`, which is not type correct because `ord` expects a character as input.

It is written in the option monad because the transformation may fail if e.g. the function is not tail-recursive. To deal with variable capture, *fname* is an option; the idea is that if the name of the function is shadowed, *fname* is None. *allow_fname* is a flag which is T if the expression under consideration is in tail position; this is used to determine whether to inject runtime errors or not. then_tyerr adds an expression which causes runtime errors to another expression.

The main result of this section is that the above is a sound technique for establishing POSTd specifications:

$$\vdash \mathsf{make\_repeat\_closure}\ fv = \mathsf{Some}\ gv \wedge \mathsf{wellformed}\ fv \wedge \\ \{\!|H|\!\}\ gv \cdot x\ \{\!|\mathsf{POSTd}\ Q|\!\} \Rightarrow \\ \{\!|H|\!\}\ fv \cdot x\ \{\!|\mathsf{POSTd}\ Q|\!\}$$

Here make_repeat_closure is the main entry point for the transformation, which lifts make_single_app from function bodies to function closures. It returns a new closure value gv, which is a function of the form `repeat g` for some `g`. A closure value is wellformed if it is not mutually recursive and the function name is distinct from the argument name (this precludes eg. `fun f f = f`).

The proof is tedious and ugly because it is done directly in terms of the CakeML semantics and not in CF. Large parts of it are focused on massaging binding environments and semantic clocks to line up in highly specific ways. To put it another way, the proof consists of exactly the kind of low-level reasoning that we want CF to abstract away from. Doing it here, once and for all, means that when a CF user verifies a diverging program, she won't have to.

We conclude this section by discussing some limitations of our program transformation. Recall that we restrict ourselves to tail-recursion. We do not consider functions with multiple (curried) arguments, nor do we consider mutual recursion. Extensions to handle both should be straightforward, if tedious, to implement; we have not yet done so because for the programs we are interested in verifying, the need has not arisen. One possibility is to add further program transformations on top, encoding curried arguments as tupled arguments and mutual recursion as direct recursion over sum types. It is also worth noting that the proof rule from Section 4.1 is not built into the CF infrastructure, but derived from it. Hence a possible direction for future work is to derive further proof rules covering more exotic forms of recursion, such as recursion through the store.

5 Examples

In this section, we present a number of example program verifications with the intention to showcase various features of our program logic.

Silent loop

Our first example is a function that just calls itself:

```
fun pureLoop x = pureLoop x;
```

This example illustrates that we can reason about loops without I/O, and that the shortest possible divergent program is trivial to verify – the proof script is four lines. The specification we prove is the following:

$$\vdash \mathsf{limited\_parts}\ ns \Rightarrow \\ \{\!|\mathsf{one}\ (\mathsf{FFI\_part}\ s\ u\ ns\ [])|\!\}\ \mathsf{pureLoop} \cdot [xv]\ \{\!|\mathsf{POSTd}\ io.\ io = [\!|\,|\!]\}$$

After applying the tactic described in Section 4, the user must exhibit streams of heap predicates, value predicates and events that describe the state at the nth iteration of the loop body, which in this case is `fn x => x`. We instantiate these variables with the constant functions that return, respectively, emp, $\lambda\, x.$ T and []. The remaining two lines are to prove that `fn x => x` does nothing, and that flattening the infinite list of empty lists is $[\![]\!]$.

Conditional divergence

In this section, we revisit the following example from Section 4.2:

```
fun condLoop x = if x = 0 then 0 else condLoop (x - 1);
```

The point here is to illustrate how to prove specifications about programs that may either terminate or diverge. In this case, the specification is the following:

$$
\begin{array}{l}
\vdash \mathsf{limited\_parts}\ ns \wedge \mathsf{int}\ x\ xv \Rightarrow \\
\quad \{|\mathsf{one}\ (\mathsf{FFI\_part}\ s\ u\ ns\ [])|\} \\
\quad\ \mathsf{condLoop} \cdot [xv] \\
\quad \{|\mathsf{POSTvd} \\
\quad\ (\lambda\, v.\ \langle 0 \leq x \wedge \mathsf{int}\ 0\ v\rangle * \mathsf{one}\ (\mathsf{FFI\_part}\ s\ u\ ns\ [])) \\
\quad\ (\lambda\, io.\ x < 0 \wedge io = [\![]\!])|\}
\end{array}
$$

where POSTvd Q_1 Q_2 abbreviates the disjunction of POSTv Q_1 and POSTd Q_2. Note that the POSTv includes the conditions under which the program terminates ($0 \leq x$), and vice versa for the POSTd part.

The proof proceeds by a case split on whether x is negative. If it is, the POSTvd condition is equivalent to POSTd $io.\ io = [\![]\!]$. From there, the proof is similar to `pureLoop`, with one added step: we must show that the loop maintains the invariant that x is negative.

If x is non-negative, the POSTvd condition is equivalent to the POSTv part. The rest of the proof proceeds by induction on x.

This proof strategy – case splitting on the conditions under which divergence or termination holds – is usually a forced move on the part of the user. An unfortunate side-effect of this is situations where reasoning about the loop body may be duplicated in the POSTv and POSTd cases, but not reusable across them. In practice, this issue is mostly obviated by factoring out code into auxiliary functions, whose specifications will be automatically applied in both the POSTv and POSTd cases. A pragmatic reason for preferring this state of affairs is backwards compatibility: there are already substantial case studies and infrastructure built on top of CF for terminating CakeML programs [12, 17], and since we keep reasoning about divergence separate, there is no need for them to change.

Input and output

In Guéneau et al. [15], CF for CakeML was used to develop and verify an implementation of the Unix `cat` utility; this was later extended to a more efficient implementation on a more realistic file system model [12]. Both developments share a limitation: they use a file system model where the contents of every file and standard stream (eg. `stdin`) can only be finite. Thus the theorems about them are not meaningful in situations with infinite input, such as `cat /dev/zero`, or `yes | cat`, or even just Unix `cat`.[5]

[5] `/dev/zero` is an infinite stream of null characters. Unix `cat` with no arguments will read from `stdin`.

In this section, we will show how to lift this limitation. File system modelling is not the topic of the present paper, so in order to avoid getting lost in file system details, we consider only the case where we read from `stdin` and write to `stdout`. Our example is:

```
fun catLoop (u:unit) = case get_char () of
    None   => ()
  | Some c => (put_char c; catLoop ());
```

In the following Hoare triple, SIO *input events* abbreviates a heap predicate which states that an FFI_part that can read from `stdin` and write to `stdout` is present. Here *input* is a lazy list of characters yet to be read from `stdin`, and *events* is the list of I/O events so far. This allows *input* to be infinite, which lifts the aforementioned limitation of the previous work [15, 12]. Interaction with the standard streams is encapsulated by `get_char` and `put_char`, which are (verified) CakeML library functions that make FFI calls to the corresponding `stdlib` functions, and do the necessary marshalling and unmarshalling.

The function cat abbreviates the I/O we expect to see for a character stream ll:

$$\mathsf{cat}\ ll \stackrel{\mathsf{def}}{=} \mathsf{lflatten}\ (\mathsf{lmap}\ (\lambda\, c.\ [\![\mathsf{get\_char\_event}\ c;\ \mathsf{put\_char\_event}\ c]\!])\ ll)$$

The Hoare triple which specifies the whole function is this:

$$\begin{array}{l}
\vdash \mathsf{limited_parts\ names} \Rightarrow \\
\quad \{|\mathsf{SIO}\ \mathit{input}\ []|\} \\
\quad\ \mathsf{catLoop} \cdot [\mathit{uv}] \\
\quad \{|\mathsf{POSTvd} \\
\quad\ (\lambda\, v. \\
\qquad \langle \mathsf{finite}\ \mathit{input} \wedge \mathsf{unit_type}\ ()\ v\rangle * \\
\qquad \mathsf{SIO}\ [\![]\!]\ (\mathsf{snoc\ get_char_eof_event}\ (\mathsf{the}\ (\mathsf{toList}\ (\mathsf{cat}\ \mathit{input}))))) \\
\quad\ (\lambda\, \mathit{io}.\ \neg\mathsf{finite}\ \mathit{input} \wedge \mathit{io} = \mathsf{cat}\ \mathit{input})|\}
\end{array}$$

As with the `condLoop` example, the postcondition is in POSTvd form. It will either terminate or diverge, depending on whether *input* is finite or not. If it is finite, we return unit, consume all pending inputs from SIO, and produce the expected sequence of I/O events, with a final EOF event corresponding to the failed `get_char` when *input* is empty. If *input* is infinite, the I/O events are cat *input*. The whole proof is around 90 lines of HOL script.

Traversing cyclic pointer structures

We now consider an example that combines divergence with separation logic-style reasoning about the shape of memory. Here we will traverse a cycle of cons cells containing characters on the heap, and print each character we encounter. The code is as follows:

```
fun pointerLoop c =
  case !c of (a,b) =>
    (put_char a; pointerLoop b);
```

As an aside, the reader may notice that, in standard Hindley–Milner type systems, this program has no type: it requires `c` to have a type `'a` such that `'a = (char * 'a) ref`. That's fine since CakeML's raw evaluation semantics is untyped, and so the only purpose of the type system is to establish the absence of a certain class of runtime errors. Here, we establish this absence by proving Hoare triples instead.

We use the heap predicate ref_list to describe pointer cycles:

$$\mathsf{ref\_list}\ rv\ []\ A\ [] \stackrel{\mathsf{def}}{=} \exists\ loc.\ \langle rv = \mathsf{Loc}\ loc\rangle$$

$$\mathsf{ref\_list}\ rv\ (rv_2{::}rvs)\ A\ (x{::}l) \stackrel{\mathsf{def}}{=}$$
$$\exists\ loc\ v_1.\ \langle rv = \mathsf{Loc}\ loc\rangle * loc \mapsto (v_1,\ rv_2) * \langle A\ x\ v_1\rangle * \mathsf{ref\_list}\ rv_2\ rvs\ A\ l$$

The idea is that ref_list rv rvs A xs describes an encoding of a list segment with elements xs of type A, where rv is a pointer to the memory location where this encoding resides, and rvs are pointers to the encodings of the tails. Note that there is no indication on the heap of where the segment ends; rather, the last pointer of rvs is left dangling. A cyclic lazy list, whose elements are those of xs over and over, is represented by a heap predicate ref_list rv (snoc rv rvs) A xs where the last pointer points back to the beginning. This predicate allows a concise specification of `pointerLoop`:

$$\begin{array}{l}
\vdash \mathsf{limited_parts\ names} \Rightarrow \\
\quad \{|\mathsf{SIO}\ [|]\ []\ *\ \mathsf{ref_list}\ rv\ (\mathsf{snoc}\ rv\ rvs)\ \mathsf{char}\ l|\} \\
\quad\ \mathsf{pointerLoop} \cdot [rv] \\
\quad \{|\mathsf{POSTd}\ io.\ io = \mathsf{lmap\ put_char_event}\ (\mathsf{lrepeat}\ l)|\}
\end{array}$$

In the successor case, the proof uses the fact that the ref_list predicate satisfies a kind of rotational symmetry – intuitively, any tail of a cyclic list with cycle xs is a cyclic list whose cycle is a rotation of xs. In the limit case, we use bisimulation up-to context [33] to reduce the size of the candidate relation to one pair only.

Verifying repeat with repeat

We now turn to a question of meta-verification: can the `repeat` construct described in Section 4.2 be used to verify `repeat` itself? For trivial syntactic reasons, the immediate answer is no: `repeat` is curried, and the transformation only considers one-argument functions. However, the answer changes if we allow ourselves to consider an uncurried version:

```
fun myRepeat (f,r) = myRepeat(f,f(r))
```

For such a function, we can easily (in just 8 lines) prove the following specification.

$$\begin{array}{l}
\vdash \mathsf{limited_parts}\ ns \Rightarrow \\
\quad \{|H\ * \\
\quad \langle vs\ 0\ xv \wedge H \Rightarrow Hs\ 0\ *\ \mathsf{one}\ (\mathsf{FFI_part}\ (ss\ 0)\ u\ ns\ (events\ 0)) \wedge \\
\quad (\forall\ i\ xv. \\
\qquad vs\ i\ xv \Rightarrow \\
\qquad \{|Hs\ i\ *\ \mathsf{one}\ (\mathsf{FFI_part}\ (ss\ i)\ u\ ns\ [])|\} \\
\qquad\ fv \cdot [xv] \\
\qquad \{|\mathsf{POSTv}\ v'. \\
\qquad\quad \langle vs\ (i+1)\ v'\rangle * Hs\ (i+1)\ * \\
\qquad\quad \mathsf{one}\ (\mathsf{FFI_part}\ (ss\ (i+1))\ u\ ns\ (events\ (i+1)))|\}) \wedge \\
\quad Q\ (\mathsf{lflatten}\ (\mathsf{lgenlist}\ (\mathsf{fromList} \circ events)\ \mathsf{None}))\rangle|\} \\
\quad\ \mathsf{myRepeat} \cdot [(fv,\ xv)] \\
\quad \{|\mathsf{POSTd}\ io.\ Q\ io|\}
\end{array}$$

Note that the preconditions of the Hoare triple above are essentially the same as the assumptions of the induction principle from Figure 1. In other words, we have given `repeat` a CF specification by applying the `repeat` transformation to `repeat` itself (modulo currying).

Listing 1 Excerpts from the filter source code.

```
fun forward_loop inputarr =
    (#(accept_call) "" inputarr;
     let val ln = Word8Array.substring inputarr 0 256;
         val ln' = cut_at_null ln;
     in
       if match_string ln' then
         #(emit_string) ln' dummyarr
       else ()
     end;
     forward_loop inputarr);

fun forward_matching_lines u =
    let val inputarr = Word8Array.array 256 (Word8.fromInt 0);
    in
      forward_loop inputarr
    end
```

6 Case study: verified filter components

In this section we describe the application of the techniques developed in this paper to a case study: the development of verified architectural components for systems built on the formally verified seL4 microkernel [21]. The particular domain we consider is unmanned aerial vehicles (UAVs), but the techniques can be applied to other systems too. The case study itself is the topic of another paper [35].

The particular component we consider is a filter. Architecturally, the filter sits between a radio driver, which receives commands from a ground station, and the rest of the UAV's flight control subsystem. Its purpose is (a) to protect the rest of the flight control subsystem from cyber-attacks based on malformed command messages, and to achieve this in a way that (b) does not require changing legacy components, (c) does not increase the attack surface of the overall system, and (d) does not prevent the rest of the system from fulfilling its mission.

Note that while (a) is a safety property, (d) is a liveness property: it requires that beyond rejecting malformed messages, the filter must never cause a well-formed message to be dropped. The precise definition of "well-formed" will of course vary; here we are interested in properties that can be decided by checking membership in a regular language $\mathcal{L}$.

An excerpt of the filter implementation is shown in Listing 1. Here `match_string` is a CakeML function that decides membership in $\mathcal{L}$. The function `forward_loop` will repeatedly invoke (via FFI) `accept_call`, which will receive a message from the radio driver via remote procedure call and write it to the buffer `inputarr`. If the contents of `inputarr` up until the first null terminator satisfies $\mathcal{L}$, we forward it to the flight controller, again via FFI (`emit_string`). The FFI calls are connected to seL4's RPC mechanism.

The theorem that states the desired liveness property is the following. In words, if *input* is an infinite stream of null-terminated strings of at most 256 characters[6], then `forward_matching_lines` will not terminate or abort (POSTd) and the messages it sends are precisely the inputs filtered by the language $\mathcal{L}$.

[6] The requirements on null-termination and message length show up as assumptions in this proof, but in practice they do not constitute attack vectors because they are enforced by our communication backend, namely the CAmkES component platform for seL4 [22].

```
⊢ llimited_parts ["accept_call"; "emit_string"] ∧ llength input = None ∧
  every null_terminated_w input ∧ every ((≥) 256 ∘ length) input ⇒
  {|seL4_IO input [] ∗ w8array dummyarr_loc []|}
   forward_matching_lines · [rv]
  {|POSTd io.
     lfilter is_emit io =
     lmap (output_event_of ∘ cut_at_null_w) (lfilter (ℒ ∘ cut_at_null_w) input)|}
```

Prior to this paper, the same liveness property was proved by Slind et al. [35] for the same program; the painful nature of those proofs was part of our motivation for extending CF with divergence. Having no verification framework at hand with support for divergence, the proofs were done directly in terms of the operational semantics (see Section 3.2). The result is proofs that spend inordinate amounts of energy massaging clocks and environments while carefully stepping through the interpretation of the program, e.g., unfold the definition of evaluate 11 times, then unfold some auxiliary definitions to find a particular value in the binding environment, then case split on whether we ran out of clock or not, then unfold evaluate 5 times, et cetera ad nauseam.

Redoing these proofs in CF, the results are more pleasant. At no point do clocks or binding environments enter into the proofs: instead, the granularity of proof steps is about the granularity of statements in the source program, with intermediate verification conditions generated at each step. Moreover, before deriving the equation about outputs in the POSTd condition above, Slind et al. [35] expend significant energy proving an explicit characterisation for the supremum of the I/O events. By using the induction principle from Figure 1 we get an explicit characterisation for free, so this effort is no longer necessary.

Besides the higher abstraction level, the proofs are shorter: the CF version of the theory that performs filter synthesis and verification comprises 1479 lines of HOL4, while the non-CF version is 1971 lines long. The former line count also includes infrastructure for lifting the filter's FFI model to CF's FFI abstraction.

In the CF version, we also derive a theorem from the specification above that gives the same liveness property directly in terms of the operational semantics, with no reference to CF abstractions such as heaps, FFI parts or Hoare triples:

```
⊢ llength input = None ∧ every null_terminated_w input ∧ every ((≥) 256 ∘ length) input ⇒
  ∃ events.
    semantics_prog … … [val () = forward_matching_lines ()] (Diverge events) ∧
    lfilter is_emit events =
    lmap (output_event_of ∘ cut_at_null_w) (lfilter (ℒ ∘ cut_at_null_w) input)
```

Here the elided arguments to semantics_prog are the program's initial state and environment.

The fact that we can prove theorems such as the one above means that our use of CF does not increase the trusted computing base. More importantly, it means our POSTd specification can be fed through CakeML's compiler correctness theorem [36] to obtain corresponding liveness theorems about the resulting binary (with the current caveat that the compiler correctness theorem allows the binary to exit early with an out-of-memory error, see Section 8).

7 Related work

The historical roots of characteristic formulae go back to the modal logic characterisation of bisimilarity by Hennessy and Milner [16]. Charguéraud's CFML work [5, 6] builds on this idea to develop a verification framework for impure functional programs. The CakeML CF framework [15] adapts these ideas for CakeML, and adds a mechanised soundness proof as well as support for exceptions and I/O [12]. Characteristic formulae have also been used to reason about complexity [9, 14], higher-order representation predicates [8], and read-only permissions [10].

Transfinite models have been used in program analysis in areas such as term rewriting [20] and program slicing [13]. In these cases program flow is modelled to continue after infinite loops, for the purpose of investigating how the loop affects succeeding computations. In our setting we are not interested in considering computations beyond infinite loops. As a result, we only need to consider the smallest infinite ordinal ω in our transfinite induction.

There is a large body of work on non-termination; most relevant to us are works that consider Hoare-like logics [18, 19, 11, 23, 24], coinduction [26, 1, 7, 4, 32], and interactive theorem proving [26]. We will focus the discussion on work that also treat I/O behaviour.

Nakata and Uustalu [27] introduce coinductive big-step semantics for a simple While language, formalised in Coq. They use coinductively defined state traces to reason uniformly about both termination and non-termination. In a follow-up paper [28] they define a Hoare logic for their big-step semantics, where postconditions describe state traces rather than a single final state. In another paper [29], they extend their semantics to handle I/O using resumptions. Resumptions can be thought of as coinductive trees that describe the I/O behaviour of all possible runs of a program. They do not extend their Hoare logic to this resumption semantics. In contrast, CakeML gives semantics to divergent programs not by coinduction, but by taking the limit of a clocked inductive semantics. An advantage of Nakata and Uustalu's approach is that it treats termination and non-termination uniformly, while we need to treat the two cases separately. On the other hand, this necessitates the introduction of silent actions (that do not correspond to I/O) into their traces, so that termination and silent divergence can be distinguished. The presence of silent actions lead, in turn, to observationally equivalent programs potentially exhibiting different traces. To recover observational equivalence, they can either consider traces up to termination-sensitive weak bisimilarity on the meta-level, or use one of two alternative semantics – one constructive and one classical – that do not produce silent actions. However, the constructive semantics fails to account for silent divergence, and the classical version does not treat termination and divergence uniformly. A more practical difference is that our Hoare logic considers I/O behaviour, and that CakeML is a much richer language than While.

Penninckx et al. [31] define a program logic for reasoning about I/O, where I/O events occur in the preconditions rather than the postconditions. These can be thought of as permission to do these events. Their assertion language is a separation logic where the heaps are Petri nets: the transitions are I/O events, and the nodes are analogous to our FFI states. For terminating programs, a Hoare triple can express that the right I/O events were performed in the right order by specifying which nodes have tokens in the postcondition. For a non-terminating program, the preconditions express an upper bound on the I/O events, but unlike our work, not necessarily a *least* upper bound. Hence they can prove safety but not liveness for non-terminating programs.

Ancona et al. [2, 3] have recently explored using corules and coaxioms – intuitively, auxiliary rules used to filter out judgements with undesired conclusions from infinite proof trees – to give semantics in terms of I/O traces for divergent executions in a lambda calculus

and a small Java-like language. The authors focus on semantics and do not develop a program logic, but they present an example verification similar to our `cat` example, albeit directly in terms of the operational semantics and with a more abstract treatment of I/O. Their work is not formalised in a proof assistant.

8 Conclusion

We have seen how characteristic formulae for CakeML, an existing verification framework for total correctness of impure terminating programs with I/O, can be extended to support liveness of non-terminating programs. The extension is non-invasive: existing proofs about terminating programs need not change at all. We support syntax-directed reasoning about loops, that reduces proofs about loops to proofs about the loop body. We support silent divergence without the need to involve clocks or special silent actions.

The framework is proven sound with respect to the CakeML semantics and thus integrated into the wider CakeML ecosystem, including in particular a verified optimising compiler [36]. Thus we can verify real programs, and reify our specifications to the machine code that runs them. Currently this comes with a caveat: liveness properties carry over to the binary only under the assumption that we do not run out of memory. The missing puzzle piece for unconditional liveness at the binary level is a means to discharge this assumption, which we are working towards by developing a verified space-cost semantics.

References

1. Davide Ancona. Soundness of Object-Oriented Languages with Coinductive Big-Step Semantics. In James Noble, editor, *Object-Oriented Programming (ECOOP)*. Springer, 2012. doi:10.1007/978-3-642-31057-7_21.
2. Davide Ancona, Francesco Dagnino, and Elena Zucca. Reasoning on divergent computations with coaxioms. *PACMPL*, 1(OOPSLA), 2017. doi:10.1145/3133905.
3. Davide Ancona, Francesco Dagnino, and Elena Zucca. Modeling Infinite Behaviour by Corules. In *Object-Oriented Programming (ECOOP)*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPIcs.ECOOP.2018.21.
4. Richard Bubel, Crystal Chang Din, Reiner Hähnle, and Keiko Nakata. A Dynamic Logic with Traces and Coinduction. In *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*. Springer, 2015. doi:10.1007/978-3-319-24312-2_21.
5. Arthur Charguéraud. Program verification through characteristic formulae. In Paul Hudak and Stephanie Weirich, editors, *International Conference on Functional Programming (ICFP)*. ACM, 2010.
6. Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In *International Conference on Functional Programming (ICFP)*. ACM, 2011. doi:10.1145/2034773.2034828.
7. Arthur Charguéraud. Pretty-Big-Step Semantics. In *European Symposium on Programming (ESOP)*. Springer, 2013. doi:10.1007/978-3-642-37036-6_3.
8. Arthur Charguéraud. Higher-order representation predicates in separation logic. In *Certified Programs and Proofs (CPP)*, 2016. doi:10.1145/2854065.2854068.
9. Arthur Charguéraud and François Pottier. Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation. In *Interactive Theorem Proving (ITP)*, 2015. doi:10.1007/978-3-319-22102-1_9.
10. Arthur Charguéraud and François Pottier. Temporary Read-Only Permissions for Separation Logic. In *European Symposium on Programming (ESOP)*. Springer, 2017. doi:10.1007/978-3-662-54434-1_10.

11 Hong Yi Chen, Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter W. O'Hearn. Proving Nontermination via Safety. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2014. doi:10.1007/978-3-642-54862-8_11.

12 Hugo Férée, Johannes Åman Pohjola, Ramana Kumar, Scott Owens, Magnus O. Myreen, and Son Ho. Program Verification in the Presence of I/O - Semantics, Verified Library Routines, and Verified Applications. In *Verified Software: Theories, Tools, Experiments (VSTTE)*, 2018. doi:10.1007/978-3-030-03592-1_6.

13 Roberto Giacobazzi and Isabella Mastroeni. Non-Standard Semantics for Program Slicing. *Higher-Order and Symbolic Computation*, 16(4), 2003. doi:10.1023/A:1025872819613.

14 Armaël Guéneau, Arthur Charguéraud, and François Pottier. A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification. In *European Symposium on Programming (ESOP)*. Springer, 2018. doi:10.1007/978-3-319-89884-1_19.

15 Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. Verified Characteristic Formulae for CakeML. In Hongseok Yang, editor, *European Symposium on Programming (ESOP)*, volume 10201 of *LNCS*. Springer, 2017. doi:10.1007/978-3-662-54434-1_22.

16 Matthew Hennessy and Robin Milner. On Observing Nondeterminism and Concurrency. In J. W. de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming (ICALP)*, LNCS. Springer, 1980. doi:10.1007/3-540-10003-2_79.

17 Son Ho, Oskar Abrahamsson, Ramana Kumar, Magnus O. Myreen, Yong Kiam Tan, and Michael Norrish. Proof-Producing Synthesis of CakeML with I/O and Local State from Monadic HOL Functions. In *Automated Reasoning – International Joint Conference (IJCAR)*. Springer, 2018. doi:10.1007/978-3-319-94205-6_42.

18 Marieke Huisman and Bart Jacobs. Java Program Verification via a Hoare Logic with Abrupt Termination. In *Fundamental Approaches to Software Engineering (FASE)*, 2000. doi:10.1007/3-540-46428-X_20.

19 Bart Jacobs and Erik Poll. A Logic for the Java Modeling Language JML. In *Fundamental Approaches to Software Engineering (FASE)*, 2001. doi:10.1007/3-540-45314-8_21.

20 Richard Kennaway, Jan Willem Klop, M. Ronan Sleep, and Fer-Jan de Vries. Transfinite Reductions in Orthogonal Term Rewriting Systems. *Inf. Comput.*, 119(1), 1995. doi:10.1006/inco.1995.1075.

21 Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6), 2010. doi:10.1145/1743546.1743574.

22 Ihor Kuz, Yan Liu, Ian Gorton, and Gernot Heiser. CAmkES: A component model for secure microkernel-based embedded systems. *Journal of Systems and Software*, 80(5), 2007. doi:10.1016/j.jss.2006.08.039.

23 Ton Chanh Le, Cristian Gherghina, Aquinas Hobor, and Wei-Ngan Chin. A Resource-Based Logic for Termination and Non-termination Proofs. In *International Conference on Formal Engineering Methods (ICFEM)*, 2014. doi:10.1007/978-3-319-11737-9_18.

24 Ton Chanh Le, Shengchao Qin, and Wei-Ngan Chin. Termination and non-termination specification inference. In *Programming Language Design and Implementation (PLDI)*. ACM, 2015. doi:10.1145/2737924.2737993.

25 Xavier Leroy. A Formally Verified Compiler Back-end. *J. Autom. Reasoning*, 43(4), 2009. doi:10.1007/s10817-009-9155-4.

26 Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2), 2009. doi:10.1016/j.ic.2007.12.004.

27 Keiko Nakata and Tarmo Uustalu. Trace-Based Coinductive Operational Semantics for While. In *Theorem Proving in Higher Order Logics (TPHOLs)*. Springer, 2009. doi:10.1007/978-3-642-03359-9_26.

28 Keiko Nakata and Tarmo Uustalu. A Hoare Logic for the Coinductive Trace-Based Big-Step Semantics of While. In *European Symposium on Programming (ESOP)*. Springer, 2010. doi:10.1007/978-3-642-11957-6_26.

29 Keiko Nakata and Tarmo Uustalu. Resumptions, Weak Bisimilarity and Big-Step Semantics for While with Interactive I/O: An Exercise in Mixed Induction-Coinduction. In *Structural Operational Semantics (SOS)*, 2010. doi:10.4204/EPTCS.32.5.

30 Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional Big-Step Semantics. In Peter Thiemann, editor, *European Symposium on Programming (ESOP)*, LNCS. Springer, 2016. doi:10.1007/978-3-662-49498-1_23.

31 Willem Penninckx, Bart Jacobs, and Frank Piessens. Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs. In *European Symposium on Programming (ESOP)*. Springer, 2015. doi:10.1007/978-3-662-46669-8_7.

32 Casper Bach Poulsen and Peter D. Mosses. Flag-based big-step semantics. *J. Log. Algebr. Meth. Program.*, 88, 2017. doi:10.1016/j.jlamp.2016.05.001.

33 Davide Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8(5), October 1998. doi:10.1017/S0960129598002527.

34 Dana Scott and J.W. De Bakker. A Theory of Programs. Unpublished manuscript, IBM Vienna, 1969.

35 Konrad Slind, David S. Hardin, Johannes Åman Pohjola, and Michael Sproul. Synthesis of Verified Architectural Components for Autonomy Hosted on a Verified Microkernel. Draft, 2019.

36 Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29, 2019. doi:10.1017/S0956796818000229.

37 D. A. Turner. Total Functional Programming. *J. UCS*, 10(7), 2004. doi:10.3217/jucs-010-07-0751.

The DPRM Theorem in Isabelle

Jonas Bayer
Freie Universität Berlin, Arnimallee 2, 14195 Berlin, Germany
jonas.bayer@fu-berlin.de

Marco David
Jacobs University Bremen, Campus Ring 1, 28759 Bremen, Germany
m.david@jacobs-university.de

Abhik Pal
Jacobs University Bremen, Campus Ring 1, 28759 Bremen, Germany
ab.pal@jacobs-university.de

Benedikt Stock
Jacobs University Bremen, Campus Ring 1, 28759 Bremen, Germany
b.stock@jacobs-university.de

Dierk Schleicher
Technische Universität Berlin, Germany
dierk.schleicher@gmx.de

Abstract

Hilbert's 10th problem asks for an algorithm to tell whether or not a given diophantine equation has a solution over the integers. The non-existence of such an algorithm was shown in 1970 by Yuri Matiyasevich. The key step is known as the DPRM theorem: every recursively enumerable set of natural numbers is Diophantine. We present the formalization of Matiyasevich's proof of the DPRM theorem in Isabelle. To represent recursively enumerable sets in equations, we implement and arithmetize register machines. Using several number-theoretic lemmas, we prove that exponentiation has a diophantine representation. Further, we contribute a small library of number-theoretic implementations of binary digit-wise relations. Finally, we discuss and contribute an `is_diophantine` predicate. We expect the complete formalization of the DPRM theorem in the near future; at present it is complete except for a minor gap in the arithmetization proofs of register machines and extending the `is_diophantine` predicate by two binary digit-wise relations.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Theory of computation → Higher order logic

Keywords and phrases DPRM theorem, Hilbert's tenth problem, Diophantine predicates, Register machines, Recursively enumerable sets, Isabelle, Formal verification

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.33

Category Short Paper

Supplement Material Isabelle formalisation: `https://gitlab.com/hilbert-10/dprm`

Acknowledgements We want to thank everyone who has contributed to this project: Deepak Aryal, Bogdan Ciurezu, Yiping Deng, Prabhat Devkota, Simon Dubischar, Malte Haßler, Yufei Liu and Maria Antonia Oprea. Without their involvement, most notably Yufei Liu's implementation of `is_diophantine`, the project would not stand where it is today. Moreover, we would like to express our sincere gratitude to the entire welcoming and supportive Isabelle community. In particular, we are indebted to Mathias Fleury for all his help with Isabelle. Finally, a big thank you Yuri Matiyasevich for inspiring and initiating the project as well as to our supervisor Dierk Schleicher, for motivating us throughout the project, connecting us to many experts in the field, and all his critical feedback.

1 Introduction

The mathematician David Hilbert is well known for the axiomatic method and his *Hilbert program* on a quest to formalize mathematics. While the dawn of the twentieth century did not witness any computers, let alone interactive theorem provers, Hilbert did write a list of 23 problems to direct the international mathematical community. In the tenth problem, he asked for an algorithm to decide any diophantine equation. After the presentation of the problem in 1900, a negative solution was conjectured by Martin Davis in 1950. Yet, the proof that there is no such algorithm was only completed in 1970 by Yuri Matiyasevich, resulting in the Davis-Putnam-Robinson-Matiyasevich theorem.

In an ongoing effort, we formalize the negative solution to Hilbert's tenth problem and in first instance the proof of the DPRM theorem in Isabelle. This paper presents the formalization of Matiyasevich's proof [4] of the DPRM theorem. A core result is a representation of exponentiation in terms of Diophantine equations, obtained from a generalized Fibonacci sequence. Additionally, we implement and arithmetize register machines, Minsky machines in our case. The simulation of their execution in equations allows us to express a recursively enumerably set in equations. Finally, an obvious prerequisite for the formalization is an `is_diophantine` predicate, which we implement and apply to e.g. the exponential relation.

For our formalization of the DPRM theorem, we discuss three conceptual ingredients in Sections 2, 3 and 4: diophantine predicates, the fact that exponentiation is Diophantine and the arithmetization of Minsky machines. They culminate in the DPRM theorem in Section 5 before we provide the overall conclusion in Section 6.

2 Diophantine Predicates

A diophantine polynomial is constructed from addition and multiplication of integer constants as well as natural variables and parameters. Diophantine relations and sets are then defined as follows.

▶ **Definition 1.** *An n-ary predicate $\mathcal{P}$ is called* diophantine *if there exists a diophantine polynomial D, such that a tuple of parameters $\mathbf{a} = (a_1, \ldots, a_n) \in \mathbb{N}^n$ satisfies $\mathcal{P}$ if and only if there exist variables $\mathbf{x} = (x_1, \ldots, x_m) \in \mathbb{N}^m$ such that $D(\mathbf{a}, \mathbf{x}) = 0$.*

▶ **Definition 2.** *A set $\mathcal{A} \subset \mathbb{N}^n$ of n-tuples is called* diophantine *if there exists a diophantine predicate $\mathcal{P}$ such that $\mathbf{a} \in \mathcal{A} \iff \mathcal{P}(\mathbf{a})$.*

Examples of diophantine predicates are $\mathcal{P}(a, b) \equiv a \leq b$ or $\mathcal{P}(a, b) \equiv a \mid b$, represented respectively as $\exists x.\, D(a, b, x) = a - b + x = 0$ or $\exists x.\, D(a, b, x) = ax - b = 0$. The sets of all tuples (a, b) satisfying these relations are examples, respectively, of diophantine sets. A third, more surprising, example is that the set of all primes is diophantine, for which a simple diophantine polynomial in 26 variables can be found [4, Section 1.4.1].

It is an elementary fact that conjunctions and disjunctions of diophantine predicates are diophantine. Rather non-trivial is the fact that exponentiation of natural numbers is diophantine as well: for a long time, the opposite was conjectured, and the negative solution to Hilbert's 10th Problem was established once it was established that exponentiation is diophantine. The proof of this assertion constitutes one of the core steps of the proof of DPRM and is presented in the following section. Using exponentiation, we may also access a much more general class of relations and prove that they are diophantine. For a diophantine representation of binomial coefficients in terms of exponential, diophantine equations, we formalize Lucas's theorem. Similarly, one finds that the binary digit-wise

relations orthogonality ($a \perp b := \forall k.\, a_k b_k = 0$) and masking ($a \preceq b := \forall k.\, a_k \leq b_k$) are diophantine. From there, digit-wise multiplication (&&, i.e. binary AND) can be expressed as a diophantine relation, too:

$$a \mathbin{\&\&} b = c \iff c \preceq a \wedge c \preceq b \wedge (a - c) \perp (b - c)$$

One might expect that the above relations should be handled easily on a low level, but this was not the case. In fact, they constituted a significant amount of the formalization efforts and required development of new number-theoretic library of utilities to handle natural numbers digit-wise. To demonstrate this, note that most lemmas in this part of the formalization rely on new functions to access the n-th digit in binary or base b representation of a natural number, which did not exist before (in Isabelle).

The implementation of `is_diophantine`. In principle, diophantine predicates and polynomials are straightforwardly implemented and equipped with an `eval` function. However, there are many possibilities to model the details of representing variables and parameters, which show different usability in formal proofs. Due to the non-existence of dependent types in Isabelle, n-tuples of parameters and variables can be implemented either as maps `nat` ⇒ `nat` from indices to values (which are eventually zero) or as finite lists. Additionally, one may choose to treat variables and parameters separately, or consider them part of the same map or list. Each of these implementations has its (dis)advantages, and different progress has been made towards formalizing necessary relations using these predicates.

A difficulty common to all approaches is the exchange and relabelling of variables and parameters. This is necessary when proving that a relation which is expressed as a compound diophantine expression (i.e. as conjunctions and disjunctions of smaller diophantine expressions) is diophantine, too – a fact which is usually mentioned only on the side in paper proofs. The most successful approach so far uses an implementation of `is_diophantine` using one `nat` ⇒ `nat` map for parameters and variables alike.

However, as the currently developed theory requires much manual work in its proofs, we are developing and experimenting with alternative implementations in parallel in order to bridge the open gaps. In particular, we still need to prove that the aforementioned binary relations, which are used later in the diophantine representation of register machines, are indeed diophantine.

3 Exponentiation is Diophantine

Exponential relations of the type $p = q^r$ and in particular their diophantine representations are the key bridge to connect the notions of *recursively enumerable* and *diophantine* since exponentiation arises in the arithmetization of register machines, as described in the next section.

Following Matiyasevich [4, Section 3], we define a second-order recurrence $\alpha_b(n)$ similar to the Fibonacci numbers to later obtain an expression of q^r in terms of this sequence. In intermediate steps, the proof uses 2×2 matrices to obtain a first-order sequence as well as a diophantine, closed form for the $\alpha_b(n)$. Unless explicitly stated otherwise, all variables are natural numbers and may take values from $\mathbb{N} = \{0, 1, 2, \ldots\}$.

▶ **Definition 3.** *Let $\alpha_b(n)$ denote the unique second-order recurrence of natural numbers parameterized by $b \geq 2$, that satisfies $\alpha_b(0) = 0$ and $\alpha_b(1) = 1$, and for all $n \in \mathbb{N}$*

$$\alpha_b(n + 2) = b\alpha_b(n + 1) - \alpha_b(n)$$

To follow the rough argument, first note that $\alpha_2(n) = n$ is linear but $b > 2$ implies that $(b-1)^n \leq \alpha_b(n+1) \leq b^n$, i.e. $\alpha_b(n)$ grows exponentially. Then, using various divisibility and congruence relations[1] of the $\alpha_b(n)$, we obtain a diophantine system of 15 equations in 8 variables (in addition to the three parameters a, b, c) for the relation $a = \alpha_b(c)$ given $b > 3$. Combining these two results, with $m = bq - q^2 - 1$, we can then express

$$q^r \equiv q\alpha_b(r) - \alpha_b(r-1) \pmod{m}$$

which is a diophantine representation of exponentiation for $q > 0$ as intended, using the fact that congruence relations have a diophantine representation. Adding the case $q = 0$ and expanding, we obtain the following.

▶ **Theorem 4.** *The predicate $\mathcal{P}_{\exp}(p,q,r) \equiv p = q^r$ has a diophantine representation which is implicitly given by the equivalence to a boolean combination of simpler diophantine relations.*

$$\begin{aligned}\mathcal{P}_{exp}(p,q,r) \iff\ & (q = 0 \wedge r = 0 \wedge p = 1) \vee \\ & (q = 0 \wedge r < 0 \wedge p = 0) \vee \\ & \exists b, m.\ (b = \alpha_{q+4}(r+1) + q^2 + 2\ \wedge\ m = bq - q^2 - 1 \\ & \qquad p < m\ \wedge\ p \equiv q\alpha_b(r) - b\alpha_b(r) + \alpha_b(r+1) \bmod m).\end{aligned}$$

In our contribution, this theorem is fully formalized in rather verbose 2800 loc using 86 intermediate lemmas. Using the currently most successful `is_diophantine` predicate explained above, we then show `is_diophantine` $\mathcal{P}_{\exp}$ in additional 270 loc.

4 Arithmetization of Minsky machines

A core concept in our work are Minsky machines, a type of register machine. We implement them in Isabelle and formally prove their arithmetization, i.e. simulation through equations. Although known to be equivalent to Turing machines, their simpler mode of operations and simpler instructions makes this process easier than for a Turing machine. Every register machine has a finite number of registers R_l and states S_k and is able to execute three types of instructions:

i) S_k: `INC` R_l; S_i
ii) S_k: `DEC` R_l; S_i; S_j
iii) S_k: `HALT`

Each register stores a natural number. In state k, the k-th instruction from the *program* p, i.e. list of instructions, is executed. Instructions of type i) increment some register R_l and move to another state S_i; instructions of type ii) decrement a register R_l and move to some state S_i if the register value was larger than zero, else move to another state S_j; and instructions of type iii) halt execution. In analogy to Turing machines, call the list of register values the *tape* $\mathcal{T}$. A tuple $(k, \mathcal{T})$ is then called a *configuration* of the register machines at some time step t.

At every time step, the current instruction is fetched from the program and the tape is updated accordingly. This way, the next configuration is obtained, until the halt state is reached. With the existing implementation of Turing and Abacus machines by Xu et al. [6]

[1] Matiyasevich notes that these "required properties of numbers $\alpha_b(n)$ can be proved by induction, however, many of them can be made more visual by using matrices." We follow his proof using matrices as given, however an alternative, possibly more direct approach using induction seems feasible, too.

at hand, we modeled our register machines in a fetch-update-step cycle similar to their approach. In addition to being as modular as possible, this hopefully allows more easily for future consolidation of both implementations.

Now, the goal is to find equations which simulate the execution of a register machine. The arithmetization as done by Matiyasevich [4] obtains a set of equations with parameter a which are satisfied if and only if the register machine terminates upon being given a as input in the first register in the initial configuration. In this regard, define the number $r_{l,t}$ to be the value of register l at time t. Similarly, define $s_{k,t}$ to be 1 if the machine is in state k at time t and 0 otherwise. In order to model whether a register has value 0 or not, needed for all decrement states, define *zero indicators* $z_{l,t}$ which are 0 if $r_{l,t} = 0$ and 1 otherwise.

It is straightforward to construct equations for all l, k, t relating all the above numbers to sufficiently and necessarily guard that the program is properly executed and that the machine will halt after a finite number of steps q. However, depending on the input a, q may vary. Should the set of all valid inputs be unbounded, any finite set of equations may not be enough to guarantee termination for all valid inputs. Hence, explicit time-dependence needs to be removed from the equations. This is done by representing the time-evolution of any value in a single natural number, encoded with a sufficiently large basis b, chosen as a power of 2. For example, we accumulate all values of the register l in the number $r_l = \sum_{t=0}^{q} r_{l,t} b^t$. With z_l defined accordingly, the simple inequality $\forall t.\, z_{l,t} \leq 1$ is then encoded as the masking relation $z_l \preceq \sum_{t=0}^{q} 1 \cdot b^t$.

After removal of all explicit time-dependence, only 15 equations remain. We have successfully formalized that these are necessary for an initially valid[2] register machine to terminate in finite time (2400 loc). The much simpler converse statement, the sufficiency of these equations, is almost completely formalized (currently 1100 loc). For its completion, a few more properties of the 15 equations need to be shown; additionally, a few more utilities to work digit-wise with the base b representation of natural numbers need to be developed.

5 All recursively enumerable sets are Diophantine

In a final step, the equations obtained during the arithmetization of register machines need to be proven diophantine. Here, the result of section 3 is again crucial as many exponential relations occur due to the nature of aggregation over time by finite geometric sums as above. The connection to recursively enumerable sets is then readily made as exactly the sets accepted by a register machine are recursively enumerable. Register machines present one instance of an algorithm that can accept the elements of a recursively enumerable set, which is equivalent to having an algorithm that enumerates all elements of the set.

▶ **Definition 5.** *A set $\mathcal{A}$ is* recursively enumerable *if there exists a register machine, i.e. a program p, such that for the initial configuration $(k = 0, \mathcal{T} = [a, 0, \ldots, 0])$, we have $a \in \mathcal{A}$ if and only if the register machine halts after executing p on this configuration for a finite number of steps q.*

Combining all the results of the previous sections, the arithmetization of register machines and the diophantine representation of the resulting equations, including the diophantine representation of exponentiation, we can finally prove and formalize the DPRM theorem.

▶ **Theorem 6** (DPRM). `is_recursively_enumerable` $\mathcal{A} \implies$ `is_diophantine` $\mathcal{A}$

[2] The phrase "initially valid" refers to a set of common-sense validity assumptions about the program and initial configuration, e.g. that all references to registers and states are within bounds, that there is exactly one halt state, etc.

6 Conclusion

Summary of current progress. Our contribution comprises the partial formalization of the proof of the DPRM theorem in Isabelle. This includes an `is_diophantine` predicate for relations and sets, a library of digit-wise operations for natural numbers and corresponding utility functions and lemmas, and an implementation and arithmetization of register (Minsky) machines. The formalization is almost complete, in the sense that the bulk of the proof has been formalized, however two gaps remain. As a minor point, we are yet to complete the proof that the equations obtained from the arithmetization of a register machine are sufficient for the machine to terminate. More importantly, however, we still need to extend the `is_diophantine` predicate and show that binary digit-wise multiplication and binary masking are diophantine relations. Then, we intend to contribute this project to the Isabelle Archive of Formal Proofs (`https://www.isa-afp.org`).

Note that the project is carried out solely by undergraduate students (except the last named author, who is their supervisor and not directly involved in the implementation). They all, including the supervisor, had no prior experience in formalizing proofs. With an overall time span of so far 20 months, this is – to the best of our knowledge – the first major theorem formalized entirely by non-experts in theorem proving. For a more detailed discussion of these aspects of the project, and a reflection of the learning process, please refer to [1].

Related work. Related work on both the DPRM theorem and Hilbert's tenth problem has been carried out in Coq, Mizar and Lean. Larchey-Wendling and Forster [3], working in Coq, recently formalize a clever alternative using Conway's FRACTRAN language to simulate register machines and show undecidability of Hilbert's tenth problem in general. Working in Mizar, Pąk [5] published several articles on formalizing arithmetic properties related to Diophantine equations, notably that exponentiation is diophantine. Carneiro [2], using Pell equations, formalized that exponentiation is diophantine in Lean.

Future outlook. In order to arrive at undecidability of Hilbert's tenth problem from the DPRM theorem, a connection to the undecidability of the Halting problem will need to be made. This requires reference to a specific model of computation, for example our register machines. One possibility is to prove their equivalence to the Abacus or Turing machines formalized by Xu et al. [6] who have previously obtained a suitable undecidability result. Alternatively, the undecidability of our implementation of register machines could be shown directly. Future work may extend this contribution to formalize the whole solution of Hilbert's tenth problem in Isabelle – in the spirit of Hilbert himself.

References

1 Jonas Bayer, Marco David, Abhik Pal, and Benedikt Stock. Beginners' Quest to Formalize Mathematics: A Feasibility Study in Isabelle. In C. Kaliszyk, E. Brady, A. Kohlhase, and C. Sacerdoti Coen, editors, *Conference on Intelligent Computer Mathematics.*, volume 11617 of *Lecture Notes in Computer Science*, 2019. (to appear).

2 Mario Carneiro. A Lean formalization of Matiyasevič's Theorem. `arXiv:1802.01795v1`.

3 Yannick Forster Dominique Larchey-Wendling. Hilbert's Tenth Problem in Coq. URL: `http://www.ps.uni-saarland.de/Publications/documents/Larchey-WendlingForster_2019_H10_in_Coq.pdf`.

4 Yuri Matiyasevich. On Hilbert's Tenth Problem. In Michael Lamoureux, editor, *PIMS Distinguished Chair Lectures*, volume 1. Pacific Institute for the Mathematical Sciences, 2000.

5 Karol Pąk. Progress in the Formalization of Matiyasevich's theorem in the Mizar system. URL: http://alioth.uwb.edu.pl/~pakkarol/articles/FMM_2018_KP.pdf.

6 Jian Xu, Xingyuan Zhang, and Christian Urban. Mechanising Turing Machines and Computability Theory in Isabelle/HOL. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving. ITP 2013.*, volume 7998 of *Lecture Notes in Computer Science*, pages 147–162. Springer Berlin, Heidelberg, 2013.

Hammering Mizar by Learning Clause Guidance

Jan Jakubův
Czech Technical University in Prague, Czech Republic

Josef Urban
Czech Technical University in Prague, Czech Republic

Abstract

We describe a very large improvement of existing hammer-style proof automation over large ITP libraries by combining learning and theorem proving. In particular, we have integrated state-of-the-art machine learners into the E automated theorem prover, and developed methods that allow learning and efficient internal guidance of E over the whole Mizar library. The resulting trained system improves the real-time performance of E on the Mizar library by 70% in a single-strategy setting.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Computing methodologies → Theorem proving algorithms; Computing methodologies → Machine learning

Keywords and phrases Proof automation, ITP hammers, Automated theorem proving, Machine learning

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.34

Category Short Paper

Funding Supported by the *AI4REASON* ERC Consolidator grant number 649043, and by the Czech project AI&Reasoning CZ.02.1.01/0.0/0.0/15_003/0000466 and the European Regional Development Fund.

1 Introduction

Proof automation for interactive theorem provers (ITPs) has been a major factor behind the recent progress in formal verification. In particular, *Hammers* linking ITPs with automated theorem provers (ATPs) produce a major speedup of formalization [4]. The main AI component of existing hammers has so far been *premise selection* [1], where only the most relevant facts are chosen from the large ITP libraries as axioms for proving a new conjecture. Machine learning from the large number of proofs in the ITP libraries has resulted in the strongest premise selection methods [1, 17, 19, 8, 3, 2]. Premise selection however does not guide the theorem proving processes once the premises are selected. The success of machine learning in the high-level premise selection task has motivated development of low-level *internal proof search guidance*. This has been recently started both for ATPs [29, 18, 15, 23, 9] and also in the context of tactical ITPs [10, 13].

Recently, we have added [6] two state-of-the-art machine learning methods to the ENIGMA [15, 16] algorithm that efficiently guides saturation-style proof search in ATPs such as E [25, 26]. The first method trains gradient boosted trees on efficiently extracted manually designed (handcrafted) clause features. The second method uses end-to-end training of recursive neural networks, thus removing the need for handcrafted features . While the second method seems very promising and already improves on a simpler linear classifier when used for guidance, its efficient training and use over a large ITP library is still practically challenging. On the other hand, our recent experiments with efficient *feature hashing* have shown that the very good performance of gradient boosted trees is maintained even after significant dimensionality reduction of the feature set [6]. This opens the way to training

learning-based internal guidance of saturation search even on very large ITP libraries, where the hundreds of thousands of handcrafted features would otherwise make the trained guiding systems impractically slow.

In this work we conduct the first practical evaluation of learning-based internal guidance of state-of-the-art saturation provers such as E in a realistic large-library hammer setting, with realistic time limits. The results turn out to be unexpectedly good, improving the real-time performance of E on the whole Mizar Mathematical Library (MML) [12] by 70% in a single-strategy setting. We believe that this is a breakthrough that will quickly lead to ubiquitous deployment of ATPs equipped with learning-based internal guidance in large-theory theorem proving and in hammer-style ITP assistance.

The rest of the paper is organized as follows. Section 2 summarizes the general saturation-style ATP setting and explains how machine learning can be trained and used over a large library of problems to guide the saturation search. Section 3 discusses the practical implementation of ENIGMA, i.e., the features, classifiers, and the feature hashing used to make the ENIGMA guidance both strong and efficient on a large library. Section 4 is our main contribution. We evaluate the latest ENIGMA on the whole Mizar Mathematical Library and show that in several iterations of proving and learning we can develop very strong strategies and solve in low time limits many previously unsolved problems.

2 Enhancing ATPs with Machine Learning

Automated Theorem Proving. State-of-the-art saturation-based automated theorem provers (ATPs) for first-order logic (FOL), such as E [25] and Vampire [22] are today's most advanced tools for general reasoning across a variety of mathematical and scientific domains. Many ATPs employ the *given clause algorithm*, translating the input FOL problem $T \cup \{\neg C\}$ into a refutationally equivalent set of clauses. The search for a contradiction is performed maintaining sets of *processed* (P) and *unprocessed* (U) clauses. The algorithm repeatedly selects a *given clause* g from U, moves g to P, and extends U with all clauses inferred with g and P. This process continues until a contradiction is found, U becomes empty, or a resource limit is reached. The search space of this loop grows quickly and it is a well-known fact that the selection of the right given clause is crucial for success. Machine learning from a large number of proofs and proof searches may help guide the selection of the given clauses.

E allows the user to select a *proof search strategy* $\mathcal{S}$ to guide the proof search. An E strategy $\mathcal{S}$ specifies parameters such as term ordering, literal selection function, clause splitting, paramodulation setting, premise selection, and, most importantly for us, the *given clause selection* mechanism. The given clause selection in E is implemented using a collection of *weight functions*. These weight functions are used in a round robin manner to select the given clause.

Machine Learning of Given Clause Selection. To facilitate machine learning research, E implements an option under which each successful proof search gets analyzed and the prover outputs a list of clauses annotated as either *positive* or *negative* training examples. Each processed clause which is present in the final proof is classified as positive. On the other hand, processing of clauses not present in the final proof was redundant, hence they are classified as negative. Our goal is to learn such classification (possibly conditioned on the problem and its features) in a way that generalizes and allows solving related problems.

Given a set of problems $\mathcal{P}$, we can run E with a strategy $\mathcal{S}$ and obtain positive and negative training data $\mathcal{T}$ from each of the successful proof searches. Various machine learning methods can be used to learn the clause classification given by $\mathcal{T}$, each method yielding a

classifier or *model* $\mathcal{M}$. In order to use the model $\mathcal{M}$ in E, $\mathcal{M}$ needs to provide the function to compute the weight of an arbitrary clause. This weight function is then used to guide future E runs.

Guiding ATPs with Learned Models. A model $\mathcal{M}$ can be used in E in different ways. We use two methods to combine $\mathcal{M}$ with a strategy $\mathcal{S}$. Either (1) we use $\mathcal{M}$ to select *all* the given clauses, or (2) we combine $\mathcal{M}$ with the given clause guidance from $\mathcal{S}$ so that roughly half of the clauses are selected by $\mathcal{M}$. Proof search settings other than given clause guidance are inherited from $\mathcal{S}$. We denote the resulting E strategies as (1) $\mathcal{S} \odot \mathcal{M}$, and (2) $\mathcal{S} \oplus \mathcal{M}$.

3 ENIGMA: Inference Guiding Machine

Machine Learning in Practice. ENIGMA [15, 16] is our *efficient* learning-based method for guiding given clause selection in saturation-based ATPs, implementing the framework suggested in the previous Section 2. First-order clauses need to be represented in a format recognized by the selected learning method. While neural networks have been very recently practically used for internal guidance with ENIGMA [6], the strongest setting currently uses manually engineered *clause features* and fast non-neural state-of-the-art gradient boosted trees library [5].

Clause Features. Clause features represent a finite set of various syntactic properties of clauses, and are used to encode clauses by a fixed-length numeric vector. Various machine learning methods can handle numeric vectors and their success heavily depends on the selection of correct clause features. Various possible choices of efficient clause features for theorem prover guidance have been experimented with [15, 16, 20, 21]. The original ENIGMA [15] uses term-tree walks of length 3 as features, while the second version [16] reaches better results by employing various additional features.

Since there are only finitely many features in any training data, the features can be serially numbered. This numbering is fixed for each experiment. Let n be the number of different features appearing in the training data. A clause C is translated to a feature vector φ_C whose i-th member counts the number of occurrences of the i-th feature in C. Hence every clause is represented by a sparse numeric vector of length n. Additionally, we embed information about the conjecture currently being proved in the feature vector, yielding vectors of length $2n$. See [6, 16] for more details.

From Logistic Regression to Decision Trees. So far, the development of ENIGMA was focusing on fast and practically usable methods, allowing E users to directly benefit from our work. Simple but fast linear classifiers such as *linear SVM* and *logistic regression* efficiently implemented by the LIBLINEAR open source library [7] were used in our initial experiments [16]. Our recent experiments [6] report improved performance with *gradient boosted trees*, while maintaining efficiency. Gradient boosted trees are ensembles of decision trees trained by tree boosting. In particular, we use their implementation in the XGBoost library [5].

The model $\mathcal{M}$ produced by XGBoost consists of a set (*ensemble* [24]) of decision trees. The inner nodes of the decision trees consist of conditions on feature values, while the leafs contain numeric scores. Given a vector φ_C representing a clause C, each tree in $\mathcal{M}$ is navigated to the unique leaf using the values from φ_C, and the corresponding leaf scores are aggregated across all trees. The final score is translated to yield the probability that φ_C

Table 1 Number of Mizar problems solved in 10 seconds by various ENIGMA strategies.

	$\mathcal{S}$	$\mathcal{S} \odot \mathcal{M}_9^0$	$\mathcal{S} \oplus \mathcal{M}_9^0$	$\mathcal{S} \odot \mathcal{M}_9^1$	$\mathcal{S} \oplus \mathcal{M}_9^1$	$\mathcal{S} \odot \mathcal{M}_9^2$	$\mathcal{S} \oplus \mathcal{M}_9^2$	$\mathcal{S} \odot \mathcal{M}_9^3$	$\mathcal{S} \oplus \mathcal{M}_9^3$
solved	14933	16574	20366	21564	22839	22413	23467	22910	23753
$\mathcal{S}$%	+0%	+10.5%	+35.8%	+43.8%	+52.3%	+49.4%	+56.5%	+52.8%	+58.4
$\mathcal{S}+$	+0	+4364	+6215	+7774	+8414	+8407	+8964	+8822	+9274
$\mathcal{S}-$	-0	-2723	-782	-1143	-508	-927	-430	-845	-454

	$\mathcal{S} \odot \mathcal{M}_{12}^3$	$\mathcal{S} \oplus \mathcal{M}_{12}^3$	$\mathcal{S} \odot \mathcal{M}_{16}^3$	$\mathcal{S} \oplus \mathcal{M}_{16}^3$
solved	24159	24701	25100	25397
$\mathcal{S}$%	+61.1%	+64.8%	+68.0%	+70.0%
$\mathcal{S}+$	+9761	+10063	+10476	+10647
$\mathcal{S}-$	-535	-295	-309	-183

Table 2 Comparison of several developed strategies in higher time limits.

	$\mathcal{S}$ (30s)	$\mathcal{S} \oplus \mathcal{M}_9^2$ (30s)	$\mathcal{S} \oplus \mathcal{M}_9^2$ (60s)	$\mathcal{S} \oplus \mathcal{M}_9^3$ (60s)	$\mathcal{S} \oplus \mathcal{M}_{12}^3$ (60s)	$\mathcal{S} \oplus \mathcal{M}_{16}^3$ (60s)
solved	15554	24154	24495	24762	25540	26107
hard	75	891	956	1017	1192	1296

represents a positive clause. When using $\mathcal{M}$ as a weight function in E, the probabilities are turned into binary classification, assigning weight 1.0 for probabilities ≥ 0.5 and weight 10.0 otherwise. Our experiments with scaling of the weight by the probability did not yet yield improved functionality.

Feature Hashing. The vectors representing clauses have so far had length n when n is the total number of features in the training data $\mathcal{T}$ (or $2n$ with conjecture features). Experiments revealed that XGBoost is capable of dealing with vectors up to the length of 10^5 with a reasonable performance. This might be enough for smaller benchmarks but with the need to train on bigger training data, we might need to handle much larger feature sets. In experiments with the whole translated Mizar Mathematical Library, the feature vector length can easily grow over 10^6. This significantly increases both the training and the clause evaluation times. To handle such larger data sets, we have implemented a simple *hashing* method to decrease the dimension of the vectors.

Instead of serially numbering all features, we represent each feature f by a unique string and apply a general-purpose string hashing function[1] to obtain a number n_f within a required range (between 0 and an adjustable *hash base*). The value of f is then stored in the feature vector at the position n_f. If different features get mapped to the same vector index, the corresponding values are summed up. See [6] for more details.

4 Experiments

The experiments are done on a large benchmark of 57880 Mizar40 [19] problems[2] from the MPTP dataset [27]. Since we are here interested in internal guidance rather than in premise selection, we have used the small (*bushy*, re-proving) versions of the problems, however

[1] We use the following hashing function *sdbm*: $h_i = s_i + (h_{i-1} \ll 6) + (h_{i-1} \ll 16) - h_{i-1}$.

[2] `http://grid01.ciirc.cvut.cz/~mptp/7.13.01_4.181.1147/MPTP2/problems_small_consist.tar.gz`

Table 3 Training statistics and inference speed for different tree depths.

Tree depth	training error	real time	CPU time	model size (MB)	inference speed
9	0.201	2h41m	4d20h	5.0	5665.6
12	0.161	4h12m	8d10h	17.4	4676.9
16	0.123	6h28m	11d18h	54.7	3936.4

Table 4 Effect of looping on 10k randomly selected problems.

	$\mathcal{S}$	$\mathcal{S} \oplus \mathcal{M}^0$	$\mathcal{S} \oplus \mathcal{M}^1$	$\mathcal{S} \oplus \mathcal{M}^2$	$\mathcal{S} \oplus \mathcal{M}^3$	$\mathcal{S} \oplus \mathcal{M}^4$	$\mathcal{S} \oplus \mathcal{M}^5$	$\mathcal{S} \oplus \mathcal{M}^6$
solved	2487	3204	3625	3755	3838	3854	3892	3944
$\mathcal{S}$%	+0%	+28.8%	+45.7%	+50.9%	+54.3%	+54.9%	+56.4%	+58.5%

without previous ATP minimization. We start with a good evolutionarily optimized [14] E strategy $\mathcal{S}$ that performed best in previous experiments on the smaller MPTP2078 dataset. We run $\mathcal{S}$ for 10s on the whole library, producing the first proofs, we learn from them the next guiding strategy, and this is iterated with the growing body of proofs. All problems are run on the same hardware[3] and with the same memory limits employing multiple cores (around 300) for massive parallel evaluation.

Table 1 shows the number of Mizar problems solved in 10 seconds by the baseline strategy $\mathcal{S}$ and by each iteration of learning and proving with the learned guidance. The model $\mathcal{M}_9^0$ is trained on the training data coming from the problems solved by $\mathcal{S}$ with the maximum depth of XGBoost decision trees set to 9. We further *loop* this process and models $\mathcal{M}_9^n$ are trained on all the problems solved by $\mathcal{S}$, and by all the previous $\mathcal{S} \odot \mathcal{M}_9^k$ and $\mathcal{S} \oplus \mathcal{M}_9^k$ for $k < n$. Models $\mathcal{M}_{12}^3$ and $\mathcal{M}_{16}^3$ are trained on the same data as $\mathcal{M}_9^3$ but with the tree depth increased to 12 and 16. XGBoost models contain 200 decision trees and the hash base is set to 2^{15}. In the row $\mathcal{S}$% we show the percentage gain over the baseline strategy $\mathcal{S}$, while $\mathcal{S}+$ and $\mathcal{S}-$ are the additions and missing solutions w.r.t. $\mathcal{S}$. We can see that new problems are added with every iteration of looping. Combined versions ($\oplus$) typically perform better and lose less solutions. Increasing the tree depth to 16 leads to a strategy that outperforms the baseline by rather astonishing 70%.

Table 2 compares several of our new strategies with higher time limits and also shows the number of solved *hard* problems, i.e., the problems unsolved by any method developed previously in [19]. Our best strategy $S \oplus \mathcal{M}_{16}^3$ solves 26107 problems in 60s. Note that the 60s portfolio of our six best previous evolutionarily developed strategies for Mizar (i.e., each run for 10s) solves only 22068 problems, i.e., the single new strategy is 18.3% better. Vampire in the CASC (best portfolio) mode run in 300s has solved 27842 of these problems in 300s in [19].

Table 3 shows the training times, model sizes and inference speeds of XGBoost in the 4th iteration of proving and learning, using different tree depths. The training data is a sparse matrix with 65536 ($= 2 * 2^{15}$) columns (features) consisting of 63M examples. The total number of non-empty entries in the matrix is 5B (40GB). The inference speed is the average of the generated clauses per second measured on problems that timed out in all three runs. Note that despite the decrease of the inference speed with the more complicated XGBoost models, their accuracy and real-time performance grows (cf. Table 2). Training of better models on the millions of proof search examples however already requires significant resources – almost 12 CPU days for the best model with tree depth 16.

[3] Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz with 256G RAM.

Table 4 presents additional shorter experiments with more looping performed on a randomly selected 10k problems. The tree depth is set to 9. Again, the model $\mathcal{M}^0$ is trained only on the problems solved by $\mathcal{S}$ and the next models are obtained by looping. The highest improvement is achieved after the first learning ($\mathcal{M}^0$), however, the next iterations continue to add improvements.

5 Conclusion and Future Work

We have taken a good previously tuned E strategy and turned it into a learning-guided strategy that is 70% stronger in real time. We have done that by several iterations of MaLARea-style [28] feedback loop between proving and learning over a large mathematical library. The iterations here are however not done for learning premise selection as in MaLARea, but for learning efficient internal guidance. While developing this kind of efficient internal guidance for state-of-the-art saturation ATPs has been challenging and took time, the very large gains obtained here show that this has been very well invested effort. Future work will certainly focus on even stronger learning methods and also on more dynamic proof state characterization such as ENIGMAWatch [11]. It is however clear that this is the point when machine learning guidance has very strongly overtaken the human development of ATP strategies over large problem corpora.

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 Alexander A. Alemi, François Chollet, Niklas Eén, Geoffrey Irving, Christian Szegedy, and Josef Urban. DeepMath - deep sequence models for premise selection. In Daniel D. Lee, Masashi Sugiyama, Ulrike V. Luxburg, Isabelle Guyon, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 2235–2243, 2016. URL: http://papers.nips.cc/paper/6280-deepmath-deep-sequence-models-for-premise-selection.

3 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.

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 Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In *KDD*, pages 785–794. ACM, 2016.

6 Karel Chvalovský, Jan Jakubuv, Martin Suda, and Josef Urban. ENIGMA-NG: efficient neural and gradient-boosted inference guidance for E. *CoRR*, abs/1903.03182, 2019. arXiv:1903.03182.

7 Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008. doi:10.1145/1390681.1442794.

8 Michael Färber and Cezary Kaliszyk. Random Forests for Premise Selection. In Carsten Lutz and Silvio Ranise, editors, *Frontiers of Combining Systems - 10th International Symposium, FroCoS 2015, Wroclaw, Poland, September 21-24, 2015. Proceedings*, volume 9322 of *Lecture Notes in Computer Science*, pages 325–340. Springer, 2015. doi:10.1007/978-3-319-24246-0_20.

9 Michael Färber, Cezary Kaliszyk, and Josef Urban. Monte Carlo tableau proof search. In Leonardo de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 563–579. Springer, 2017. doi:10.1007/978-3-319-63046-5_34.

10 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 Series in Computing*, pages 125–143. EasyChair, 2017. URL: http://www.easychair.org/publications/paper/340355.

11 Zarathustra Goertzel, Jan Jakubuv, and Josef Urban. ProofWatch meets ENIGMA: First experiments. In Gilles Barthe, Konstantin Korovin, Stephan Schulz, Martin Suda, Geoff Sutcliffe, and Margus Veanes, editors, *LPAR-22 Workshop and Short Paper Proceedings*, volume 9 of *Kalpa Publications in Computing*, pages 15–22. EasyChair, 2018. doi:10.29007/z7qx.

12 Adam Grabowski, Artur Korniłowicz, and Adam Naumowicz. Mizar in a nutshell. *J. Formalized Reasoning*, 3(2):153–245, 2010.

13 Thomas Gransden, Neil Walkinshaw, and Rajeev Raman. SEPIA: search for proofs using inferred automata. In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pages 246–255, 2015. doi:10.1007/978-3-319-21401-6_16.

14 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.

15 Jan Jakubův and Josef Urban. ENIGMA: efficient learning-based inference guiding machine. In Herman Geuvers, Matthew England, Osman Hasan, Florian Rabe, and Olaf Teschke, editors, *Intelligent Computer Mathematics - 10th International Conference, CICM 2017, Edinburgh, UK, July 17-21, 2017, Proceedings*, volume 10383 of *Lecture Notes in Computer Science*, pages 292–302. Springer, 2017. doi:10.1007/978-3-319-62075-6_20.

16 Jan Jakubův 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.

17 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.

18 Cezary Kaliszyk and Josef Urban. FEMaLeCoP: Fairly efficient machine learning connection prover. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, volume 9450 of *Lecture Notes in Computer Science*, pages 88–96. Springer, 2015. doi:10.1007/978-3-662-48899-7_7.

19 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.

20 Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olsák. Reinforcement Learning of Theorem Proving. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada.*, pages 8836–8847, 2018. URL: http://papers.nips.cc/paper/8098-reinforcement-learning-of-theorem-proving.

21 Cezary Kaliszyk, Josef Urban, and Jirí Vyskocil. Efficient Semantic Features for Automated Reasoning over Large Theories. In *IJCAI*, pages 3084–3090. AAAI Press, 2015.

22 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.

23 Sarah M. Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep Network Guided Proof Search. 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 Series in Computing*, pages 85–105. EasyChair, 2017. URL: http://www.easychair.org/publications/paper/340345.

24 Robi Polikar. Ensemble based systems in decision making. *Circuits and Systems Magazine, IEEE*, 6(3):21–45, 2006.

25 Stephan Schulz. E - A Brainiac Theorem Prover. *AI Commun.*, 15(2-3):111–126, 2002. URL: http://iospress.metapress.com/content/n908n94nmvk59v3c/.

26 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.

27 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.

28 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.

29 Josef Urban, Jiří Vyskočil, and Petr Štěpánek. MaLeCoP: Machine learning connection prover. In Kai Brünnler and George Metcalfe, editors, *TABLEAUX*, volume 6793 of *LNCS*, pages 263–277. Springer, 2011. doi:10.1007/978-3-642-22119-4_21.

Declarative Proof Translation

Cezary Kaliszyk
University of Innsbruck, Austria
University of Warsaw, Poland
cezary.kaliszyk@uibk.ac.at

Karol Pąk
University of Białystok, Poland
pakkarol@uwb.edu.pl

Abstract

Declarative proof styles of different proof assistants include a number of incompatible features. In this paper we discuss and classify the differences between them and propose efficient algorithms for declarative proof outline translation. We demonstrate the practicality of our algorithms by automatically translating the proof outlines in 200 articles from the Mizar Mathematical Library to the Isabelle/Isar proof style. This generates the corresponding theories with 15301 proof outlines accepted by the Isabelle proof checker. The goal of our translation is to produce a declarative proof in the target system that is both accepted and short and therefore readable. For this three kinds of adaptations are required. First, the proof structure often needs to be rebuilt to capture the extensions of the natural deduction rules supported by the systems. Second, the references to previous items and their labels need to be matched and aligned. Finally, adaptations in the annotations of individual proof step may be necessary.

2012 ACM Subject Classification Theory of computation → Interactive proof systems

Keywords and phrases Declarative Proof, Translation, Isabelle/Isar, Mizar

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.35

Category Short Paper

Supplement Material The translated formalization is available at:
`http://cl-informatik.uibk.ac.at/cek/itp19mml200/`

Funding Polish National Science Center granted by decision n°DEC-2015/19/D/ST6/01473

1 Introduction

Declarative proof languages have been included in many proof assistants, since they provide more readable and more maintainable proofs. Examples include Isabelle/Isar [9], the Mizar proof language [5], Lean [4], the Coq declarative proof mode C_zar [3], and various declarative proof modes for HOL [10, 11, 6]. They all imitate natural deduction, because it has been developed as a minimal language capable of describing natural logical reasonings. However, all extend or modify natural deduction, usually depending on how they were developed or because of the motivations of the language creators. Some were designed to fit an existing infrastructure (for example an LCF prover), while some focus on imitating the mathematical practice. The largest example of the latter is the Mizar Mathematical Library (MML) [5, 2], which includes many constructs non-standard to natural deduction.

In this paper we discuss the incompatibilities between the declarative styles and propose translations between the features of such languages and showcase this on a large part of the Mizar Mathematical Library. The particular contributions are:

- A comparison of the features present in the declarative proof styles (Section 2) and efficient scalable translations that eliminate the features nor present in the other styles (Section 3);
- An automated translation of the declarative proof outlines of 200 articles from the Mizar Mathematical Library to Isabelle/Isar (Section 4). The application of the translation gives 15301 declarative toplevel proof outlines accepted by Isabelle in the Isabelle/Mizar object logic [8]. The proof skeleton transformation steps are all automatically correctly justified, but the justifications of the individual Mizar `by` steps are mostly not covered by the Isabelle/Mizar automation and are assumed.

Related work. We have [7] previously translated the toplevel statements of a smaller part of the MML to Isabelle without any proofs. Many translations between procedural proofs have been proposed in the past. Adams [1] gives an overview of such translations. Additionally he considers the efficiency of such translations, which has been a major issue for proof auditing, for example in the Flyspeck project. Proof translations between declarative proofs and procedural proofs in a single system has been considered before [11].

2 Declarative Proof Styles

We first discuss the features present in the declarative proof modes of different proof assistants and later present a table that compares the presence of these features in the systems (Table 1).

The two earliest declarative proof languages, the Mizar language [5] and Isabelle/Isar [9], differ most as they were developed quite differently. The former started as an extension of the Jaśkowski natural deduction. The latter tried to add declarative natural deduction elements to an LCF style theorem prover, which meant combining declarative proofs with procedural ones. These two styles have influenced declarative proof modes developed since.

A common feature of all such systems is a set of basic natural deduction steps (also referred to as *skeleton* steps). Matching these steps with the reasoning can be done explicitly, using a so-called *reasoning path*. The reasoning path is a list of rules used in procedural systems, which describes the process in which the goal needs to be transformed or simplified. We will first discuss the use of reasoning path in the various systems and their advantages, and later discuss other differences that arise.

Isabelle/Isar allows the goal to be transformed and rebuilt in a most flexible manner, however all transformation rules must be provided before the start of an individual reasoning. A drawback of such a solution is, for example, the treatment of the existential quantifier. In order to instantiate it, the suitable term needs to be available before the proof and cannot be constructed in the proof block. A simplification of the reasoning path that removes this restriction has been considered in Lean [4] where the `exists.intro` rule can be formulated after a witness is obtained.

A further restriction of the reasoning path makes the thesis completely implicit. This has been considered in Mizar, C_zar [3], and the two declarative modes for HOL Light (`miz3` [10, 11] and Harrison's Mizar Mode [6], which we will denote shortly MM_H). In such systems the implicit thesis can be referred to as `thesis`. A limited procedure for transforming it in every skeleton step is necessary. Additionally, the order of the skeleton steps is mostly specified by the shape of the proved formula. A partial conclusion allows specifying the proved conjunct and proceed to subsequent ones. C_zar is most flexible in this respect, since the implicit thesis can be transformed by the `reconsider thesis as` construction.

Table 1 Comparison of features present in the declarative proof styles of different proof assistants. `miz3` refer's to Wiedijk's Mizar mode for HOL and MM_H refers to Harrison's Mizar mode for HOL. For features present, but where their semantics slightly differ, we mark this with the syntax.

	Mizar	Lean	Isabelle/Isar	C_zar	miz3	MM_H
reason-path	–	+	+	–	–	–
inline $\exists_{\text{intro}}$	take	ex.intro	–	take	take	take
unfold	partial	partial	full	full	–	–
cases	after	before, EM	before, EM	after	after	after
thesis	thus/hence	show	show/thus	thus	thus	thus
$\exists_{\text{elim}}$	consider	obtain	obtain	consider	consider	consider
diffuse	now...end	–	{...}	–	now...end	–

Mizar is the only system that implicitly unfolds user-selected definitions to match the thesis to the provided skeleton steps. Unfolding definitions in all other systems is manual, and often all the occurrences of a given definition must be unfolded together. Isabelle/Isar and Lean include attributes that transform facts before their use (e.g. [simplified]).

The proof modes also include two possible ways how reasoning by cases is realized. In the first approach, the user specifies all the cases before the reasoning and then proceeds with each individual case. The second approach allows the user to directly prove the necessary cases. At the end of the reasoning the system will build the alternative based on the explicitly given cases and possibly ask the user to justify that all the cases have been covered. The latter approach has been considered in Mizar, miz3, MM_H, and C_zar. In Isabelle and Lean it is necessary to specify the cases (or give a formula ϕ for which excluded middle, EM will be used) before the reasoning.

Certain declarative modes support the extraction of information from nested proof blocks without explicitly giving the proof goal. This is referred to as a *diffuse statement* and supported by Mizar, miz3, and Isabelle/Isar. There are minor differences in the flexibility of such constructions, so we mark them by the corresponding syntax (now...end and {...}) in Table 1. Similarly, the existential elimination construction may or may not allow linking to the statement about the witness. We again mark this using the corresponding syntax (obtain / consider) in the table.

3 Translations

In this section we assume the the foundations are compatible and that we know how to translate the syntax of individual statements. A statement syntax translation will be necessary for each pair of systems and we will use one in the next section. A translation between two systems comprises of: rebuilding the proof structure to skeleton steps provided by the systems; adapting the references to previous items and labels; possibly adding the annotations of individual proof steps by the reasoning path. We will attempt to reconstruct the proof structure by introducing a small number of skeleton steps supported by the target system. The skeleton steps will be annotated only with the justification elements necessary in the target system, such as $\forall_{\text{intro}}$, $\Rightarrow_{\text{intro}}$, or explicit references to the conclusion (such as show). We discuss below eliminating particular features, if they are not supported by the target system. After the application of these transformation, the resulting proof text needs to be optimized to make use of the special features of the target system and the labels, references, and justifications updated.

∃ **introduction.** If not supported by the target system, they can be eliminated by introducing a cut with the existential formula available as a lemma and used in the reasoning path or explicitly given by a command, depending on the target system.

diffuse statement. In a similar way, diffuse statements (defined in the previous section 2) can be eliminated from the proof skeleton if they are not supported by the target proof system. For this, the thesis of the proof block needs to be reconstructed and explicitly provided.

cases. Proofs by cases are replaced by a case covering lemma and series of lemmas *case* → *thesis* justified by the reasonings given in the source system.

thesis reference. If the target system does not support a reference to the thesis, it is replaced by the formulation extracted from the source system. The only case where the target thesis is used, would be when the original thesis is not modified. For example in Isabelle, the use of proof- allows avoiding a repetition of the whole goal statement.

reasoning path. If the target system does require a reasoning path, the proof needs to be transformed to a shape where we can provide a correct reasoning path. In particular we assume that before any universal quantifier introduction (fix/let depending on the system) the thesis is universally quantified, for implication introduction (assume) it is an implication, and the show/thus is the formula or its first conjunct. This generates quite unnatural parentheses, which can be removed in a post-processing phase. Also note that in some systems (mostly logical frameworks) separating assume steps changes the reasoning path. The transformation follows the diagram:

skeleton step	new thesis	additional rules
fix/let x	$\forall x.$ thesis	ballI
assume A_1:α_1 and A_2:α_2 and...and A_{n-1}:α_{n-1} and A_n:α_n	$\alpha_1 \wedge (\alpha_2 \wedge (\ldots(\alpha_{n-1} \wedge \alpha_n)\ldots)) \longrightarrow$ thesis	$\underbrace{\text{impMI}, \ldots, \text{impMI}}_{n-1 \text{ times}}$, impI
show/thus α	$\alpha \wedge$ thesis	conjMI
take term	$\exists x.$ thesis(x:=term)	bexI[of "term"]

where impMI connects uncurry and impI; conjMI is a modification of conjI; ballI, bexI are the bounded quantifier introduction rules used with object-level types.

identifier scopes and namespaces. Newly introduced identifiers (x:=term) are also not treated uniformly across systems (for example in Mizar, the second kind of take construction may introduce a variable with the same name). In order to avoid problems, in cases where ambiguities can arise (it will be only 17 cases in all the proofs in the next section), identifiers will be renamed.

final thesis adjustment. The transformations discussed above derive for every block a thesis that is equivalent to the original one, but not always syntactically identical. If it is not identical, we introduce a cut in the target system. Finally the proof is adapted for readability in the target system, removing e.g. references to previous steps if they can be implicit or use then etc. Further refinements of the resulting text are left as future work.

4 Case Study

We have implemented these transformations and applied them to the 200 articles of the Mizar library obtaining natural deduction proof outlines that can be expressed in Isabelle/Isar. Isabelle accepts all the proof outlines, however the current Isabelle/Mizar automation is not able to handle most of the individual proof steps justifications yet, and these are assumed so far. In this section we showcase two original and translated lemmas. For details on the Isabelle/Mizar object logic and its notations we refer to [8].

```
scheme DrinkerParadox{P[set]}:
  ex x st P[x] implies for y holds P[y]
proof
  per cases;


    suppose ex x st not P[x];
      then consider x such that
A1:     not P[x];
      take x;

      assume P[x];
      hence for y holds P[y] by A1;

    end;


    suppose
A2:     for x holds P[x];

      take x=the set;

      assume P[x];
      thus for y holds P[y] by A2;
    end;


end;
```

```
theorem Drinker_paradox:
  ∃x. P(x) ⟶ (∀y. P(y))
proof-
  have cases: (∃x.¬P(x)) ∨ (∀x. P(x)) by auto
  have case1: (∃x.¬P(x)) ⟶ (∃t. P(t) ⟶ (∀y. P(y)))
  proof(rule impI)
    assume ∃x. ¬P(x)
    then obtain x where [ty]: x be set and
    A1: ¬P(x) by auto
    show ∃t. P(t) ⟶ (∀y. P(y))
    proof(rule bexI[of _ x],rule impI)
      assume P(x)
      thus ∀y. P(y) using A1 by simp
    qed auto
  qed
  have case2: (∀x. P(x)) ⟶ (∃x. P(x) ⟶ (∀y. P(y)))
  proof(rule impI)
    assume A2: ∀x. P(x)
    obtain x where [ty]: x be set and
    xDef: x = the set by auto
    show ∃x. P(x) ⟶ (∀y. P(y))
    proof(rule bexI[of _ x],rule impI)
      assume P(x)
      show ∀y. P(y) using A2 by simp
    qed auto
  qed
  show ?thesis using cases case1 case2 by auto
qed
```

Figure 1 Drinker's paradox in Mizar and its automated translation to Isabelle. Variables are implicitly typed as set. The example is a schematic extension of Wenzel and Wiedijk's example comparing Mizar with Isar [9].

In Figure 1 we present a simple proof that showcases the transformations the four different kinds of skeleton step reconstruction, variable rename in take, and uses existential introduction. In the proof automatically translated according to the introduced transformations Isabelle/Mizar's mauto works as a justification of every step. Every take step requires an additional obtain and type calculation. The proof by cases uses excluded middle, which is supported by Isabelle. Among the 3236 proofs by cases, 1354 required a justification that the considered cases are complete, and the most complex proof involves 16 cases.

Figure 2 showcases a more advanced MML proof, where automated thesis adjustments are also necessary. Also the Isabelle/Mizar automation does not support Mizar's term generation for properties, so the individual proof step justification required additional facts. These were symmetry $a+b = b+a$, reductions $a+b-a = b$, and the reflexivity of $\leq$. Last was for example necessary to derive B1: $\mathrm{t} <> \mathbf{0}_{\mathrm{M}}$ from A1. All other steps were successfully proved by mauto.

Among the 20233 subproofs in MML200, we need the additional cut to transform the thesis in 14827 cases (large majority are the same modulo parentheses). When it comes to definition unfolding, the unfolded definition needs to be explicitly provided. This occurs in 5144 subproofs. Inline existential introduction steps introduce 13027 additional proof blocks.

5 Conclusion

We proposed translation techniques for the various features present in declarative proof languages and we automatically translated the proof outlines from 200 articles of the MML to Isabelle/Isar. Isabelle accepts all the translated proof outlines and the increase in the proof size imposed by our translation is relatively small (factor 1.7) . Future work includes extending the translation to Mizar structures and proof schemes which would allow applying

```
theorem :: ROLLE:4 Lagrange Theorem
  for x,t be Real st 0<t
    for f be PartFunc of REAL,REAL st
      [.x,x+t.] c= dom f &
      f|[.x,x+t.] is continuous &
      f is_differentiable_on ].x,x+t.[
  ex s be Real st 0<s & s<1 &
    f.(x+t) = f.x + t*diff(f,x+s*t)
proof
  let x,t be Real such that
A1: 0<t;
  let f be PartFunc of REAL,REAL;
  assume [.x,x+t.] c= dom f &
    f|[.x,x+t.] is continuous &
    f is_differentiable_on ].x,x+t.[;
  then consider x0 be Real such that
A2: x0 in ].x,x+t.[ and
A3: diff(f,x0)=(f.(x+t)-f.x)/(x+t-x)
    by...

  take s = (x0-x)/t;


  x0 in {r where r is Real:x<r & r<x+t} by...
  then
A4: ex g be Real st g=x0 & x<g & g<x+t by...
  then 0<x0-x by...
  then 0/t < (x0-x)/t by...
  hence 0<s by ...
  x0-x<t by...
  then (x0-x)/t<t/t by...
  hence s<1 by...
A5: s*t+x = (x0-x)+x by...
  f.x+t*diff(f,x0)=f.x+(f.(x+t)-f.x) by...
  hence thesis by ...

end;
```

```
mtheorem Lagrange:
  ∀x:Real. ∀t:Real. 0_M < t ⟶
   ∀x:PartFunc of ℝ,ℝ.
     ([[ x,x+t ]] ⊆ dom f ∧
      f|_[x,x+t] be continuous) ∧
      f is differentiable on (] x,x+t [) ⟶
  ∃s:Real. 0_M < s ∧ (s < 1_M ∧
    f.(x+t) = f.x + t * diff(f,x+s*t))
proof(rule ballI,rule ballI,rule impI,rule ballI,rule impI)
  fix x assume [ty]: x be Real fix t assume [ty]: t be Real
  assume A1: 0_M < t   hence B1: t <> 0_M ...
  fix f assume [ty]: f be PartFunc of ℝ,ℝ
  have [ty]: f be Relation ...
  assume ([[ x,x+t ]] ⊆ dom f ∧ f|_[x,x+t] be continuous) ∧
     f is differentiable on (] x,x+t [)
  then obtain x0 where [ty]: x0 be Real and
  A2: x0 ∈ (] x,x+t [) and
  A3: diff(f,x0) = (f.(x+t) - f .x)/(x+t-x) ...
  obtain s where [ty]: s be set and sDef: s = (x0-x)/t ...
  have [ty]: s is Real ...
  show ∃s:Real. 0_M < s ∧ (s < 1_M ∧
    f.(x+t) = f.x + t * diff(f,x+s*t))
  proof(rule bexI[of _ s],rule conjMI,rule conjMI)
    have x0 ∈ {r where r be Real : x < r ∧ r < x + t} ...
    hence
    A4: ∃g:Real. (g = x0 ∧ x < g) ∧ g < x + t ...
    hence 0_M < x0 - x ...
    hence 0_M / t < (x0 - x)/t ...
    thus 0_M < s ...
    have x0 - x < t ...
    hence (x0 - x)/ t < t / t ...
    thus s < 1_M ...
    have A5: s * t + x = x0 - x + x ...
    have f.x + t * diff(f,x0) = f.x + (f.(x+t) - f.x) ...
    thus f.(x + t) = f.x + t * diff(f,x+s*t) ...
  qed
qed
```

Figure 2 The Lagrange theorem in Mizar and its automated translation to Isabelle. The individual proof step justifications have been omitted, and are available in the accompanying formalization.

the techniques to a large subsequent part of the Mizar library. Finally, developing a more powerful Mizar-like automation would be necessary to verify all the individual proof steps.

References

1 Mark Adams. Proof Auditing Formalised Mathematics. *J. Formalized Reasoning*, 9(1):3–32, 2016. URL: https://jfr.unibo.it/article/view/4576.

2 Grzegorz Bancerek, Czesław Byliński, Adam Grabowski, Artur Korniłowicz, Roman Matuszewski, Adam Naumowicz, and Karol Pąk. The Role of the Mizar Mathematical Library for Interactive Proof Development in Mizar. *Journal of Automated Reasoning*, 2017. doi:10.1007/s10817-017-9440-6.

3 Pierre Corbineau. A Declarative Language for the Coq Proof Assistant. In Marino Miculan, Ivan Scagnetto, and Furio Honsell, editors, *Types for Proofs and Programs, International Conference, TYPES 2007*, volume 4941 of *LNCS*, pages 69–84. Springer, 2007. doi:10.1007/978-3-540-68103-8_5.

4 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*, volume 9195 of *LNCS*, pages 378–388. Springer, 2015. doi:10.1007/978-3-319-21401-6_26.

5 Adam Grabowski, Artur Korniłowicz, and Adam Naumowicz. Four Decades of Mizar. *Journal of Automated Reasoning*, 55(3):191–198, 2015. doi:10.1007/s10817-015-9345-1.

6 John Harrison. A Mizar Mode for HOL. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: TPHOLs 1996*, volume 1125 of *LNCS*, pages 203–220. Springer, 1996. doi:10.1007/BFb0105406.

7 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, *11th International Conference on Intelligent Computer Mathematics (CICM 2018)*, volume 11006 of *LNCS*, pages 131–146. Springer, 2018. doi:10.1007/978-3-319-96812-4_13.

8 Cezary Kaliszyk and Karol Pąk. Semantics of Mizar as an Isabelle Object Logic. *Journal of Automated Reasoning*, 2018. doi:10.1007/s10817-018-9479-z.

9 Markus Wenzel and Freek Wiedijk. A Comparison of Mizar and Isar. *J. Autom. Reasoning*, 29(3-4):389–411, 2002. doi:10.1023/A:1021935419355.

10 Freek Wiedijk. Mizar Light for HOL Light. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2001*, volume 2152 of *LNCS*, pages 378–394. Springer, 2001. doi:10.1007/3-540-44755-5_26.

11 Freek Wiedijk. A Synthesis of the Procedural and Declarative Styles of Interactive Theorem Proving. *Logical Methods in Computer Science*, 8(1), 2012. doi:10.2168/LMCS-8(1:30)2012.

Formalization of the Domination Chain with Weighted Parameters

Daniel E. Severín
Depto. de Matemática, Universidad Nacional de Rosario, Argentina
CONICET, Argentina
http://www.fceia.unr.edu.ar/~daniel
daniel@fceia.unr.edu.ar

Abstract

The *Cockayne-Hedetniemi Domination Chain* is a chain of inequalities between classic parameters of graph theory: for a given graph G, $ir(G) \leq \gamma(G) \leq \iota(G) \leq \alpha(G) \leq \Gamma(G) \leq IR(G)$. These parameters return the maximum/minimum cardinality of a set satisfying some property. However, they can be generalized for graphs with weighted vertices where the objective is to maximize/minimize the sum of weights of a set satisfying the same property, and the domination chain still holds for them. In this work, the definition of these parameters as well as the chain is formalized in Coq/Ssreflect.

2012 ACM Subject Classification Mathematics of computing → Graph theory

Keywords and phrases Domination Chain, Coq, Formalization of Mathematics

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.36

Category Short Paper

Supplement Material The Coq formalization and the solver accompanying this paper can be found at https://dx.doi.org/10.17632/h5j5rvrz2r.2.

Funding Partially supported by grants ANPCyT PICT-2016-0410 and PID-UNR ING538.

Acknowledgements I want to thank Ricardo Katz for his careful reading and suggestions.

1 Introduction

The domination parameters and the relationship between them is a very active research area due to the numerous applications that can be modeled with them. They are introduced below, following the treatment given in the textbook [9].

Let $G = (V, E)$ be a simple graph. For any $v \in V$, let $N(v)$ be the set of vertices adjacent to v and $N[v] \doteq N(v) \cup \{v\}$. For any $S \subseteq V$, let $N(S) \doteq \bigcup_{v \in S} N(v)$ and $N[S] \doteq \bigcup_{v \in S} N[v]$.

A set $S \subseteq V$ is called a *stable set* if $N(S) \cap S = \emptyset$. Alternatively, S is a stable set if no vertex in S is adjacent to any other vertex in S. The *independence number* $\alpha(G)$ of a graph G is the maximum cardinality of a stable set in G.

A set $D \subseteq V$ is called a *dominating set* if $N[D] = V$. Alternatively, D is a dominating set if for all $v \in V - D$, there exists a vertex $u \in D$ such that u is adjacent to v. The *domination number* $\gamma(G)$ of a graph G is the minimum cardinality of a dominating set in G and the *independence domination number* $\iota(G)$ is the minimum cardinality of a set which is stable and dominating simultaneously.

A property p is called *hereditary* if whenever a set S satisfies p, so does every proper subset $S' \subset S$. Analogously, p is called *superhereditary* if whenever a set S satisfies p, so does every proper superset $S' \supset S$. "To be stable" is an hereditary property while "to be dominating" is superhereditary.

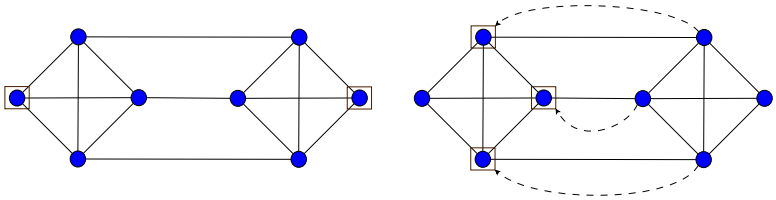


Figure 1 $\Gamma(G) = 2 < 3 = IR(G)$.

A set $S \subseteq V$ satisfying an hereditary property p is *maximal* if, for every $v \in V - S$, $S \cup \{v\}$ does not satisfy p. Similarly, a set S satisfying a superhereditary property p is *minimal* if, for every $v \in S$, $S - \{v\}$ does not satisfy p. For instance, a *minimal dominating set* D is a dominating set such that any proper subset of D is not dominating. Note that a dominating set of minimum cardinality is, in particular, minimal. Since finding $\gamma(G)$ is an NP-Hard problem, heuristics approaches to address them are usual and, in particular, a greedy heuristic consisting of adding elements to a set until it becomes dominating is one of these approaches. Such heuristic always returns a minimal dominating set by definition. Its worst case leads to the definition of the *upper domination number* $\Gamma(G)$ which is the maximum cardinality of a minimal dominating set in G.

For a given set $D \subset V$ and vertex $v \in D$, let $s_D(v) \doteq N[v] - N[D - \{v\}]$. This set has those vertices only dominated by v, whose are called *private vertices* of v in D. A set $D \subseteq V$ is called an *irredundant set* if, for every $v \in D$, $s_D(v) \neq \emptyset$. In other words, each vertex of D must dominate at least one vertex not dominated by any other vertex from D. The *upper irredundance number* $IR(G)$ is the maximum cardinality of an irredundant set in G. "To be irredundant" is an hereditary property and, thus, one might be interested in finding the minimum cardinality of a maximal irredundant set. The latter is called the *lower irredundance number* and denoted by $ir(G)$.

Figure 1 shows an example of a minimal dominating set (on the left) and an irredundant set (on the right). Both sets are represented by vertices inside boxes. In the right graph, arrows link vertices from the irredundant set to their private vertices. This graph is a known example where Γ and IR differs [10].

According to Favaron et al. [6], more that 1500 research papers about dominating sets have been published and, in particular, more than 100 explore properties of irredundant sets in graphs, showing the importance of this topic (which is still active [2]). Despite that, and to the best of my knowledge, these concepts have not been formalized yet.

This ongoing work intends to reduce the gap between what is already informally proved and what is not, such that other graph theorists may have a framework to formalize their results, especially when their proofs require the analysis of dozens of mechanical cases (the Four-Color Theorem is an example of a result involving an overwhelming number of cases [7]). In particular, this work is the basis to prove later that IR_w (defined in the next section) is polynomial on $\{claw, bull, P_6, \overline{C_6}\}$-free graphs [12]. However, it requires to consider several "boring" cases and its formalization could be a way to channel this result, reaching a twofold goal: on the one hand, to get confident about the proof and, on the other, to have the advantage that a reader can accept it without the need of manually checking step by step (it eventually could reduce the time spent in the peer-review process).

Another line of research that motivated this work is presented at the end of the paper.

A starting point is to formalize in Coq/Ssreflect [8] the Cockayne-Hedetniemi domination chain, which is the basis for many other results [9]. It states that for any graph G,

$$ir(G) \leq \gamma(G) \leq \iota(G) \leq \alpha(G) \leq \Gamma(G) \leq IR(G).$$

The proof relies on the following facts:

- A stable set D is maximal if and only if D is stable and dominating (see Prop. 3.5 of [9]).
- A maximal stable set is a minimal dominating set (see Prop. 3.6 of [9]).
- A dominating set D is minimal if and only if D is dominating and irredundant (see Prop. 3.8 of [9]).
- A minimal dominating set is a maximal irredundant set (see Prop. 3.9 of [9]).

For instance, in order to prove $ir(G) \leq \gamma(G)$ one can pick a dominating set D of minimum cardinality, i.e. $|D| = \gamma(G)$. Since D is minimal dominating, it is also maximal irredundant. Therefore, $|D| \geq ir(G)$.

2 Weighted parameters

The parameters defined in the previous section can be generalized as follows. For a given graph $G = (V, E)$, consider a positive integer weight $w(v)$ associated to each vertex v, i.e. $w : V \to \mathbb{N}_1$, where $\mathbb{N}_1$ denotes the set of natural numbers starting from 1. For any $S \subseteq V$, define the weight of S as $w(S) \doteq \sum_{v \in S} w(v)$. Let $\beta \in \{ir, \gamma, \iota, \alpha, \Gamma, IR\}$ be a parameter consisting of minimizing (or maximizing) the cardinality of a set S satisfying the corresponding property p (e.g. if $\beta = \alpha$ then the objective is "to maximize" and p is "to be a stable set"), and define $\beta_w(G)$ as the value of $w(S)$ such that S satisfies p and minimizes (maximizes resp.) $w(S)$.

Since weights are positive, sets of minimum (maximum resp.) weight are also minimal (maximal resp.), and the domination chain still holds for these parameters:

▶ **Theorem 1.** *For any graph G and weights $w : V(G) \to \mathbb{N}_1$, $ir_w(G) \leq \gamma_w(G) \leq \iota_w(G) \leq \alpha_w(G) \leq \Gamma_w(G) \leq IR_w(G)$.*

In particular, the problems of finding $\gamma_w(G)$ and $\alpha_w(G)$ are the classic optimization problems Minimum Weighted Dominating Set and Maximum Weighted Stable Set. Nevertheless, the weighted versions of the remaining parameters are also beginning to be studied: some theoretical results about $\Gamma_w(G)$ have recently been reported [3] and algorithms for obtaining (a generalized form of) $\iota_w(G)$ have been proposed [4].

Therefore, it makes sense to directly formalize the domination chain for the weighted case, and the original chain can be proved straightforwardly by setting $w(v) = 1$ for all $v \in V$. The code accompanying this paper (from now on, *the code*) contains 3 Coq files, described below:

Name	Definitions	Proofs	Lines (spec)	Lines (proof)
`basics.v`	12	46	282	303
`dom.v`	55	60	311	551
`example.v`	1	17	62	166

The second and third column display the number of global definitions and proofs, and the fourth and fifth column show the number of lines of specification and proof reported by the tool `coqwc`. The total number of lines (spec + proof) amounts to 1675. Also, there is a browsable version of the code made with *CoqDocJS*, and a solver for computing parameters γ_w, ι_w, α_w, Γ_w and IR_w. The solver can also generate a Coq file with a proof of $\alpha(G) \geq k$.

3 Graph definition

This section is devoted to briefly discussing how to represent a finite simple graph in the language Coq. First, a description of current representations is given.

The Mathematical Components library [13] (from now on, *MC library*) is equipped with a definition of finite graphs, which can be consulted in the file `fingraph.v`. Basically, vertices are elements of a finite type T and a graph is represented by a function of type $T \to$ `seq` T, i.e. an assignment from vertices to lists containing their adjacencies (here, `seq` is the ssreflect type for sequences, see `seq.v` from [13]). This library also has some basic results about connectivity, which is seen as the transitive closure of the adjacency relation, and they are used in the formal proof of the Four-Color Theorem [7] (at that time the file was called `connect.v`).

Recently, another representation was given by Dockzal, Combette and Pous [5] since `fingraph` in the MC library as well as other results in the formal proof of the Four-Color Theorem were conceived to deal with planar graphs and do not fulfill some requirements needed for a general theory of graphs. The authors define a graph G as a structure $\langle V, R\rangle$ (called `sgraph`) where V is a finite type inhabited by the vertices of G and $R : V \to V \to$ `bool` is a symmetric and irreflexive relation representing the adjacency relation of G (and denoted by "`--`"). Several results about connectivity, morphisms, minor relation and treewidth among others are formalized.

In this work, the latter representation is adopted. Moreover, the code is compatible with the one proposed in [5] and it can certainly extend that library.

Formalizations of some aspects of graph theory are not restricted to the language Coq. One of them is the work of Noschinski [11] for Isabelle/HOL. He defines a simple graph (not necessarily finite) as a pair $(V, E) \in \mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathcal{P}(\mathbb{N}))$ (where $\mathcal{P}(X)$ denotes the powerset of X) satisfying the condition $\forall\, e \in E \bullet e \subseteq V \wedge |e| = 2$. As this set-theoretic representation can be more intuitive for newcomers, the file `basics.v` (from the code) defines the edge set $E(G)$ in terms of the adjacency relation. Some results are then expressed with $E(G)$, including one of the first classic facts given in textbooks: the sum of the degrees of all vertices is equal to twice the number of edges.

```
Theorem sumdeg_2E : ∀ G : sgraph, 2 * #|E(G)| = Σ(w in V(G)) deg G w.
```

The file `basics.v` also has simple results about finite sets and summations not found in the MC library, and definitions of open and closed neighborhoods, and the degree of vertices.

4 Formalizing the domination chain

This section exposes the most relevant details about the formalization performed in the file `dom.v`, which contains definitions and results about: 1) stable, dominating and irredundant sets, 2) private sets, 3) hereditary and superhereditary properties, 4) maximal and minimal sets, 5) sets of maximum and minimum weights, and 6) weighted and unweighted parameters.

One of the obstacles found was that it was easier to prove statements about properties over sets when they were defined as `Prop`-terms rather than `bool`-terms, while the MC library commonly uses the latter. For that reason, the concept of *property* was packaged in a structure, where a property p comes in two flavors: `vsbool`, which is a compact definition of p having type {`set G`} $\to$ `bool` (see the definition of `pred` in the MC library), and `vsprop`, which is the same property written in terms of quantifiers and having type {`set G`} $\to$ `Prop`, where {`set G`} denotes the type of sets of vertices:

```
Record vsproperty := VertexSetProperty {
  vsprop :> {set G} → Prop ;
  vsbool :  pred {set G} ;
  vsrefl :  ∀ D : {set G}, reflect (vsprop D) (vsbool D) ;
  vsinhb :  {set G} ;
  vspinh :  vsprop vsinhb
}.
```

A boolean reflection view `vsrefl` is used to prove the equivalence between the two. In addition, the structure is equipped with a set `vsinhb` satisfying the property p. Its proof is given in `vspinh`. For instance, stable sets are defined as follows:

```
Definition stable := @VertexSetProperty
       p_stable pb_stable stableP ∅ st_empty.
```

where `p_stable` and `pb_stable` are the two versions given below, `stableP` is the reflection view between them, and `st_empty` is a proof that the empty set is stable.

```
Definition p_stable := ∀ u v : G, u ∈ S → v ∈ S → ¬ (u -- v).
Definition pb_stable := NS(S) ∩ S == ∅.
```

In the code, `NS(S)` is notation for $N(S)$, i.e. the open neighborhood of a set S.

Having different definitions in both types is useful and has already been applied previously in the MC library: for example, the lemma `setOPn` proves the equivalence between the `Prop`-term "∃ `x, x ∈ A`" and the `bool`-term "`A != ∅`" in the file `finset.v`. As it was pointed out previously, `vsbool` is mainly used when interacting with the MC library while `vsprop` is preferred for performing proofs. A coercion between `vsproperty` and `vsprop` is declared since the latter is used intensively and improves readability.

For a given property p and a given set of vertices D, the latter is a maximal set if it satisfies p but no proper superset F of D does. In addition, if p is hereditary, it is possible to apply the definition of maximal set given in the introduction (here called `maximal_altdef`):

```
Definition maximal := p D ∧ (∀ F : {set G}, D ⊂ F → ¬ p F).
Definition hereditary := ∀ F : {set G}, F ⊆ D → p D → p F.
Theorem maximal_altdef : hereditary p →
       (maximal ↔ (p D ∧ (∀ v : G, v ∉ D → ¬ p (D ∪ [set v])))).
```

Something similar is done for the definitions of minimal and superhereditary. From now on, only concepts related to maximal sets are presented (keeping in mind that the same is done for minimal ones).

In order to define the property that a given set is maximal irredundant, it is required to propose an inhabitant of that property. The code gives a tool called `ex_maximal` for providing these kind of sets. For instance, `ex_maximal irredundant` generates a maximal irredundant set and `maximal_exists` gives a proof that the generated set satisfies that property.

Let p be a property, $F \doteq$ `vsinhb p`, i.e. a set satisfying p, and $pb \doteq$ `vsbool p`, i.e. the `bool`-version of the property. The following function provides a set of maximum weight:

```
Definition maximum_set := [arg max_(D > F | pb D) weight_set D].
```

where `weight_set` is the weight of a given set. Note that `maximum_set` is defined in terms of [`arg max_`$(D > F \mid P)$ M], a function from the MC library that returns an object D maximizing M subject to P, where P holds for F.

Now, we have all the elements to introduce the parameters. For instance, $IR_w(G)$ is defined as the weight of the irredundant set of maximum weight.

```
Definition IR_w := weight_set weight (maximum_set weight irredundant).
```

For unweighted cases, cardinality is used. That is:

```
Definition maximum_set_card := [arg max_(D > F | pb D) #|D|].
Definition IR := #|maximum_set_card irredundant|.
```

Then, the equivalence between both cases (when weights are `ones`) is established:

```
Lemma IR_is_IR1 : IR = IR_w ones.
```

Finally *Theorem 1* is proved and the original chain is derived as a consequence of that theorem. For instance, for the statements $\Gamma_w(G) \leq IR_w(G)$ and $\Gamma(G) \leq IR(G)$ we have:

```
Theorem Gamma_w_leq_IR_w : ∀ (G : sgraph) (weight : G → nat),
       Γ_w G weight ≤ IR_w G weight.
Corollary Gamma_leq_IR : ∀ G : sgraph, Γ G ≤ IR G.
```

The file `example.v` shows an example on how to use these concepts: it proves that a complete graph K satisfies $\alpha_w(K) = \Gamma_w(K) = IR_w(K) = \max\{w(v) : v \in V(K)\}$, by bounding $\alpha_w(K)$ from below and $IR_w(K)$ from above, and applying Theorem 1 for collapsing the three parameters. It is also shown that $ir(K) = \gamma(K) = \iota(K) = \alpha(K) = \Gamma(K) = IR(K) = 1$.

A future research line related to this work conceives the idea of obtaining the proof (as a Coq file) of the value of a parameter over instances of reasonable size. For example, suppose that a certain application is modeled as a Maximum Stable Set Problem and, after all, one wants to verify $\alpha(G) = k$, for some G and k. Certifying that $\alpha(G) \geq k$ is easy (i.e. polynomial in the size of G): propose a set of k vertices and prove that it is stable. In fact, this is done by the solver provided in the supplement material. Therefore, the effort should be put in the generation of a proof of $\alpha(G) \leq k$ as small as possible. Below, a technique is briefly elaborated. Consider an Integer Linear Programming formulation that models the problem, e.g. maximize $\sum_{i \in V(G)} x_i$ subject to $x_i + x_j \leq 1$ for all $i, j \in E(G)$ and $x_i \in \{0, 1\}$ for all i. By adding the contraint $\sum_{i \in V(G)} x_i \geq k + 1$, the formulation turns infeasible. Next, find an Irreducible Infeasible Subsystem (IIS). There are tools that perform this task, e.g. *Conflict Refiner* of IBM CPLEX (or, even better, one can get the *minimum* IIS by solving a set covering problem). Then, solve the IIS via Branch-and-Bound (the number of explored nodes can be reduced by using a *strong branching* strategy). It generates a tree where each leaf corresponds to a infeasible Linear Programming (LP) problem. Hence, the proof of $\alpha(G) \leq k$ consists mainly of enumerating these LP problems and certifying that each one is infeasible, which can be done via Farkas Lemma. The library of formalized LP concepts provided in [1] might be useful here. Integer LP formulations for γ_w, ι_w and α_w are well-known, while recent ones for Γ_w and IR_w have been proposed [12] and implemented (see supplement material).

References

1 X. Allamigeon and R. Katz. A Formalization of Convex Polyhedra Based on the Simplex Method. *J. Autom. Reasoning*, pages 1–23, 2018. doi:10.1007/s10817-018-9477-1.

2 C. Bazgan, L. Brankovic, K. Casel, H. Fernau, K. Jansen, K.-M. Klein, M. Lampis, M. Liedloff, J. Monnot, and V. Th. Paschos. The many facets of upper domination. *Theor. Comput. Sci.*, 717:2–25, 2018. doi:10.1016/j.tcs.2017.05.042.

3 A. Boyaci and J. Monnot. Weighted upper domination number. *Electron. Notes Discrete Math.*, 62:171–176, 2017. doi:10.1016/j.endm.2017.10.030.

4 P. P. Davidson, C. Blum, and J. Lozano. The weighted independent domination problem: Integer linear programming models and metaheuristic approaches. *Eur. J. Oper. Res.*, 265:860–871, 2018. doi:10.1016/j.ejor.2017.08.044.

5 C. Doczkal, G. Combette, and D. Pous. A Formal Proof of the Minor-Exclusion Property for Treewidth-Two Graphs. *Lect. Notes Comput. Sc.*, 10895:178–195, 2018. doi:10.1007/978-3-319-94821-8_11.

6 O. Favaron, T. Haynes, S. Hedetniemi, M. Henning, and D. Knisley. Total irredundance in graphs. *Discrete Math.*, 256:115–127, 2002. doi:10.1016/S0012-365X(00)00459-3.

7 G. Gonthier. Formal proof - the Four-Color Theorem. *Notices Amer. Math. Soc.*, 55:1382–1393, 2008. URL: http://www.ams.org/notices/200811/tx081101382p.pdf.

8 G. Gonthier and A. Mahboubi. An introduction to small scale reflection in Coq. *J. Form. Reason.*, 3:95–152, 2010. doi:10.6092/issn.1972-5787/1979.

9 T. Haynes, S. Hedetniemi, and P. Slater. *Fundamentals of Domination in Graphs*. Marcel Dekker, Inc., 1998.

10 M. Jacobson and K. Peters. Chordal graphs and upper irredundance, upper domination and independence. *Discrete Math.*, 86:59–69, 1990. doi:10.1016/0012-365X(90)90349-M.

11 L. Noschinski. Proof Pearl: A Probabilistic Proof for the Girth-Chromatic Number Theorem. *Lect. Notes Comput. Sc.*, 7406:393–404, 2012. doi:10.1007/978-3-642-32347-8_27.

12 D. Severín and G. Nasini. The Maximum Weighted Upper Irredundance Problem (in Spanish). In *Communications of the Unión Matemática Argentina*. UNLP, La Plata, September 2018.

13 The Mathematical Components team. Mathematical components, 2018. URL: http://math-comp.github.io/math-comp/.