13th International Conference on Interactive Theorem Proving

ITP 2022, August 7–10, 2022, Haifa, Israel

Edited by

June Andronick

Leonardo de Moura

Editors

June Andronick
Proofcraft, UNSW and the seL4 Foundation, Australia
june.andronick@proofcraft.systems

Leonardo de Moura
Microsoft Research, Redmond, WA, US
leonardo@microsoft.com

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

ISBN 978-3-95977-252-5

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-252-5.

Publication date
August, 2022

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.2022.0

ISBN 978-3-95977-252-5 **ISSN 1868-8969** **https://www.dagstuhl.de/lipics**

LIPIcs – Leibniz International Proceedings in Informatics

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

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

13th International Conference on Interactive Theorem Proving (ITP 2022).
Editors: June Andronick and Leonardo de Moura
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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, the thirteenth to be held under the ITP name, is co-located with the Federated Logic Conference (FLoC 2022), Haifa, Israel. Previous ITP conferences took place in Edinburgh 2010, Nijmegen 2011, Princeton 2012, Rennes 2013, Vienna 2014, Nanjing 2015, Nancy 2016, Brasilia 2017, Oxford 2018 and Portland 2019; 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 66 submissions (63 regular papers and 3 short papers). Each paper was systematically reviewed by at least three program committee members or appointed external reviewers, as a result of which the PC winnowed down the selection to be presented at the conference: 31 papers (30 regular papers and 1 short paper). 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 Amy Felty (School of Electrical Engineering and Computer Science, University of Ottawa) and Bohua Zhan (State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences). The present volume collects all the accepted papers contributed to the conference as well as the two invited papers. This is the third time that the ITP proceedings are published in the LIPIcs series. We thank all those at Dagstuhl for their responsive feedback on all matters associated with the production of the finished proceedings.

We are grateful to all of the FLoC organizers and thankful to the ITP Steering Committee for their guidance throughout.

13th International Conference on Interactive Theorem Proving (ITP 2022).
Editors: June Andronick and Leonardo de Moura

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

Modelling and Verifying Properties of Biological Neural Networks

Amy Felty
School of Electrical Engineering and Computer Science, University of Ottawa, Canada

Abstract

In this talk, I present a formal model of biological neural networks and discuss the use of model checking and interactive theorem proving to verify some of their properties. Having a formal model can increase our understanding of the behavior and properties of such networks, as well as provide insight into their response to external factors such as disease, medicine, and environmental changes. We focus on *neuronal* micro-networks, considering properties of single neurons as well as properties of slightly larger ones called *archetypes*, which represent specific computational functions. Archetypes, in turn, represent the building blocks of larger more complicated neuronal circuits. I first present work by colleagues on a model checking approach, and then present our joint work on a newer theorem proving approach. Using interactive theorem proving allows us to generalize the kinds of properties that we can prove. This work is joint with Abdorrahim Bahrami and Elisabetta De Maria.

2012 ACM Subject Classification Theory of computation → Logic and verification; Applied computing → Systems biology

Keywords and phrases Neuronal networks, Model checking, Theorem proving, Coq

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.1

Category Invited Talk

1 Summary

Formal verification has become increasingly important in a variety of areas such as providing important guarantees for software systems, and building an increasingly large library of formalized mathematics. In our work, we consider the application of such formal techniques in the area of biological systems; in particular, we consider human neural networks. This talk reports on our past [1, 5, 6] and current work in this area.

It would be extremely difficult to use real biological experiments to prove the results that are expected from the biological theory that we are working with. Our main goal is to contribute to this area via modelling and verification. A longer term goal is to be able to apply what we learn to treat medical disorders. As an example, we hope that continuing this line of work will lead to an ability to detect inactive regions of the human brain and to treat mental disorders. Furthermore, our approach can be generalized to the verification of other kinds of networks, such as regulatory, metabolic, or environmental networks.

We focus on human *neuronal* networks, which can be viewed as the micro level of human neural networks, where a node represents a single neuron. *Archetypes* group several neurons together in a circuit that performs a specific function. To model archetypes, we adopt boolean *spiking neuron models* [3], where a *leaky integrate-and-fire* (LI&F) model is used to describe the electrical properties of the neurons. Neuronal archetypes are represented as weighted directed graphs whose nodes are neurons and whose edges represent synaptic connections. Time is modelled as discrete steps, where at each time unit, the neurons of the network compute their membrane potential from various parameters such as the current input signals and a *leak factor* that expresses signals degrading over time in a given temporal window.

Two examples of basic neuronal *archetypes* of our model [8] include a simple linear series of neurons and two neurons forming a loop where the output of one feeds into the input of the other, and this input either activates or inhibits the other neuron. Such archetypes represent primitives that can be composed in various ways to form larger networks. We prove some basic properties that have been identified in discussions with neurophysiologists. For example, the *delayer* effect is a property that states that under certain conditions, a single neuron or a series of neurons has an output which is the same as the input after a certain amount of delay. As another example, the *filter* effect also applies to a single or a series of neurons, and expresses that under certain conditions two spikes are never emitted in two consecutive time steps. Furthermore, we show that every neuron always exhibits either the delayer or filter effect.

In the work of De Maria et. al. [7, 8], the synchronous reactive language Lustre is used to represent neuronal networks and express their properties, which are then checked automatically by the Kind2 [4] model checker. To do so, various parameters must be given fixed values, such as the number of neurons in a series, the number of time steps, the specific input sequence, the value of the leak factor, as well as other parameters used to compute the membrane potential. In our work [1, 6], we use the Coq Proof Assistant [2], express such properties generally so that they hold for any values of these parameters, and prove them interactively with the help of some limited automation. Neurons and archetypes are modelled using dependent records, and we rely on Coq's powerful mechanisms for carrying out proofs by structural induction and case analysis, as well as on its standard libraries for reasoning about a variety of basic data types such as lists and rational numbers. Our work also includes a comparison of the two approaches on a fixed set of properties (see [5]).

References

1 Abdorrahim Bahrami, Elisabetta De Maria, and Amy Felty. Modelling and verifying dynamic properties of biological neural networks in Coq. In *Ninth International Conference on Computational Systems-Biology and Bioinformatics (CSBio)*, pages 12:1–12:11, 2018.

2 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development–Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

3 Wulfram Gerstner and Werner M. Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002.

4 George Hagen and Cesare Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *2008 Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–9, 2008.

5 Elisabetta De Maria, Abdorrahim Bahrami, Thibaud L'Yvonnet, Amy Felty, Daniel Gaffé, Annie Ressouche, and Franck Grammont. On the use of formal methods to model and verify neuronal archetypes. *Frontiers of Computer Science*, 16(3), 2022. Article No: 163404.

6 Elisabetta De Maria, Joëlle Despeyroux, Amy Felty, Pietro Lió, Carlos Olarte, and Abdorrahim Bahrami. Computational logic for biomedicine and neurosciences. To appear as a chapter in an ISTE-Wiley book, 2020.

7 Elisabetta De Maria, Thibaud L'Yvonnet, Daniel Gaffé, Annie Ressouche, and Franck Grammont. Modelling and formal verification of neuronal archetypes coupling. In *Eighth International Conference on Computational Systems-Biology and Bioinformatics (CSBio)*, pages 3–10, 2017.

8 Elisabetta De Maria, Alexandre Muzy, Daniel Gaffé, Annie Ressouche, and Franck Grammont. Verification of temporal properties of neuronal archetypes modeled as synchronous reactive systems. In *Fifth International Workshop on Hybrid Systems Biology (HSB)*, pages 97–112, 2016.

User Interface Design in the HolPy Theorem Prover

Bohua Zhan
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

Abstract

HolPy is a new interactive theorem prover implemented in Python. It is designed to achieve a small trusted-code-base with externally checkable proofs, writing proof automation using a Python API, and permit a wide variety of user interfaces for different application scenarios.

In this talk, I will focus on the design of user interfaces in HolPy. While most interactive theorem provers today use text-based user interfaces, there have been several existing work aiming to build point-and-click interfaces where the user perform actions by clicking on part of the goal or choosing from a menu. In our work, we incorporate into the design extensive proof automation and heuristic suggestion mechanisms, allowing construction of proofs on a large scale using this method. We demonstrate the approach in two common scenarios: general-purpose theorem proving and symbolic computation in mathematics.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Proof assistants, User interface, Proof automation

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.2

Category Invited Talk

Funding This work was partially supported by the National Natural Science Foundation of China under Grant No. 62002351, and the Chinese Academy of Sciences Pioneer 100 Talents Program under Grant No. Y9RC585036.

Candle: A Verified Implementation of HOL Light

Oskar Abrahamsson
Chalmers University of Technology, Gothenburg, Sweden

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

Ramana Kumar
London, UK

Thomas Sewell
University of Cambridge, UK

Abstract

This paper presents a fully verified interactive theorem prover for higher-order logic, more specifically: a fully verified clone of HOL Light. Our verification proof of this new system results in an end-to-end correctness theorem that guarantees the soundness of the entire system down to the machine code that executes at runtime. Our theorem states that every exported fact produced by this machine-code program is valid in higher-order logic. Our implementation consists of a read-eval-print loop (REPL) that executes the CakeML compiler internally. Throughout this work, we have strived to make the REPL of the new system provide a user experience as close to HOL Light's as possible. To this end, we have, e.g., made the new system parse the same variant of OCaml syntax as HOL Light. All of the work described in this paper has been carried out in the HOL4 theorem prover.

2012 ACM Subject Classification Software and its engineering → Software verification

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

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.3

Supplementary Material *Software (Proofs and Prebuilt Binaries)*: `https://cakeml.org/candle`

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

Acknowledgements We want to thank Freek Wiedijk and Yong Kiam Tan. We are grateful for Freek Wiedijk's question at ITP'11. Following a presentation about the verification of a runtime for Milawa [10] at ITP'11, Wiedijk asked: "Can you do the same for HOL Light, please?" Wiedijk's question can be seen as the seed that set us thinking about the possibility of a verified HOL Light implementation and eventually lead us to construct the verified Candle ITP, presented in this paper. We want to thank Yong Kiam Tan for helping with some proofs involving the the CakeML type inferencer. These proofs were part of the proof of safety of CakeML's new read-eval-print loop.

1 Introduction

Interactive theorem provers (ITPs) for higher-order logic, such as HOL4, HOL Light, Isabelle/HOL and ProofPower, are designed to be as sound as possible. Their implementations follow an LCF-style architecture, which means that each prover has a small kernel that implements the inference rules of the hosted logic (higher-order logic) and the rest of the system is set up in such a way that all soundness-critical inferences must be performed by the functions inside the small kernel. The beauty of this approach is that there is not much soundness-critical source code, which means that this code can quite easily be manually inspected (or even verified). As a result, soundness bugs in these ITPs are very rare.

In search of ever stronger assurance guarantees, one might ask: is it really the case that only the code of the kernel needs to be trusted in order to trust the soundness of an entire ITP implementation? At the level of source code, the answer is yes. However, source code is not what runs on real machines. As a result, one should also take into consideration the implementation of the programming language that hosts the ITP. All of the ITPs mentioned above rely on complex implementations of functional programming languages, such as Poly/ML and OCaml, and they rely on their interactive implementations, where users can (and do) provide new program text while the ITP is running. The implementations of these hosting functional languages are far more complex than the kernels of these ITPs.

In this paper we address the question: is it possible to develop an ITP for higher-order logic (HOL) for which soundness can be proved down to the machine code that runs it? In prior work, soundness has been proved for kernels of ITPs, but not for entire HOL ITPs. Our question requires us to consider a proof of soundness for the prover including the *at runtime* user-provided source code, and beyond that, for the *interactive* implementation of the underlying functional programming language. Our answer is: yes, it is possible to develop such an end-to-end verified ITP for HOL, as we explain in this paper.

Contributions

This paper's contribution is a new ITP called Candle[1], which consists of a clone of HOL Light running on top of a proved-to-be-safe CakeML-based read-eval-print loop (REPL).

- Our verification efforts result in a machine-code program, the Candle prover, for which we have proved that any theorem statement that it outputs follows by the inference rules of HOL (and, since these rules are sound, is valid by the semantics of HOL). To the best of our knowledge, this is the most comprehensive soundness result proved for a HOL ITP.
- This development can be seen as a major case study of the CakeML project, since it touches on almost every aspect of the CakeML project. This work is the first use of CakeML's new source primitive called Eval, which allows compilation and execution of user-provided program text at runtime.
- The resulting Candle ITP is designed to provide a user experience as close to HOL Light's as possible. For this purpose, we have made the new system parse the same variant of OCaml syntax as HOL Light. Our aim is to make it as straightforward as possible to port HOL Light developments to Candle.

The work described in this paper has been carried out in the HOL4 theorem prover [14]. Our proofs and binaries of Candle are available at: `https://cakeml.org/candle`.

2 Approach

This section provides a high-level outline of our work on Candle. Subsequent sections provide more details.

Prior work that we build on

The results described in this paper build on substantial prior work. In particular, we build on our prior work on construction of a proved-to-be-sound implementation of a HOL Light-like kernel [9]. In that work, we proved that CakeML implementations of HOL Light's kernel

[1] The name Candle comes from the combination of CakeML and HOL Light.

functions are sound w.r.t. a formalisation of higher-order logic. Our new work on Candle also depends on many parts of the CakeML ecosystem: in particular, our proved-to-be-safe REPL relies on the CakeML compiler's ability to compile itself (bootstrapping).

Overview of new work

The new work in this paper can be divided into the following three high-level steps:

1. We prove at the source level that any reasonable program that contains the Candle kernel as a prefix can only output facts that follow from the inference rules of HOL (Sec. 3);
2. We use CakeML's new Eval primitive to construct a proved-to-be-safe read-eval-print loop (REPL) that is sufficient for HOL Light-like interaction (Sec. 4);
3. Using CakeML's compiler correctness theorem, we transport the source-level soundness results down to the machine code that is the real implementation (Sec. 5).

The work for Step 1 centres around a whole-program simulation proof which establishes that only acceptable values, v_ok, are present in the system. Here a value v is considered v_ok if all the soundness-critical values, i.e., the values representing HOL types, terms and theorems, within v are valid in the current logical context maintained by the Candle kernel. These soundness-critical values flow around unmodified outside of the kernel. The interesting case is when one of the kernel's functions is called. At these calls, we make use of our prior work on the soundness of the kernel functions. Throughout these proofs, a layer of complexity is added by the fact that CakeML's operational semantics is (almost completely) untyped.

Even though the proofs for Step 1 are mostly about maintaining v_ok throughout execution, the final soundness theorem proved in Step 1 is not about values. Instead, it is about what can be seen on the externally facing foreign-function interface (FFI). This is because our compiler correctness theorem talks about events on the FFI channels. As a result, the whole-program soundness theorem states that every output on a special theorem-printing FFI channel will only ever contain valid theorem statements.

In Step 2, the challenge was to build a REPL that allows the kind of interactivity that an ITP requires. Here we make use of an evaluate primitive, Eval, that has recently been added to the CakeML source language. This Eval primitive evaluates, at runtime, arbitrary user-provided code, which is exactly what one needs to build a program that implements a REPL. (We hope that our REPL is sufficiently similar to HOL Light's to be usable.) A key insight in this part of the work is that a full functional specification for the REPL is not required. For the purposes of our soundness theorem, it suffices to prove that the REPL is safe, i.e., it never gets the (untyped) operational semantics stuck.

Step 3 is an important step, even though it only takes a few lines of proof to complete. This step is a straightforward application of the compiler correctness theorem to: a theorem describing an in-logic evaluation of the compiler; the Candle soundness theorem proved in Step 1; and the safety theorem for the REPL from Step 2.

3 Proving source-level soundness of the Candle prover

This section explains our work for Step 1, i.e., how we prove, at the CakeML source level, that any reasonable program built from the Candle kernel is sound.

3.1 Idea: soundness-critical values only produced by kernel functions

The idea of LCF-style ITPs is that the soundness of the kernel functions together with the programming language-based protection of the soundness-critical datatypes (such as the datatypes representing HOL types, terms and theorems) imply that no malformed or false types, terms or theorems can be constructed in the ITP, no matter what user-provided code is executed at runtime.

The Candle ITP follows the LCF tradition. Our task is thus to formally show, in HOL4, that this design makes the Candle ITP sound. More specifically, in our case, the task is to prove that any reasonable program that contains the Candle kernel can only produce well-formed and sound types, terms and theorems of HOL.

3.2 Setting: CakeML's untyped operational semantics

In an LCF-style ITP, protection of soundness-critical datatypes is usually achieved by the type system of the implementation language. In ML languages, the usual route is to make the type, term, and theorem datatypes into abstract types using the module system. We take a hybrid approach, where the type system provides some of the protection, while the rest comes from syntactic safety-checks imposed by the REPL at runtime.

A source of complication arises, in our proofs, from the fact that CakeML's operational semantics is almost entirely untyped. CakeML's operational semantics is written in a functional big-step style that takes a CakeML program as input and either succeeds, returning a value or a raising exception; or gets stuck with a runtime type error. CakeML values include literals, vectors, type constructors, and function values. A function value contains code and a semantic environment, but very little type information. The CakeML operational semantics lacks information such as the type of function values.

The soundness of our type system and its inferencer implementation [16] allows us to limit ourselves to considering only programs with a non-erroneous semantics in our theorems. However, this does not rule out ill-typed programs from our proofs, as non-erroneous programs can still have ill-typed parts, as long as those parts are never executed.

3.3 Target: a theorem about externally observable events

The top-level correctness statement needs to be in terms of externally visible I/O events on the CakeML compiler's foreign-function interface (FFI). This goes against the natural way of thinking of soundness in terms of what values can and cannot be constructed during the execution of a program.

This theorem should state that whenever a value of the HOL theorem type is rendered as text and output on an FFI channel, then that value is indeed a true theorem of HOL. To achieve this, we need to separate the output of theorems rendered as text from any other output of the REPL, because the REPL can (and does) print all sorts of text during runtime; indeed, a user may instruct it to print any string of text that looks like a theorem, but isn't. Worse, the HOL Light pretty-printer is user-customisable and installed at runtime; we have no way of statically reasoning about this function in our proofs.

We put our soundness story on rock solid foundations by printing theorems on a special kernel-controlled FFI channel using a printer function which sits within the kernel. The output from this printer function is difficult to read, but it is unambiguous and invertible, meaning that a theorem and its logical context can be recovered from the text.

$$\frac{f \in \mathsf{kernel\_funs}}{\mathsf{inferred}\ ctxt\ f} \qquad \frac{\mathsf{TYPE}\ ctxt\ ty \quad \mathsf{TYPE\_TYPE}\ ty\ v}{\mathsf{inferred}\ ctxt\ v}$$

$$\frac{\mathsf{TERM}\ ctxt\ tm \quad \mathsf{TERM\_TYPE}\ tm\ v}{\mathsf{inferred}\ ctxt\ v} \qquad \frac{\mathsf{THM}\ ctxt\ th \quad \mathsf{THM\_TYPE}\ th\ v}{\mathsf{inferred}\ ctxt\ v}$$

Figure 1 The defining rules of the inferred predicate. TYPE, TERM and THM are predicates from the Candle soundness development, stating that a value is a well-formed type, term, or theorem, with respect to a logical context. TYPE_TYPE, TERM_TYPE and THM_TYPE are relations invented by the CakeML code synthesis tool [11] as it processes these types, stating that the deep-embedded CakeML values v are refinements of the shallow-embedded HOL values: ty, tm, th.

$$\frac{\mathsf{inferred}\ ctxt\ v}{\mathsf{kernel\_vals}\ ctxt\ v} \qquad \frac{\mathsf{kernel\_vals}\ ctxt\ v}{\mathsf{v\_ok}\ ctxt\ v} \qquad \frac{\mathsf{every}\ (\mathsf{v\_ok}\ ctxt)\ vs}{\mathsf{v\_ok}\ ctxt\ (\mathsf{Vectorv}\ vs)}$$

$$\frac{\mathsf{kernel\_vals}\ ctxt\ f \quad \mathsf{v\_ok}\ ctxt\ v \quad \mathsf{do\_partial\_app}\ f\ v = \mathsf{Some}\ g}{\mathsf{kernel\_vals}\ ctxt\ g}$$

$$\frac{\begin{array}{c}\mathsf{every}\ (\mathsf{v\_ok}\ ctxt)\ vs \\ \forall\, tag\ x.\ opt = \mathsf{Some}\ (\mathsf{TypeStamp}\ tag\ x) \Rightarrow x \notin \mathsf{kernel\_types}\end{array}}{\mathsf{v\_ok}\ ctxt\ (\mathsf{Conv}\ opt\ vs)}$$

Figure 2 A few of the defining rules of the v_ok predicate. Here do_partial_app constructs a partial application, but fails if the function is fully applied. Conv is a constructor value, and Vectorv is a vector value, illustrating the recursive definition of v_ok.

3.4 Proof: values stay wellformed

The Candle kernel uses datatypes to represent soundness-critical HOL values: types, terms and theorems (sequents). It also defines functions that consume and produce values of these datatypes. Syntactic safety checks imposed by the REPL prevent values from being created using the HOL type constructors. Thus, the only way to create new values of these datatypes at runtime is by using the kernel functions.

We wish to establish the soundness of this design formally: that any reasonable program executed from a state which contains only well-formed HOL values should arrive in a state which contains only well-formed values. Values and functions defined inside the kernel are well-formed. So are types containing only defined type operators, well-typed terms containing only known constants, and theorems for which there is a derivation in the HOL proof calculus.

We say that a value is *safe*, written v_ok, if it contains only well-formed HOL values, written inferred v; or, if it is not a HOL specific value, all of its sub-values are v_ok. Type constructors for HOL values are not v_ok (or they would satisfy inferred), nor are references maintained by the kernel, as they could be used to modify the kernel state. Figure 1 shows the definition of inferred, and some of the rules defining v_ok are shown in Figure 2.

We say that code is safe, written safe_dec, if it does not directly mention the kernel FFI channel, nor call the constructors for the HOL datatypes. Safe code is still allowed to pattern match on HOL constructors and call the kernel functions.

We lift the v_ok predicate to environments and semantics states. An environment is env_ok if its values are v_ok, and if it maps the HOL constructor names to the correct HOL types. The state predicate, state_ok, maintains that all non-kernel references contain v_ok values, and ensures that all kernel-owned references point to values that are refinements of

the references in the state of the shallow-embedded kernel. It also guarantees that all events on the kernel FFI channel come from well-formed theorems, written ok_event (defined in Section 3.6), and sets up the EVAL mechanism (described in Section 4.1) in such a way that it rejects code that is not safe_dec. Furthermore, state_ok asserts that the state has come far enough in its type numbering to not reuse a type number belonging to the kernel types.

We can now state and explain the proof of the following simulation result.

▶ **Theorem 1.** *Any execution of safe code (safe_dec), starting from a safe state (state_ok), in a safe environment (env_ok) either:*

- *diverges, producing a (potentially infinite) trace of ok_event I/O events; or*
- *ends in a state_ok post-state and results in an env_ok environment.*

The majority of the proof of Theorem 1 follows any run-of-the-mill CakeML simulation proof. The most interesting case is when a kernel function is applied to an argument, since this is the only case where soundness-critical values are not simply being propagated.

For each kernel function, we prove a safety result stating that, when the kernel function is applied to v_ok arguments, it produces v_ok results. For these function-specific safety proofs, we make use of theorems from prior work on verification of the functions of a HOL kernel.

- The CakeML code implementing the kernel's functions is automatically generated by a proof-producing code synthesis tool [11]. This tool proves, for each shallow embedding of a kernel function f, that, if the CakeML code generated for f is given arguments of the correct form/type, then the CakeML code will compute the same result as an application of f to those arguments (at the level of the shallow embedding of f).
- From prior work [9], we have a soundness theorem for each kernel function. These theorems state that when they are applied to well-formed HOL values (i.e. satisfying inferred), then they produce well-formed HOL values.

However, there is a challenge here brought by the untyped setting and the assumption "if the CakeML code generated for f is given arguments of the correct form/type". Our untyped setting means that we cannot always immediately know that the arguments passed to the kernel functions are of the right form/type. Instead, all we know is that they satisfy v_ok.

Our solution is to insert dead code that makes the operational semantics perform a dynamic type check. For example, the kernel function `ASSUME` has type `term -> thm`, but the operational semantics does not see that it will only be applied to values that are terms. We insert a `case`-expression that pattern matches on a top-level constructor of the term type (`Var`). This `case`-expression triggers a dynamic type check in our semantics.

```
fun ASSUME tm = ((case tm of Var _ _ => () | _ => ()); ...);
```

The inserted code, i.e. `(case tm of Var _ _ => () | _ => ())`, has no impact on performance since the compiler removes it as dead code.

3.5 Towards a top-level soundness theorem

The Candle ITP program is made up from the CakeML basis library, the Candle kernel, and the user-facing REPL. The safety of the REPL is discussed separately in Section 4. To instantiate Theorem 1, we need to show that the initial state and environments satisfy state_ok and env_ok, respectively.

The program starts by running the basis library, for which state_ok does not hold: at this stage, the kernel references are not yet allocated, and the counter for type numbering has not yet reached the kernel types. Hence, Theorem 1 is not applicable.

We prove a separate simulation theorem, stating that evaluation of the basis program produces an environment that is env_ok, and a state which contains only v_ok values and with a next type number counter set to the number used by the first HOL datatype definition. The type counter and the number of references grow monotonically during execution, and all values produced by a program refer only to type numbers and reference locations that do not exceed the counts kept in state. Thus all values produced by this execution are trivially v_ok. From the resulting state, it is possible to define the kernel types and its references, and end up in a state that is state_ok.

Showing that the Candle post-state and post-environment (where the REPL starts executing) is state_ok and env_ok is straightforward but tedious. We automate the process by making use of some simple facts about env_ok environments:

- All kernel functions and values are v_ok by definition.
- When two env_ok environments are merged, the result is also env_ok.
- When one adds a v_ok value to an env_ok environment, the result is also env_ok.

The result is a small piece of custom proof automation which steers HOL4 to a proof showing that the concrete environments of the kernel values are env_ok, thereby allowing us to establish state_ok and env_ok for the setting in which the REPL program executes.

We use these theorems together with Theorem 1 to show that the safety invariants v_ok, env_ok and state_ok are preserved in any subsequent code executed by the program.

3.6 Source-level soundness theorem

Our source-level soundness proof builds up to the following top-level soundness theorem stated in terms of CakeML's observable semantics, semantics_prog.

Here semantics_prog returns a set of behaviours. A behaviour is Fail (for type error), Terminate k l (for termination) or Diverge ll (for a non-terminating run). Here l is a list of I/O events performed by the run and ll is a potentially infinite list of I/O events.

We prove that each generated I/O event must be well-formed according to ok_event, which is defined to require that any event that communicates on the special kernel_ffi channel must contain output that can be produced using a thm_to_string function applied to a sequent th that can be derived (THM) by the inference rules of higher-order logic in a context $ctxt$.

$$\mathsf{ok\_event}\ (\mathsf{IO\_event}\ n\ out\ y) \stackrel{\mathsf{def}}{=} \\ n = \mathsf{kernel\_ffi} \Rightarrow \exists\, ctxt\ th.\ \mathsf{THM}\ ctxt\ th \wedge \mathsf{thm\_to\_string}\ ctxt\ th = out$$

Since the thm_to_string function is crucial for our soundness theorem, we have made sure that its output is invertible. The actual output is not particularly human readable in most cases, but it is unambiguous. A small sample output is shown in Figure 3.

Our source-level soundness theorem states that every event satisfies ok_event, for any non-Fail behaviour and for any program that consists of declarations candle_code ++ $prog$, where $prog$ is any list of declarations that syntactically satisfies every safe_dec.

▶ **Theorem 2.** *Any non-Fail behaviour res that is in the behaviours of candle_code ++ prog will only contain externally visible events that satisfy ok_event.*

$$\vdash res \in \mathsf{semantics\_prog}\ (\mathsf{init\_eval\_state\_for}\ cl\ fs)\ \mathsf{init\_env}\ (\mathsf{candle\_code} +\!\!+ prog) \wedge \\ \mathsf{every}\ \mathsf{safe\_dec}\ prog \wedge res \neq \mathsf{Fail} \Rightarrow \\ \forall\, e.\ e \in \mathsf{events\_of}\ res \Rightarrow \mathsf{ok\_event}\ e$$

```
# The following is a theorem of higher-order logic

(Sequent nil (Const T (Tyapp bool)))

# which is proved in the following context

(ConstSpec
   ((T ...))
   (Comb
      (Comb ...)
      (Comb ...)))

(NewConst = (Tyapp fun (Tyvar A) (Tyapp fun (Tyvar A) (Tyapp bool))))

(NewType bool 0)

(NewType fun 2)
```

Figure 3 Output of `print_thm` applied to the theorem $\vdash \mathsf{T}$. Here `Sequent` contains `nil` to indicate that there are no hypothesis on this theorem. This theorem is true in the context where T is defined as $\mathsf{T} \stackrel{\text{def}}{=} (\lambda\, p.\ p) = (\lambda\, p.\ p)$, which is its definition in HOL Light; equality `=` is a constant of type $\alpha \to \alpha \to \mathsf{bool}$; types `bool` and `fun` are defined. Some excess output is elided (...) above.

4 Construction of a proved-to-be-safe REPL for Candle

The previous section explained how we have proved that any reasonable program, one that satisfies every safe_dec, constructed from the Candle kernel leads to a sound prover. This section explains how we have built a program that satisfies every safe_dec and manages to implement a proved to-be-safe read-eval-print loop (REPL) that provides the kind of user-interaction that theorem proving with HOL Light requires.

4.1 CakeML's new Eval source primitive

The most technically demanding part of a REPL is the implementation of the "E" in REPL, i.e., the part that evaluates user input. This "E" must always run safely and efficiently.

To the best of our knowledge, most HOL Light users use HOL Light via the standard OCaml REPL[2] where the "E" compiles user input into bytecode and then interprets the bytecode. For CakeML and Candle, we implement the "E" in REPL as: compile, at runtime, the user input to machine code, drop that machine code into the code segment of the running process and execute the new machine code by performing a jump to it.

Such runtime compilation can be achieved in CakeML by using CakeML's new Eval source primitive. The exact details, implementation and verification of the new Eval source primitive will be the subject of a different publication. However, for this paper, it suffices to have an approximate understanding of its semantics and to know that the Eval primitive has been fully integrated into the CakeML compiler and its proofs.

From a bird's eye view, CakeML's Eval primitive has the following semantics: it expects as input (among other things): a value representing the AST for CakeML source declarations to execute, and a value holding a semantic environment (mappings from names to values and

[2] HOL Light adjusts the OCaml REPL so that it uses a custom HOL Light-specific OCaml parser.

```
01: fun repl (parse, types, conf, env, decs, input_str) =
02:   (* input_str is passed in here only for error reporting purposes *)
03:   case check_and_tweak (decs, types, input_str) of
04:     Inl msg => repl (parse, types, conf, env, report_error msg, "")
05:   | Inr (safe_decs, new_types) =>
06:     (* here safe_decs are guaranteed to not crash;
07:        the last declaration of safe_decs calls !Repl.readNextString *)
08:     case eval (conf, env, safe_decs) of
09:       Compile_error msg => repl (parse, types, conf, env, report_error msg, "")
10:     | Eval_exn e new_conf =>
11:       repl (parse, roll_back (types, new_types), new_conf, env, show_exn e, "")
12:     | Eval_result new_env new_conf =>
13:       (* check whether the program that ran has loaded in new input *)
14:       if !Repl.isEOF then () (* exit if there is no new input *) else
15:         let val new_input = !Repl.nextString in
16:           (* if there is new input: parse the input and recurse *)
17:           case parse new_input of
18:             Inl msg =>
19:               repl (parse, new_types, new_conf, new_env, report_error msg, "")
20:           | Inr new_decs =>
21:               repl (parse, new_types, new_conf, new_env, new_decs, new_input)
22:         end
```

Figure 4 CakeML code (in CakeML syntax) implementing the main loop of the new REPL.

type information); if called correctly, Eval evaluates the given declarations using the supplied environment, and, on successful completion, returns a value holding a new environment that can be used for subsequent calls to Eval.

4.2 Building a REPL in CakeML source code

The Eval primitive enables us to implement our REPL conveniently in CakeML source code. The source code for the main loop of the REPL is shown in Figure 4. This section attempts to explain the code shown in Figure 4.

When planning this implementation, a key insight was that we do not need to prove any input-output-style functional correctness theorem of the REPL. Instead, for the purposes of our top-level soundness theorem, it suffices to implement a REPL that we can prove to be safe. This safety proof needs to result in a theorem stating that a semantics_prog run of the REPL program can never result in the Fail behaviour, as can be seen in the assumption $res \neq$ Fail in Theorem 2. This insight means that we can leave it up to the user to decide how input is to be read and can leave the pretty printing code quite open ended too, i.e., mostly unverified.

We will illustrate the working of the code in Figure 4 using an example. For the sake of the example suppose the `repl` function is given the AST of the following CakeML declaration as the `decs` argument.

```
let x = [1] @ [2];;
```

As can be seen on line 3 in Figure 4, `check_and_tweak` will be applied to the `decs`. We can also see that the `types` argument (the state of the type inferencer) and `input_str` are also passed to `check_and_tweak`. This `check_and_tweak` function will run the type inferencer on the given `decs`. If the type inferencer rejects them, then an error message is returned. If the type inferencer accepts `decs`, then the `check_and_tweak` function will return a tweaked version of the original declarations. For this example, the tweaked declarations are approximately the following. (In reality, the second line uses more specialised functions.)

```
let x = [1] @ [2];;
let _ = print ("x" ^ pp_list pp_int x ^ ": int list\n");;
let _ = (!Repl.readNextString)();;
```

Here the first line is, in this case, exactly the user's input; the second line causes the computed value to be printed to stdout; and, the third line runs a user-settable function for reading the next input. In the general case, the `check_and_tweak` function also adds definitions of `pp`-functions to the given declarations.

Once these adjusted declarations, called `safe_decs` on line 5, have been generated, the `repl` function hands them over to `eval`, which runs them. Running these declarations can result in one of three outcomes: the compiler might not be able to compile them (linking error or similar); the evaluation might have caused a top-level exception, which `eval` catches and returns as `e` on line 10; or, if all goes well, the evaluation will return a new declaration-level semantic environment `new_env` and a new compiler configuration `new_conf` on line 12.

From line 12, the `repl` function continues by reading a few references that the execution of `!Repl.readNextString` is to have assigned new values to; a `true` in `!Repl.isEOF` indicates that there is no new input; if input exists, then the new input is in `!Repl.nextString`. In the case of new input, the parser is called on the content of `!Repl.nextString` and the loop begins from the top again.

The loop starts off by evaluating the declarations that correspond to the concrete syntax for `let _ = (!Repl.readNextString)();;`. The initial value of this `Repl.readNextString` reference is a function that returns the content of a user-modifiable start-up file `candle_boot.ml`. This file is supposed to install an appropriate new function in `Repl.readNextString`, which includes a user-configurable parser for `‘...‘`-terms, and support for special file loading directives.

One can argue that some aspects of the implementation of the `repl` function seem peculiar, e.g., that the call to `!Repl.readNextString` is always appended to the declarations sent to `eval`. Our design of `repl` is arrange this way in order to collect all state changing code into the execution of `eval`, since such a design makes the safety proof simpler.

4.3 Proving safety of the REPL

As mentioned above, we need to prove that the REPL is safe to execute. More specifically, that semantics_prog cannot give the REPL program the Fail behaviour.

The conventional way to prove safety of a CakeML program is via type inference: if the type inferencer accepts a program, then the program is typeable and, by type soundness, we know that the program is safe, i.e., does not have Fail behaviour. Unfortunately, we cannot take this route because the Eval primitive, in its current form, does not fit with CakeML's type system, since static typing information is not enough to show that the Eval won't get stuck when run. As a result, we prove safety of the REPL via an interactive proof.

We prove the following safety theorem for the REPL program called repl_source_prog.

▶ **Theorem 3.** *The REPL program does not have Fail behaviour.*

$$\vdash \mathsf{has\_repl\_flag}\ (\mathsf{tl}\ cl) \wedge \mathsf{basis\_init\_ok}\ cl\ fs \Rightarrow \\ \mathsf{Fail} \notin \mathsf{semantics\_prog}\ (\mathsf{init\_eval\_state\_for}\ cl\ fs)\ \mathsf{init\_env}\ \mathsf{repl\_source\_prog}$$

The most challenging aspect of the proof of Theorem 3 is that it requires bringing together results from different parts of the CakeML ecosystem. Fortunately, all proofs to do with type inference could quite cleanly be separated from the proofs about stepping through the operational semantics. It is worth noting that the REPL implementation uses the type

```
# let T_DEF = new_basic_definition `T = ((\p:bool. p) = (\p:bool. p))`;;
val T_DEF = |- T <=> (\p. p) = (\p. p): thm
# let th1 = SYM T_DEF
  and th2 = REFL `\p:bool. p`;;
val th1 = |- (\p. p) = (\p. p) <=> T: thm
val th2 = |- (\p. p) = (\p. p): thm
# let TRUTH = EQ_MP th1 th2;;
val TRUTH = |- T: thm
```

Figure 5 Sample interaction with the Candle REPL in the OCaml syntax of HOL Light.

inferencer to establish safety of the user-provided code, which means that the user can unfortunately not currently mention the EVAL primitive because EVAL is not typeable.

We also prove the following syntactic property about the REPL program in order to meet the assumptions of the Candle soundness theorem, Theorem 2.

▶ **Theorem 4.** *The REPL program has* candle_code *as a prefix and satisfies* every safe_dec.

$$\vdash \exists \mathit{prog}.\ \mathsf{repl\_source\_prog} = \mathsf{candle\_code} \mathbin{++} \mathit{prog} \wedge \mathsf{every}\ \mathsf{safe\_dec}\ \mathit{prog}$$

This theorem is proved by rewriting and evaluation. It is used in Section 5.

4.4 A REPL with a parser for HOL Light-style OCaml syntax

The sharp-eyed reader might have noticed that the CakeML code of Figure 4 does not use the OCaml syntax that HOL Light users expect. Instead it uses the standard way to write CakeML code, i.e., in syntax that is aligned with Standard ML. In order to make the user experience as close as possible to that of HOL Light, we have equipped the Candle REPL with a parser for HOL Light's version of OCaml syntax. Figure 5 shows a snippet of an interaction with the Candle REPL, where one can see a glimpse of OCaml-style concrete syntax supported by Candle. Figure 5 also shows our quote filter in action: it processes the quoted terms `...` correctly.

5 Proving soundness for the machine-code implementation

In this section, we apply the compiler to the source-level REPL implementation of Candle, and transport the safety and soundness proofs down to the level of the machine code that runs when the Candle prover is used.

We evaluate the CakeML compiler on the repl_source_prog program inside HOL4 in order to arrive at the concrete machine_code implementation of the REPL by proof in the logic. The resulting theorem is the following.

▶ **Theorem 5.** *The CakeML compiler produces* machine_code *when applied to* repl_source_prog.

$$\vdash \mathsf{compile}\ \mathsf{init\_conf}\ \mathsf{repl\_source\_prog} = \mathsf{Some}\ (\mathsf{machine\_code},\mathsf{ro\_data},\mathsf{result\_conf})$$

We use the CakeML compiler's correctness theorem to transport correctness properties down to the level of machine code. Theorem 6 below is an instantiated version of the relevant CakeML compiler correctness theorem. In this theorem, we collect a bunch of assumptions into a constant repl_ready_to_run *cl fs ms* which, among other things, requires that the generated machine_code is installed in memory in machine state *ms* and the program counter of *ms* points at the start of the code.

▶ **Theorem 6.** *If the source-level program* repl_source_prog *does not have* Fail *behaviour, then any machine-code level execution starting from a* repl_ready_to_run *machine state* ms *can only produce behaviours that are contained in the set of source-level behaviours (extended with the possibility of early exits due to hitting resource limits,* extend_with_resource_limit*).*

$$\begin{aligned}&\vdash \mathsf{Fail} \notin \mathsf{semantics\_prog}\ (\mathsf{init\_eval\_state\_for}\ cl\ fs)\ \mathsf{init\_env}\ \mathsf{repl\_source\_prog} \wedge \\ &\quad \mathsf{repl\_ready\_to\_run}\ cl\ fs\ ms \Rightarrow \\ &\quad \mathsf{machine\_sem}\ (\mathsf{basis\_ffi}\ cl\ fs)\ ms \subseteq \\ &\quad \mathsf{extend\_with\_resource\_limit} \\ &\qquad (\mathsf{semantics\_prog}\ (\mathsf{init\_eval\_state\_for}\ cl\ fs)\ \mathsf{init\_env}\ \mathsf{repl\_source\_prog})\end{aligned}$$

We will not go into details of extend_with_resource_limit, but only note that it is trivial to prove the following interaction between events_of and extend_with_resource_limit.

$$\begin{aligned}&\vdash e \in \mathsf{events\_of}\ res_1 \wedge res_1 \in sem_1 \wedge sem_1 \subseteq \mathsf{extend\_with\_resource\_limit}\ sem_2 \Rightarrow \\ &\quad \exists\, res_2.\ e \in \mathsf{events\_of}\ res_2 \wedge res_2 \in sem_2\end{aligned}$$

We now have all of the parts required to prove a soundness theorem for Candle that relates the level of machine code to the ok_event from Section 3.

▶ **Theorem 7.** *Any behaviour* res *of a machine execution from a* repl_ready_to_run *machine state* ms *will not* Fail*, and any event* e *in* res *will always satisfy* ok_event*.*

$$\begin{aligned}&\vdash res \in \mathsf{machine\_sem}\ (\mathsf{basis\_ffi}\ cl\ fs)\ ms \wedge \mathsf{repl\_ready\_to\_run}\ cl\ fs\ ms \Rightarrow \\ &\quad res \neq \mathsf{Fail} \wedge \forall\, e.\ e \in \mathsf{events\_of}\ res \Rightarrow \mathsf{ok\_event}\ e\end{aligned}$$

The proof of this theorem is a simple combination of the source-level soundness theorem (Theorem 2), the two theorems about the source-level REPL program (Theorems 3 and 4), and the instantiated compiler correctness theorem (Theorem 6).

The theorem above states that any program built inside the Candle REPL can only ever export statements that are true according to the inference rules of higher-order logic.

6 Porting HOL Light scripts to Candle

Candle aims to be a verified clone of HOL Light. While the previous sections have focused on the verified part of the system, it is important to note that there is much more to HOL Light than the kernel and the basic setup of the REPL. In this section, we describe our efforts to port HOL Light's standard library to Candle.

At the time of writing, our porting efforts are still work in progress. Candle runs the majority of the scripts HOL Light executes at startup, as well as many proof scripts in the `100` and `Library` directories. Figure 6 shows a side-by-side comparison of the Candle and HOL Light REPLs.

The rest of this section describes the changes and additions we make when porting HOL Lights scripts to Candle.

6.1 Changes necessary in HOL Light scripts

With our new parser, the CakeML language supports most, but not all, of the language features HOL Light expects of its compiler. Here are the adaptations that we have made to the original HOL Light sources in order to make them compatible with Candle:

```
# g `!(x:A) y z. (x = y)
      /\ (y = z) ==> (x = z)`;;
1 subgoal (1 total)

`!x y z. x = y /\ y = z ==> x = z`

val it = (): unit
# e (REPEAT STRIP_TAC);;
1 subgoal (1 total)

  0 [`x = y`]
  1 [`y = z`]

`x = z`

val it = (): unit
# e (PURE_ASM_REWRITE_TAC []);;
1 subgoal (1 total)

  0 [`x = y`]
  1 [`y = z`]

`z = z`

val it = (): unit
# e REFL_TAC;;
No subgoals

val it = (): unit
```

```
# g `!(x:A) y z. (x = y)
      /\ (y = z) ==> (x = z)`;;
val it : goalstack = 1 subgoal (1 total)

`!x y z. x = y /\ y = z ==> x = z`

# e (REPEAT STRIP_TAC);;
val it : goalstack = 1 subgoal (1 total)

  0 [`x = y`]
  1 [`y = z`]

`x = z`

# e (PURE_ASM_REWRITE_TAC []);;
val it : goalstack = 1 subgoal (1 total)

  0 [`x = y`]
  1 [`y = z`]

`z = z`

# e REFL_TAC;;
val it : goalstack = No subgoals
```

Figure 6 Side-by-side comparison of an interactive tactic proof in Candle (left) and HOL Light (right). Here `g` sets up a new proof goal and `e` applies a given tactic to the top goal.

- The OCaml stdlib and CakeML's basis library uses different naming conventions. The effect of this on our efforts is mostly mitigated by HOL Light's own "standard library" implementation `lib.ml`. However, some functions are present in both CakeML and OCaml (e.g. `String.sub`) but with different type signatures or semantics. In such cases, we replace OCaml names with the corresponding CakeML name.
- HOL Light makes use of OCaml's polymorphic comparison operator (`Pervasives.compare`). Where possible, we have replaced all such code with concretely typed comparison operators (e.g., `Int.compare` for integers).
- HOL Light makes use of OCaml's polymorphic hash function `Hashtbl.hash` which maps arbitrary data to the integers. We don't have anything of the sort, and have to rewrite or remove this code.
- CakeML's type system imposes a value restriction. This is mostly a nuisance, but some code has to be re-structured as a consequence.
- Our parser does not deal with `let rec` forms without explicit arguments. We have to give fresh arguments to such definitions manually.
- At present, CakeML does not support `open` or `include`. We have to manually bring values into scope.
- A few files in the HOL Light basis make use of OCaml's record syntax. CakeML does not support record types at present. We omit these files in our current builds, but must either implement record types in CakeML or rework these files in order to include them.

- All special pragmas recognisable by the OCaml REPL (e.g. for installing pretty-printers) are removed. The CakeML REPL has a different way of dealing with pretty printers.
- We have made minor changes throughout HOL Light files: `hol.ml`, `system.ml`, and `lib.ml`.

6.2 Additional scripts

Here are the additions required for our REPL to be able to support HOL Light:

- Added file: `candle_pretty.ml` (Replacement for `Format`)
  We implement a functional pretty printer from a tree of pretty-printer tokens to a tree of string lists. We build an imperative interface on top of the functional printer, modelled after (a small subset of) the interface provided by the `Format` module. (250 loc.)
- Added file: `candle_nums.ml` (Replacement for `Num`)
  The `Num` library integrates both arbitrary precision integers and arbitrary precision rational numbers in one type. All integers in CakeML are arbitrary precision, and CakeML has a library for rational number arithmetic. We build a small wrapper around the CakeML integers and rationals, and provide the interface which HOL Light expects. (285 loc.)
- Added file: `candle_boot.ml` (REPL code)
  We build a read-eval-print loop (REPL) on top of the functionality provided by the CakeML compiler (see Section 4). The REPL splits user input into chunks separated by `;;`-tokens at the top-level. It supports multi-line editing, and configurable quote substitution, and a mechanism for file loading that can deal with recursive load calls.
- Added file: `candle_kernel.ml` (essentially the same as `open Kernel`)
  Our current workaround for CakeML's lack of support for `open`, as in `open Kernel`.

7 Related work

In this section, we describe related work in the area of verification of interactive theorem provers and their logics. We observe that Candle seems to be the first verified interactive theorem prover that combines: an expressive hosted logic (higher-order logic), an interactive implementation, and an end-to-end soundness theorem that reaches down to the machine code that executes the prover implementation.

7.1 Higher-order logic

Harrison [8] formalised a version of the HOL Light logic (omitting its definitional mechanisms) as well as its set-theoretic semantics, in HOL Light itself. The actual artefact being verified is a shallow-embedded implementation of HOL Light, shown to be sound with respect to the semantics. Two consistency results are proved: HOL without the axiom of infinity is shown consistent in HOL; and HOL is shown consistent in HOL extended with a larger universe of sets. However, the scope of verification does not extend past the shallow embedding; there is no formal connection between it and the actual system, which runs on interpreted OCaml and its C runtime.

Our work rests heavily on the work by Kumar et al. [9]. They build on Harrison's work and expand it along several dimensions; they formalise the definitional mechanisms omitted by Harrison [8] using contexts, and contribute a sequent calculus which is proven sound with respect to the HOL semantics. A shallow-embedded implementation is shown to refine the proof calculus. Notably, this shallow embedding can be extracted to machine code using the CakeML ecosystem, establishing a formal connection between the model-theoretic semantics and the machine code executing the kernel functions.

Gengelbach and Åman Pohjola [13, 7] further extend the work by Kumar et al. [9], adding support for ad-hoc overloading of constant definitions. This sort of mechanism is used by e.g. Isabelle/HOL to let one logical constant receive different meanings depending on what concrete types the variables in its type signature are instantiated to. At the time of writing, work on a verified cyclicity checker, which is required to ensure the soundness of instantiations of overloaded constants, has recently been completed [6]. We see no reason that our work should not build on their kernel implementation in the future.

Nipkow and Roßkopf [12] have formalised the meta-logic of Isabelle, as well as its proof-terms, and a proof checker for its proof terms. The meta-logic is used in Isabelle to define its many object logics. They formalise a proof calculus (but not a semantics) for the meta-logic in Isabelle/HOL, and implement and verify the correctness of a proof-checker for Isabelle proof-terms. Using Isabelle's (unverified) code extraction, they are able to obtain an executable checker in Standard ML. This checker can be used to check real proofs of Isabelle theorems within Isabelle, but relies on an unverified translation from Isabelle's actual proof structures into the proof-terms used by the checker. In addition, one must trust the Poly/ML compiler and its C++ runtime, which hosts both Isabelle and the checker artefact.

7.2 First-order logic

The most comprehensive ITP verification result prior to ours is Milawa by Davis and Myreen [5]. Milawa implements a quantifier-free fragment of first-order logic with recursive functions in the spirit of Nqthm and ACL2. It can execute on top of the verified Jitawa [10] Lisp runtime, and is proven sound with respect to a formal semantics. By verifying and implementing both the prover and its runtime within HOL4, the authors are able to obtain a soundness theorem which shows the soundness of the machine code that executes the Milawa system at runtime. The scope of Milawa's verification is similarly far-reaching to ours. However, it implements a fragment of first-order logic with functions, which is simpler than HOL, and relies on a Lisp runtime which is considerably less complex than the ML compilers used by LCF-style systems. An interesting feature of Milawa is its ability to bootstrap itself by successively performing conservative extensions of its own proof checker; no LCF-style system can accomplish this as far as we know.

MetaMath Zero by Carneiro [4] is a proof checker for a many-sorted first-order logic. Its logic is intended to act as a host for other object logics. A bootstrapping effort is ongoing; the proof rules of MetaMath Zero have been formalised within MetaMath Zero itself, as well as a model of the x86-64 ISA [3]. However, there is no formal connection between the formalised proof rules and any machine code refinement of said rules: the current checker implementation is unverified. As it lacks a formal semantics, verification is limited to correctness of the proof calculus, leaving out soundness. Compared to our system, MetaMath Zero is very low-level; its logic is simpler, and it offers no interactivity or proof automation, and cannot be extended at runtime. In return, one does not have to trust nor verify a complex programming language implementation. Still, the language appears practical enough to host itself and a ISA artefact [3]. Because it lacks interactivity, users do not interact directly with MetaMath Zero. Instead, they see a higher-level (unverified) ITP-like system called MM1, which produces proofs that are checked by MetaMath Zero; a design somewhat similar in spirit to that of LCF-style systems.

7.3 Dependent type theory

Barras [2] formalised the Calculus of Constructions (CC), a simpler version of the Calculus of Inductive Constructions (CIC) which is the type theory used by Coq. Barras' formalisation is done in the same spirit as Harrison's work [8], by defining a set-theoretic model of the calculus, and proving its soundness wholly inside Coq itself.

Sozeau et al. [15] present a formalisation as well as a proven-correct efficient type checker implementation for a substantial part of CIC. Their work continues the tradition of Harrison [8], Kumar et al. [9] and Barras [2], by formalising the meta-theory of Coq in Coq. The fragment of CIC under consideration omits modules and template polymorphism. An OCaml version of the verified type checker can be obtained using the unverified Coq extraction mechanism. Due to faults in the implementation of Coq's code extraction, Sozeau et al. implement and verify their own extraction mechanism, which can be used to obtain an executable checker. However, the formal verification of the extraction is done against an untyped λ-calculus, and not the actual OCaml language.

Anand and Rahli [1] have formalised the proof calculus and semantics of Nuprl in Coq, and proved soundness of the Calculus. They do not provide a verified program which implements the calculus, but their plan is extract a verified implementation from the formalisation of their calculus, and to implement and verify a type checker for a large part of Nuprl.

8 Summary

The result of our efforts is an interactive theorem prover called Candle that:

1. has been proved to be sound down to the machine code that runs it (the binary is guaranteed to only output facts that are sound w.r.t. the rules of higher-order logic);
2. offers a user experience that we have made as similar as possible to that of HOL Light (Candle supports the same syntax and interactive proof manager as HOL Light).

To the best of our knowledge, Candle is the most complete and comprehensive verified LCF-style interactive theorem prover to date.

Future work

All of the proofs about the Candle prover have been completed, but some practical challenges remain before Candle can be considered a drop-in replacement for HOL Light. Most importantly, we need to port the remainder of the HOL Light base libraries.

References

1 Abhishek Anand and Vincent Rahli. Towards a formally verified proof assistant. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving (ITP)*, volume 8558 of *LNCS*. Springer, 2014. doi:10.1007/978-3-319-08970-6_3.

2 Bruno Barras. Sets in Coq, Coq in sets. *J. Formaliz. Reason.*, 3(1), 2010. doi:10.6092/issn.1972-5787/1695.

3 Mario Carneiro. Specifying verified x86 software from scratch. *CoRR*, abs/1907.01283, 2019. arXiv:1907.01283.

4 Mario Carneiro. Metamath Zero: Designing a theorem prover prover. In Christoph Benzmüller and Bruce R. Miller, editors, *Intelligent Computer Mathematics (CICM)*, volume 12236 of *LNCS*. Springer, 2020. doi:10.1007/978-3-030-53518-6_5.

5 Jared Davis and Magnus O. Myreen. The reflective Milawa theorem prover is sound (down to the machine code that runs it). *J. Autom. Reason.*, 55(2):117–183, 2015. doi:10.1007/s10817-015-9324-6.

6 Arve Gengelbach and Johannes Åman Pohjola. A verified cyclicity checker. In *Interactive Theorem Proving (ITP)*. LIPIcs, 2022.

7 Arve Gengelbach, Johannes Åman Pohjola, and Tjark Weber. Mechanisation of model-theoretic conservative extension for HOL with ad-hoc overloading. In Claudio Sacerdoti Coen and Alwen Tiu, editors, *Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP)*, volume 332 of *EPTCS*, 2020. doi:10.4204/EPTCS.332.1.

8 John Harrison. Towards self-verification of HOL Light. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning (IJCAR)*, volume 4130 of *LNCS*. Springer, 2006. doi:10.1007/11814771_17.

9 Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. Self-formalisation of higher-order logic - semantics, soundness, and a verified implementation. *J. Autom. Reason.*, 56(3):221–259, 2016. doi:10.1007/s10817-015-9357-x.

10 Magnus O. Myreen and Jared Davis. A verified runtime for a verified theorem prover. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving (ITP)*, volume 6898 of *LNCS*. Springer, 2011. doi:10.1007/978-3-642-22863-6_20.

11 Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *J. Funct. Program.*, 24(2-3), 2014. doi:10.1017/S0956796813000282.

12 Tobias Nipkow and Simon Roßkopf. Isabelle's metalogic: Formalization and proof checker. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction (CADE)*, volume 12699 of *LNCS*. Springer, 2021. doi:10.1007/978-3-030-79876-5_6.

13 Johannes Åman Pohjola and Arve Gengelbach. A mechanised semantics for HOL with ad-hoc overloading. In Elvira Albert and Laura Kovács, editors, *Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 73 of *EPiC Series in Computing*. EasyChair, 2020. doi:10.29007/413d.

14 Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5170 of *LNCS*. Springer, 2008. doi:10.1007/978-3-540-71067-7_6.

15 Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *Proc. ACM Program. Lang.*, 4(POPL), 2020. doi:10.1145/3371076.

16 Yong Kiam Tan, Scott Owens, and Ramana Kumar. A verified type system for CakeML. In Ralf Lämmel, editor, *Implementation and Application of Functional Programming Languages (IFL)*. ACM, 2015. doi:10.1145/2897336.2897344.

Use and Abuse of Instance Parameters in the Lean Mathematical Library

Anne Baanen
Department of Computer Science, Vrije Universiteit Amsterdam, The Netherlands

Abstract

The Lean mathematical library mathlib features extensive use of the typeclass pattern for organising mathematical structures, based on Lean's mechanism of instance parameters. Related mechanisms for typeclasses are available in other provers including Agda, Coq and Isabelle with varying degrees of adoption. This paper analyses representative examples of design patterns involving instance parameters in the current Lean 3 version of mathlib, focussing on complications arising at scale and how the mathlib community deals with them.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Type theory

Keywords and phrases formalization of mathematics, dependent type theory, typeclasses, algebraic hierarchy, Lean prover

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.4

Supplementary Material *Source Code (Unabridged, Interactive Listings)*:
`https://github.com/lean-forward/mathlib-classes`
archived at `swh:1:dir:3988093240860815c0aaf3d9af663f25e8e204e1`

Funding NWO Vidi grant No. 016.Vidi.189.037, Lean Forward.

Acknowledgements I want to thank the anonymous reviewers and Jeremy Avigad, Jasmin Blanchette, Bryan Gin-Ge Chen, Johan Commelin, Sander Dahmen, Manuel Eberl, Rob Lewis, Assia Mahboubi, Filippo A. E. Nuccio, Kazuhiko Sakaguchi, Enrico Tassi and Eric Wieser for their helpful comments on earlier versions on this manuscript.

1 Introduction

An essential part of a mathematical library is a discipline for representing structures, such as an algebraic hierarchy including monoids, groups, rings and fields, or a hierarchy of spaces including topological spaces and metric spaces. Many design patterns have been proposed to enable the theory of a general structure (such as monoids) to be applied seamlessly to more specific structures (such as fields), including canonical structures in Coq [13], locales in Isabelle [17, 3], unification hints in Matita [2] and attributed types in Mizar [14].

The Lean mathematical library mathlib [26] has settled on the use of the *typeclass* [28] pattern for representing structures, implemented through Lean's mechanism of *instance parameters* [9, 10]. Typeclasses were invented by Wadler to provide ad hoc polymorphism in Haskell [28]. Similar mechanisms can now be found in programming languages including Idris [5], Rust [20, Chapter 6.11] and Scala [19], and interactive theorem provers including Agda [12], Coq [24] and Isabelle [29]. Class-based libraries of comparable complexity to mathlib have previously been developed by Hölzl, Immler and Huffman [15] for analysis in Isabelle/HOL and by Spitters and Van der Weegen [25] for an algebraic hierarchy in Coq. As of February 2022, mathlib contains over 600 classes and over 8000 instances.

In the various languages implementing a mechanism analogous to typeclasses, there is also a variety of syntax and semantics for the parts of this mechanism. In this paper, I will try to avoid confusion by using the terminology "instance parameters" when emphasising Lean 3's specific implementation, while "typeclass" refers to a design pattern that is implemented in

Lean through instance parameters. This is analogous to the distinction drawn in the Scala documentation between its mechanism of "implicit parameters" and the typeclass pattern built with that mechanism.

This paper combines my original research with a survey of the mathlib community's experience in developing a class-based hierarchy, emphasising design patterns, issues arising from the use of classes and the trade-offs available for resolving these issues. Researching how to develop and maintain a large library of mathematical structures has led me to develop a number of typeclass-based patterns in Lean that have been added to mathlib. Given that mathlib is expected to upgrade soon from Lean 3 to Lean 4 [11], an upgrade that promises changes in Lean's support for typeclasses [23], now seems a good time to discuss what is achievable with the current mechanism. I have organized this paper around a representative example for each topic, based on mathlib source code. Unabridged, interactive versions of the listings are available at `https://github.com/lean-forward/mathlib-classes`.

2 Basic instance parameters in Lean 3

Lean provides the typeclass pattern through *instance parameters*, a mechanism and implementation closely resembling the same facilities in Coq [24]. Like Coq, Lean is a dependently typed language based on the calculus of constructions. Lean has a hierarchy of universes `Sort 0 : Sort 1 : Sort 2 : ...`, where `Sort 0` is more often written as `Prop` and `Sort (u + 1)` is written `Type u` or, leaving `u` implicit, `Type*`. The bottom universe `Prop` is an impredicative type of propositions that has definitional proof irrelevance.

Let us start with the following two Lean declarations, not found in mathlib, showing the main forms that parameters can take in Lean.

```
def sub {A : Type*} [add_group A] (a b : A) : A :=
add a (neg b)

lemma sub_eq_add_neg {A : Type*} [add_group A] (a b : A) :
  sub a b = add a (neg b) := by refl
```

The round brackets mark explicit parameters to be supplied when applying the lemma, the curly brackets mark implicit parameters inferred through unification, while square brackets mark the instance parameters (for which supplying a name is optional); these are used here to specify a typeclass constraint on the type `A`. Thus the term `sub a b` specifies only the `(a b : A)` parameters to `sub`, leaving the remaining parameters to be supplied by the elaborator. These parameters are then passed on to the calls to `neg` and `add` in the body of `sub`. There is no relevant distinction between the keywords `def` and `lemma` for our purposes, apart from indicating whether the declaration exists in a `Type` or the `Prop` universe.

The elaborator supplies values to instance parameters through *instance synthesis*: the parameters to the current declaration and all declarations in the global context which have been marked with the keyword `instance` are considered in turn as candidates. Each candidate instance is type-checked against the goal, and the first candidate where the types unify is returned. For example, defining `add_group ℤ` instance allows us to subtract two integers using the `sub` operator we defined above:

```
instance : add_group ℤ := sorry -- implementation omitted
lemma subtraction_example : (sub 42 37 : ℤ) = 5 := by refl
```

Instance parameters are considered a form of implicit parameters, and can thus be made explicit using the `@` operator:

```
#check sub -- sub : ?M_1 → ?M_1 → ?M_1
#check @sub -- sub : Π {A : Type*} [add_group A], A → A → A
```

Here `?M_1` stands for a free metavariable that the elaborator could not (yet) fill in.

Instance declarations can themselves have their own instance parameters. For example, the subsets of a monoid form a monoid under pointwise operations, which we can express as

```
instance pointwise_monoid {A : Type*} [monoid A] : monoid (set A) :=
{ mul := λ X Y, { (x * y) | (x ∈ X) (y ∈ Y) },
  mul_assoc := λ _ _ _, set.image2_assoc mul_assoc,
  ..sorry /- further fields omitted -/ }
```

When the synthesis of an instance of `monoid (set A)` tries to apply the `pointwise_monoid` instance, the elaborator will recurse and try to synthesize the `monoid A` dependency; if this instance is not found, search backtracks and continues with the next `monoid (set A)` instance; if the entire search tree is exhausted, synthesis has failed. Since instances' types are unified with the goal, we can view the synthesis algorithm as performing recursion on the term structure of the goal.

2.1 Class definitions

The classes themselves are expressed as records, i.e. dependent tuples, with the class fields as projections taking instance parameters. Classes use the same syntax as record types in Lean, only differing in using the keyword `class` instead of `structure`. Dependent types mean data- and proof-carrying fields are expressed in the same way; Section 7 discusses some usage differences between data and proofs. Thus, a definition for `add_group` could look like:

```
class add_group (A : Type*) : Type* :=
(zero : A) (neg : A → A) (add : A → A → A)
(add_assoc : ∀ (x y z : A), add x (add y z) = add (add x y) z)
(zero_add : ∀ (x : A), add zero x = x)
(neg_add : ∀ (x : A), add (neg x) x = zero)
```

The projections of a class automatically use instance parameters, generating the declarations:

```
def add_group.zero {A : Type*} [h : add_group A] : A := h.1
def add_group.neg {A : Type*} [h : add_group A] : A → A := h.2
def add_group.add {A : Type*} [h : add_group A] : A → A → A := h.3

def add_group.add_assoc {A : Type*} [h : add_group A] :
  ∀ (x y z : A), add_group.add x (add_group.add y z) =
    add_group.add (add_group.add x y) z) := h.4
-- and so on.
```

The instance synthesis algorithm also allows instances for non-record types. In practice most classes in mathlib are record types and indeed Lean 4 will make the use of record types for classes obligatory to simplify the elaborator.

2.2 Subclassing

The mechanisms described above result in two patterns for subclass definitions, with important distinctions in semantics. *Unbundled subclasses* take superclasses as instance parameters to the class declaration. To define abelian groups as a subclass of additive groups, we write

```
class add_comm_group (A : Type*) [add_group A] : Type* :=
(add_comm : ∀ (x y : A), add x y = add y x)
```

Elaboration of the expression `add_comm_group A`, e.g. in a parameter `[add_comm_group A]`, requires the synthesis of an `add_group A` instance occurring as parameter to `add_comm_group`. We make this instance available by adding it as another instance parameter, so all results on abelian groups take the two instance parameters `[add_group A] [add_comm_group A]`; long inheritance chains cause long parameter lists. Similarly, declaring an `add_comm_group A` instance requires a previous declaration of an `add_group A` instance.

In contrast, *bundled subclasses* make use of instance synthesis to access the superclass, only requiring an instance parameter for the subclass. A bundled subclass contains an instance of its superclass as a record field. Lean's `extends` keyword provides syntax sugar for the construction:

```
class add_comm_group (A : Type*) extends add_group A :=
(add_comm : ∀ (x y : A), add x y = add y x)
```

This has analogous effects to writing

```
class add_comm_group (A : Type*) : Type* :=
(to_add_group : add_group A)
(add_comm : ∀ (x y : A),
  @add_group.add A to_add_group x y = @add_group.add A to_add_group y x)
-- Register the projection as an instance:
attribute [instance] add_comm_group.to_add_group
```

When we look at the synthesis of an `add_group A` instance, we can see the instance `add_comm_group.to_add_group` triggers a recursive search for an `add_comm_group A` instance; in this way subclass instances automatically provide access to declarations on the superclass. Both subclass patterns are used in mathlib; the following sections discuss reasons for choosing one over the other in a given situation.

2.3 Extensions of the typeclass pattern

Beyond expressing the basic typeclass patterns above, Lean's instance parameters provide a considerable amount of flexibility. Classes do not have to be parametrized over exactly one type, unlike what the phrase "typeclass" suggests, and the only restriction on `instance` declarations is that the head symbol of its return type is declared to be a class. In Haskell terminology, all the following extensions are allowed: constrained class method types, flexible contexts, flexible instances, incoherent instances, multi-parameter classes (including nullary classes), overlapping instances, quantified constraints, type synonym instances, undecidable instances. Most of these extensions find uses throughout mathlib.

A notable difference compared with Agda is that Lean allows overlapping instances, thus enabling the diamond inheritance pattern of Section 4. Compared with Isabelle, Lean permits multi-parameter classes, as we will see in Section 5; on the other hand Isabelle provides multi-parameter hierarchies through locales [3]. Compared with Coq, Lean adds a limited syntax for expressing functional dependencies, also discussed in Section 5.

Since Lean is a dependently typed language, parameters to classes are not restricted to types. For example, mathlib uses a typeclass parametrized over a natural number p to express the characteristic of a ring:

```
class char_p (R : Type*) [semiring R] (p : ℕ) : Prop :=
(cast_eq_zero_iff : ∀ (x : ℕ), (coe x : R) = 0 ↔ p ∣ x)
```

The `char_p` class is an example of *a proof mixin*. Section 8 discusses this pattern further.

Since type-checking is used to match candidate instances with a synthesis goal, the synthesis algorithm works up to definitional equality. For example, since `2+2 : ℕ` is definitionally equal to `4`, Lean finds the instance `zmod.char_p 4 : char_p (zmod 4) 4` for the goal `char_p (zmod 4) (2 + 2)`. Thus, typeclasses in Lean are coupled to a built-in syntactic notion of equality.

This combination of features means that instance parameters can be used for small-scale automation, since the instance synthesis mechanism provides a search tactic for definite Horn clauses: a clause of the form $C = (\bigwedge_i P_i(\vec{t})) \to Q(\vec{t})$, where the P_i and Q are (not necessarily distinct) predicates and $\vec{t}$ a vector of terms, translates to an instance of the form `instance C [P_1 t_1 ... t_k] ... [P_n t_1 ... t_k] : Q t_1 ... t_k`. The branching depth-first nature of the synthesis algorithm has to be kept in mind during design in order to assure acceptable performance, as we will see in Section 10.

3 `has_mul`: notation typeclass

The typeclass pattern is used throughout mathlib for operator overloading, in much the same role that classes were originally introduced in Haskell. Generally, such a notation typeclass has one type parameter `α : Type*` and contains fields which carry only data. A basic example is the definition of the multiplication operator `*` in core Lean:

```
class has_mul (α : Type*) := (mul : α → α → α)
infix * := has_mul.mul
```

The `infix` command adds the notation `a * b` for `has_mul.mul a b`.

These notations are not directly coupled to the algebraic hierarchy: the `has_inv` class providing $^{-1}$ notation for the multiplicative inverse does not have any fields requiring a multiplicative group structure. However, in practice such notations are often provided through inheritance from an instance of a proof-carrying class in the algebraic hierarchy.

Lean uses classes to implement implicit coercions in the style of Saïbi [21]. Whenever the elaborator encounters a term `t : A` that is instead expected to have type `B`, it replaces `t` with `@coe A B _ t`, where the `_` marks an instance parameter of type `has_lift_t A B`. Similarly, when a term `f : F` produces a type error because it is expected to have a dependent function type, it is replaced with `coe_fn f` (where `coe_fn {F A} [has_coe_to_fun F A]` has type `Π (f : F), A f`), and when `t` is expected to be of the form `Sort u` (that is, either `Type v` if `u = v+1`, or `Prop` if `u = 0`), it is replaced with `coe_sort t` (where `coe_sort {A} [has_coe_to_sort A] : Sort u`). Such coercions are essential for mathlib's design of morphisms and subobjects, as we will see in Section 6.

4 `comm_monoid`: algebraic hierarchy class

The algebraic hierarchy in mathlib is built using typeclasses, based on the notation typeclasses discussed in the previous section. Similar class-based hierarchies exist in mathlib for topics including orders, topology and analysis, and all the hierarchies are connected throughout. As an example, the `comm_monoid` typeclass is implemented in mathlib essentially as follows:

```
set_option old_structure_cmd true -- explained below

class semigroup (G : Type*) extends has_mul G :=
(mul_assoc : ∀ a b c : G, a * b * c = a * (b * c))

class mul_one_class (M : Type*) extends has_one M, has_mul M :=
(one_mul : ∀ a : M, 1 * a = a) (mul_one : ∀ a : M, a * 1 = a)

class comm_semigroup (G : Type*) extends semigroup G :=
(mul_comm : ∀ a b : G, a * b = b * a)

class monoid (M : Type*) extends semigroup M, mul_one_class M

class comm_monoid (M : Type*) extends monoid M, comm_semigroup M
```

While `comm_monoid` is considered to sit low in the mathlib algebraic hierarchy, its definition already depends on seven ancestor classes in a complicated diamond inheritance pattern. Multiple inheritance paths result in two instances of `has_mul` for each `monoid` instance, thus requiring support for overlapping instances. We can also see that mathlib prefers bundled inheritance in the algebraic hierarchy, incorporating ancestor classes' fields rather than taking superclasses as instance parameters. This choice is further explained in Section 10.

The various hierarchies in mathlib are interwoven through multiple inheritance. Thus the hierarchy of order structures such as partial orders, linear orders and lattices (extending the notation typeclasses `has_le` providing $(\leq)$ and `has_lt` providing $(<)$), is combined with the algebraic hierarchy into a hierarchy of ordered algebraic structures from partially ordered commutative monoids up to linearly ordered fields.

The first line `set_option old_structure_cmd true` switches between two representations of ancestors for `structure` and `class` declarations: under the default, "new" structure behaviour, `monoid M` would contain two fields, of type `semigroup M` and `mul_one_class M`, each of which carries its own distinct `has_mul` field. Thus the `mul_assoc` field inherited from `semigroup` would refer to a multiplication operation other than the multiplication of `one_mul` inherited from `mul_one_class`; the resulting class with two binary operators would not actually specify monoids. Indeed, Lean will detect such ambiguities and produce an error if a "new" structure inherits conflicting field names.

The "old" structure behaviour avoids this issue by copying all fields from the ancestor structure into the child structure, skipping duplicates, so that `monoid` only has one `mul` field. Compare the following two desugarings of `extends`:

```
class monoid_new (M : Type*) :=
(to_semigroup : semigroup M)
(to_mul_one_class : mul_one_class M)

class monoid_old (M : Type*) :=
(mul : M → M → M) (mul_assoc : ∀ a b c : M, a * b * c = a * (b * c))
(one : M) (one_mul : ∀ a : M, 1 * a = a) (mul_one : ∀ a : M, a * 1 = a)
```

Lean 4 only implements the "new" structure command since it directly allows projecting to ancestor structures, adding support for diamond inheritance through automatically inheriting from the common ancestor and copying the remaining fields.

In the terminology of Coq's Hierarchy Builder [7], the typeclasses are specified in terms of *mixins*: the packages of operations and properties available for a given structure. Like Hierarchy Builder provides for mixins, projections from a subclass to its immediate superclasses are automatically generated as instances. There is no explicit concept in Lean corresponding to Hierarchy Builder's *factories* or *builders*. To manually construct a subclass instance given a superclass or project a subclass into a superclass, users can apply the notation `.. s`, which extends a constructor's argument list by copying the relevant fields out of a tuple `s`.

In general, mathlib's hierarchy is extended when the mathematics requires it, so there are many parts of the hierarchy that do not form a boolean algebra. Thus there is no `comm_mul_one_class` forming the direct subclass of `comm_semigroup` and `mul_one_class`, nor is there a `comm_mul_class` that provides the `mul_comm` field by itself. Adding an intermediate class to the hierarchy is a straightforward process of moving over the fields and modifying the `extends` clauses, as recently happened with the addition of `mul_one_class`.

The relative ease of modification means the hierarchy does not need to be designed up front for all potential usages. This stands in contrast to the situation for packed classes, where refactoring the hierarchy involves a deep understanding of the details involved or the usage of a tool such as Hierarchy Builder, to ensure consistency such as the uniqueness of a join for any two structures applied to the same type [22]. Careful design is still needed for instances to avoid certain cases of drastic slowdowns, as seen in Section 10.

In addition to the above multiplicative hierarchy, mathlib includes an isomorphic additive hierarchy differing only in notation: the definition of `add_monoid` renames `(*)` and `1` to `(+)` and `0`. A metaprogram `to_additive` creates an appropriately renamed duplicate additive version of declarations. The duplicate notation is required for the definition of the `semiring` class in mathlib, since bundled class inheritance cannot express the fact that the additive and multiplicative structures of a semiring both form a monoid. In comparison to mathlib's ad hoc solution, Isabelle's locales support different notations automatically, since the operations of a target context can be renamed in sublocale declarations or instantiations [4].

5 `module`: multi-parameter classes

In algebra, a (*left semi-*)*module* is an additive commutative monoid `M` that is acted on by a semiring `R` through scalar multiplication, satisfying certain axioms; the concept generalizes vector spaces by replacing the field of scalars by an arbitrary semiring. Modules are available in the mathlib algebraic hierarchy in full generality as a multi-parameter typeclass depending on both `R` and `M`. Following the pattern of monoids, the base class introduces notation, and is subclassed to add the axioms of the structure:

```
class has_scalar (α β : Type*) : Type* :=
(smul : α → β → β)
infix · := has_scalar.smul

class mul_action (M A : Type*) [monoid M] extends has_scalar M A :=
(one_smul : ∀ (x : A), (1 : M) · x = x)
(mul_smul : ∀ (r s : M) (x : A), (r * s) · x = r · (s · x))

class distrib_mul_action (M A : Type*) [monoid M] [add_monoid A]
  extends mul_action M A :=
(smul_add : ∀ (r : M) (x y : A), r · (x + y) = r · x + r · y)
(smul_zero : ∀ (r : M), r · (0 : A) = 0)
```

```
class module (R M : Type*) [semiring R] [add_comm_monoid M]
  extends distrib_mul_action R M :=
(add_smul : ∀ (r s : R) (x : M), (r + s) · x = r · x + s · x)
(zero_smul : ∀ (x : M), (0 : R) · x = 0)
```

Compare this to the class-based analysis library in Isabelle/HOL, where the absence of multi-parameter classes means only real numbers appear as scalars [15]; Isabelle instead provides multi-parameter locales [3].

Vector spaces do not have a separate definition in mathlib since they only replace the ring axioms on the scalars with field axioms, while the fields of the `module` class are unchanged. Instead, a `K`-vector space `V` is denoted through parameters `[field K] [add_comm_group V] [module K V]`. In order to make vector spaces more discoverable for users, the mathlib community has been discussing a system of parameter-level abbreviations, so that `[vector_space K V]` expands into `[field K] [add_comm_group V] [module K V]`.

5.1 Dangerous instances

We see here that the `module` hierarchy uses a mix of bundled and unbundled inheritance, unlike `comm_monoid` which solely uses bundled inheritance. This follows the rule of bundling only if the superclass has a superset of the subclass's parameters; otherwise the generated instance would be a *dangerous instance* where some parameters are undetermined. Namely, declaring that `module R M extends add_comm_monoid M` would generate the following instance:

```
instance module.to_add_comm_monoid {R M : Type*} [module R M] :
  add_comm_monoid M := sorry
```

Now instance synthesis for `add_comm_monoid M` will lead to a search for `module ?M_1 M`, where `?M_1` is a free metavariable. Since unification in the elaborator can call instance synthesis, without backtracking, finding the wrong instance for an underspecified goal may cause unification to fail where another instance would have worked.

Such dangerous instances with free variables in their constraints can be remedied in various ways. If, as above, the instance derives from a subclass constraint involving the `extends` keyword, the constraint is instead expressed through an instance parameter on the subclass; this implies no dangerous instance is generated to express inheritance. The main drawback is that this mix of unbundled and bundled inheritance is more confusing and less natural than the approach used in the MathComp library, where canonical structures allow bundling the additive monoid structure and the R-module structure [18].

If the free parameter is uniquely determined by the choice of the bound parameters, we can register this functional dependency with the `out_param` construction. Lean assigns all parameters as in-parameters, unless explicitly marked as `out_param`, in contrast to the automatic determination of the direction in Coq. For example, we can make `R` a functional dependency of `M` by instead defining:

```
class module (R : out_param Type*) (M : Type*) [semiring R] := -- etc.
```

The elaborator replaces all out parameters in the synthesis goal with a free metavariable, which is filled by unifying the goal with the type of the candidate instance. For `module` this functional dependency is not acceptable since each `add_comm_monoid M` has an instance `add_comm_monoid.nat_module : module ℕ M` reflecting the natural $\mathbb{N}$-module structure (see also Section 7), which would be incompatible with an R-module structure for other

semirings R. Moreover, the instance `add_comm_monoid.nat_module` provides a second reason that bundled inheritance is unsuitable for the subclass relation of `add_comm_monoid` and `module`: it would form a loop with the instance `module.to_add_comm_monoid`.

A final way to resolve dangerous instances is to remove the `instance` keyword so that it does not participate in synthesis. mathlib takes this approach when stating the theorem that any module over a ring has additive inverses:

```
def module.add_comm_monoid_to_add_comm_group (R M : Type*)
  [ring R] [add_comm_monoid M] [module R M] :
  add_comm_group M := sorry -- proof omitted
```

To provide `add_comm_group` instances when `R` is known, we can still make use of the instance `add_comm_monoid_to_add_comm_group` in a separate `instance` declaration.

6 `monoid_hom_class`: generic bundled morphisms

The representation of morphisms such as group homomorphisms or linear maps has changed repeatedly in mathlib, is still not unified and is still undergoing refactors. The main issue complicating the design is the trade-off between generality and ease of inference. The author of this paper has designed a pattern providing bundled morphisms with some of the advantages lost during the move from unbundled morphisms, by making theorems generic over types of morphisms. The same pattern works for subobjects, replacing "morphism" with "subobject" and "a map preserving an operation" with "a set closed under an operation".

6.1 Unbundled morphisms

The original design of algebraic homomorphisms in mathlib did not bundle maps in the same structure as their properties, allowing any function `f : R → S` to be used as a ring homomorphism if an `is_monoid_hom f` instance was available. The `is_ring_hom` predicate stated `f` preserves the ring operations `*`, `+`, `1` and `0`. Instances were available for the common operations, except composition:

```
class is_monoid_hom {M N : Type*} [monoid M] [monoid N] (f : M → N) :
    Prop :=
(map_mul : ∀ x y : M, f (x * y) = f x * f y)
(map_one : f 1 = 1)

class is_ring_hom {R S : Type*} [semiring R] [semiring S] (f : R → S)
  extends is_monoid_hom f :=
(map_add : ∀ x y : R, f (x + y) = f x + f y)
(map_zero : f 0 = 0)

instance id.is_ring_hom (R : Type*) [semiring R] :
  is_ring_hom (id : R → R) := sorry -- details omitted

lemma comp.is_ring_hom {R S T : Type*} (f : R → S) (g : S → T)
  [semiring R] [semiring S] [semiring T] [is_ring_hom f] [is_ring_hom g] :
  is_ring_hom (g ∘ f) := sorry -- details omitted
```

Synthesis for the `is_ring_hom` class struggles with the resulting higher-order matching problems. In particular, there is no instance for composition since matching `is_ring_hom (?_g ∘ ?_f)` with a goal `is_ring_hom f` would result in setting `?_f` to `f` and `?_g` to the identity function `id`. Thus, making `comp.is_ring_hom` would lead to instance synthesis diverging along the path `is_ring_hom f` → `is_ring_hom (id ∘ f)` → `is_ring_hom (id ∘ id ∘ f)` → ⋯. In the formalization of Witt vectors, these issues led Commelin and Lewis to avoid classes and instead use a custom metaprogram for generating instances of their `is_poly` predicate [8].

Apart from the inability of instances on compositions to be synthesised, under this design rewriting tactics such as the simplifier cannot easily iterate over all subterms where the `map_mul` lemma can be applied: since every subterm of any term can potentially unify with a function application (such as a constant function), any subterm would cause an instance search. Finally, the collection of morphisms could not be as easily treated as an object in its own right, for example to put a group structure on the automorphisms of a field [26].

6.2 Bundled morphisms

For these reasons, mathlib was refactored to prefer bundled morphisms:

```
structure monoid_hom (M N : Type*) [monoid M] [monoid N] :=
(to_fun : M → N)
(map_mul : ∀ x y, to_fun (x * y) = to_fun x * to_fun y)
(map_one : to_fun 1 = 1)

structure ring_hom (R S : Type*) [semiring R] [semiring S]
  extends monoid_hom R S :=
(map_add : ∀ x y, to_fun (x + y) = to_fun x + to_fun y)
(map_zero : to_fun 0 = 0)

instance monoid_hom.has_coe_to_fun (M N : Type*) [monoid M] [monoid N] :
  has_coe_to_fun (monoid_hom M N) (λ _, M → N) :=
{ coe := monoid_hom.to_fun }

def monoid_hom.id (M : Type*) [monoid M] : monoid_hom M M :=
{ to_fun := id, .. } -- details omitted

def monoid_hom.comp {M N O : Type*} [monoid M] [monoid N] [monoid O]
  (f : monoid_hom M N) (g : monoid_hom N O) : monoid_hom M O :=
{ to_fun := g ∘ f, .. } -- details omitted
```

Lean uses the `has_coe_to_fun` instance to parse `(f : monoid_hom M N) x` as `(@coe_fn _ _ (monoid_hom.has_coe_to_fun M N) f : M → N) x`. Further examples of bundled morphisms available in mathlib include ring homomorphisms, linear maps, monotone functions (order homomorphisms) and the bijective versions of the above: group, ring and order isomorphisms and linear equivalences.

Bundled morphisms do not suffer from the composition, simplification and structure issues, at the cost of all morphisms needing to be declared as such ahead of time or needing lemmas to convert between bundled and unbundled forms. This is a drawback especially when the unbundled form has convenient notation, such as the additive group endomorphism of a ring given by multiplying by a constant `c`:

```
instance mul.is_add_monoid_hom {R : Type*} [ring R] (c : R) :
  is_add_monoid_hom ((*) c) := sorry -- details omitted

def add_monoid_hom.mul_left {R : Type*} [ring R] (c : R) :
  add_monoid_hom R R := { to_fun := (*) c, ..sorry } -- details omitted
```

In addition, it is no longer possible to use `monoid_hom` lemmas for a `ring_hom`: since `monoid_hom` and `ring_hom` are two different bundled types, ring homomorphisms can be viewed as monoid homomorphisms only through (manually) inserting coercions. Although the coercion could be supplied in some cases using unification hints, the support for unification hints in Lean 3 was not sufficient to do this in every case, and in anticipation of Lean 4's new unification system, unification hints were entirely removed from Lean 3 and mathlib.

Instead, to gain fully automatic simplification, all `monoid_hom` lemmas had to be copied over to `ring_hom` and all other structures extending `monoid_hom`. Thus mathlib ended up with many copies of lemmas such as `map_prod`:

```
lemma monoid_hom.map_prod (g : monoid_hom M N) :
  g (∏ i in s, f i) = ∏ i in s, g (f i) := sorry -- proof omitted

lemma ring_hom.map_prod (g : ring_hom R S) :
  g (∏ i in s, f i) = ∏ i in s, g (f i) :=
monoid_hom.map_prod s f g.to_monoid_hom

lemma mul_equiv.map_prod ...
lemma ring_equiv.map_prod ...
lemma alg_hom.map_prod ...
lemma alg_equiv.map_prod ...
```

This duplication is further multiplied by the amount of monoid operators in mathlib: a corresponding version of each `map_prod` lemma also exists for the product of a multiset and for the product of a list. Furthermore, monoid homomorphisms preserve multiplicative inverses, powers of elements, divisibility, nth roots, and so on. The end result is that the full set of lemmas grows proportionally to the number of structures extending `monoid_hom` times the number of operations preserved by a `monoid_hom`.

This copying happened manually and typically on an ad hoc basis, so that mathlib contributors often encountered lemmas that were missing for their specific choice of morphism, needing to switch contexts and add these mathematically trivial lemmas back in by hand, waiting for the dependencies to recompile before being able to continue with their proof. To address these shortcomings, the mathlib community on initiative of the author of this paper, switched to a third design pattern that automates the derivation of lemmas when a morphism type is extended.

6.3 Morphism classes

The cause of this duplication is that the pattern of "bundled morphism" was applied informally, with no unifying programmatic interface. The key insight was to follow object-oriented practice of programming to an interface rather than a concrete class, or in Lean terms: to program to a typeclass `monoid_hom_class` rather than a concrete type such as `monoid_hom`.

The first step in introducing this interface was a typeclass `fun_like` for *types* of bundled (dependent) functions, based on Eric Wieser's `set_like` class for types of bundled subobjects.[1]

```
class has_coe_to_fun (F : Type*) (α : out_param (F → Type*)) :=
(coe : Π x : F, α x)

class fun_like (F : Type*)
  (α : out_param Type*) (β : out_param (α → Type*))
  extends has_coe_to_fun F (λ _, Π a : α, β a) :=
(coe_injective' : function.injective coe)

-- A typical instance looks like:
instance monoid_hom.fun_like : fun_like (monoid_hom M N) M (λ _, N) :=
{ coe := monoid_hom.to_fun,
  coe_injective' := λ f g h, by { cases f, cases g, congr' } }
```

After defining the instance `monoid_hom.fun_like`, instance synthesis provides function application syntax, extensionality and congruence lemmas for monoid homomorphisms.

The next step in addressing the duplication is to introduce a class for the bundled morphism types that coerce to `monoid_hom`:

```
class monoid_hom_class (F : Type*) (M N : out_param Type*)
  [monoid M] [monoid N] extends fun_like F M (λ _, N) :=
(map_one : ∀ (f : F), f 1 = 1)
(map_mul : ∀ (f : F) (x y : M), f (x * y) = f x * f y)

instance : monoid_hom_class (monoid_hom M N) M N :=
sorry -- details omitted
```

Note the difference between `is_monoid_hom f` and `monoid_hom_class F M N`: the former is a predicate on *morphisms*, the latter is a predicate on *types of morphisms*.

It is necessary to fully apply the morphism types before they can be used as a parameter to `monoid_hom_class`: since `monoid_hom` and `ring_hom` have different instance parameters, we are not able to write both `monoid_hom_class monoid_hom` and `monoid_hom_class ring_hom` type-correctly. This means the class requires parameters `M N`, which are `out_param`s so that the lemma application `map_one f` can leave these parameters implicit.

The types such as `ring_hom` extending `monoid_hom` should receive a `monoid_hom_class` instance, which we can do by subclassing `monoid_hom_class` and instantiating the subclass:

```
class ring_hom_class (F : Type*) (R S : out_param Type*)
  [semiring R] [semiring S]
  extends monoid_hom_class F R S :=
(map_zero : ∀ (f : F), f 0 = 0)
(map_add : ∀ (f : F) (x y : R), f (x + y) = f x + f y)

instance : ring_hom_class (ring_hom R S) R S := sorry -- details omitted
```

[1] https://github.com/leanprover-community/mathlib/pull/6768

Now lemmas can be made generic by parametrizing over all the types of bundled morphisms, reducing the multiplicative amount of lemmas to an additive amount: each extension of `monoid_hom` should get a `monoid_hom_class` instance, and each operation preserved by `monoid_hom`s should get a lemma taking a `monoid_hom_class` parameter.

```
lemma map_prod {G : Type*} [monoid_hom_class G M N] (g : G) :
  g (∏ i in s, f i) = ∏ i in s, g (f i) := sorry -- proof omitted
```

This design pattern has been applied in mathlib to morphisms (implemented as subclasses of `fun_like`) and subobjects (implemented as subclasses of `set_like`). The mathlib community has welcomed the morphism class design for reducing the amounts of duplication, manual work and missing lemmas, although not all usages have switched to the generic lemmas, and work is still ongoing to provide a suitable generic form of standard operations such as composition and identity maps.

7 `nsmul`: ensuring equality of instances

Each `add_comm_monoid M` structure naturally gives rise to an $\mathbb{N}$-module structure, where `n • x` is defined as `x + x + ⋯ + x`, `n` times. In addition, each `semiring R` structure naturally gives rise to an `R`-module structure on itself, where `x • y` is defined as `x * y`. These two actions are available in mathlib as instances `add_comm_monoid.nat_module` and `semiring.to_module` respectively. Note that setting `M = R = ℕ` results in two instances for `module ℕ ℕ`. The existence of multiple instances of the same type does not necessarily lead to problems in Lean. Indeed, diamond inheritance in the mathlib algebraic hierarchy exploits this possibility. Problems arise when the two instances are not definitionally equal, in cases such as a goal containing `add_comm_monoid.nat_module` in which we want to apply a lemma containing `semiring.to_module`. As an extra complication, the two instances result in the same syntax `n • k`, making incompatibilities hard to spot.

To resolve such issues, first we could ensure only one instance is found, for example by replacing the other instance with a `def` that is not considered during instance synthesis. However, both described above are mathematically useful in their respective context, and only cause an issue when this context overlaps, namely for the natural numbers. Modifying the order in which instances are considered will not work, since one instance is not merely a generalization of the other: when combining a lemma on `add_comm_monoid`s with a lemma on `semiring`s, both instances will still appear no matter the instance priorities.

When overlapping instances are required, the mathlib community ensures these are definitionally equal for all possible instantiations in the overlap. Note that Lean's implementation of diamond inheritance automatically provides definitional equality of all inheritance paths.

An advantage of the `Prop`-valued mixin classes discussed in Section 8 is that all instances are equal by proof irrelevance. For example, the mathlib community is considering replacing the data-carrying class `fintype (α : Type*) : Type*` containing a finite enumeration of the elements of a given type, with a proof-only class `finite (α : Type*) : Prop` non-constructively asserting the existence of an enumeration. Although `fintype α` is designed to be a subsingleton for all α, it is only a subsingleton up to propositional equality, meaning two different enumerations would still lead to unification issues. On the other hand, `Prop`-valued classes cannot be applied everywhere: the absence of data means it is incompatible with classes that provide notation such as scalar multiplication, and it is in general incompatible with intuitionistic logic. The class `decidable_pred {α : Type*} (p : α → Prop) : Type*` provides a decision algorithm for `p`, and is used in mathlib

for small numeric computations. While we could define this to be `Prop`-valued by setting `decidable_pred p := ∀ x, p x ∨ ¬ (p x)`, that would make it useless for actually performing this decision algorithm.

To make the two `module ℕ ℕ` instances definitionally equal, we ensure data-carrying fields of these instances are definitionally equal, using proof irrelevance for the proof-carrying fields [30]. In particular, the `smul` field of `add_comm_monoid.module` needs to be defined so that instantiated for ℕ, it equals multiplication on natural numbers `nat.mul`. While we could redefine `nat.mul` to be recursive on the left argument to match the action of left modules, this would violate the requirements of right modules, where multiplication by natural numbers must be right-recursive.

Instead, mathlib adds extra data to `add_comm_monoid`'s ancestor `add_monoid`: a field `nsmul : ℕ → M → M` defines scalar multiplication by a natural number, and two proof fields assert it (propositionally) equals the left-recursive definition:

```
class add_monoid (M : Type*) extends add_semigroup M, add_zero_class M :=
(nsmul : ℕ → M → M)
(nsmul_zero : ∀ x, nsmul 0 x = 0)
(nsmul_succ : ∀ (n : ℕ) x, nsmul (n + 1) x = x + nsmul n x)
```

The `nsmul` field can be set to the usual $*$ operator for `add_comm_monoid ℕ`, and a generic implementation `nsmul_rec {M : Type*} [has_zero M] [has_add M] : ℕ → M → M` is provided for instances where definitional equality is not a concern.

The same principle of providing a field for all definitional equalities generalizes the principle of *forgetful inheritance* [1] known also in Coq and Isabelle, that the instance creating a superclass from a subclass can only consist of projecting away fields. This rule is illustrated in mathlib by the class `metric_space` which extends `topological_space` [6, 26].

There is currently no mechanism available in mathlib for automatically detecting or resolving issues with definitional equality of instances. A linter [27] that warns for diamond issues would already be a useful improvement over the status quo of manual investigation. Even better would be a mechanism that can canonicalize instances of propositionally subsingleton classes to ensure equality also holds definitionally.

8 `unique`: proof-carrying mixin

The mathlib algebraic hierarchy is *semi-bundled*, meaning all operations and properties are passed in a single instance parameter. In contrast, mathlib also provides a large collection of mixins that can be added as separate instance parameters. For example, `subsingleton : Π (α : Type*), Prop` asserts the type α has at most one element. The subclass `unique` α of `subsingleton` α (constructively) asserts that α has exactly one element. This means `unique` α is also a subclass of `inhabited` α, which (constructively) specifies an element of α while also allowing for more. A theorem about trivial monoids will take these assumptions as separate parameters `[monoid M] [subsingleton M]`:

```
instance [monoid M] [subsingleton M] : unique (units M) :=
sorry -- proof omitted
```

In fact, `unique` α is equivalent to the conjunction of `subsingleton` α and `inhabited` α. However, the implication `∀ {α}, subsingleton` $\alpha \to$ `inhabited` $\alpha \to$ `unique` α cannot be added while keeping `subsingleton` and `inhabited` superclasses of `unique`, since that would result in an infinite loop `unique` $\to$ `subsingleton` $\to$ `unique` $\to$ `subsingleton` $\to \cdots$ during instance synthesis. The tabled instance synthesis procedure in Lean 4 will

ensure searches are performed only once per syntactically equal subgoal, resolving this specific issue [23]. The current version of mathlib still uses such conjunction classes even though instances cannot be automatically synthesized from conjuncts. Preferring a single instance parameter improves performance by reducing term size, as we will discuss in Section 10.

9 `fact`: interfacing between instances and non-instances

Suppose we want to create an instance reflecting the fact that $\mathbb{Z}/n\mathbb{Z}$ is a field if n is a prime number. This instance will take a number `n : ℕ` and a proof showing `n` is prime, and return a `field (zmod n)` instance. Given `n`, a proof that `n` is prime cannot be inferred through unification, so to make the instance synthesizable, the proof of primality must appear as an instance parameter. Thus, we could define a class `nat.prime n` asserting `n : ℕ` is a prime number, and take an instance of this class as a parameter of the `zmod.field` instance:

```
class nat.prime (n : ℕ) : Prop :=
(nontrivial : 2 ≤ n) (only_two_divisors : ∀ m | n, m = 1 ∨ m = n)

def zmod : ℕ → Type
| 0     := ℤ
| (n+1) := fin (n+1)

instance zmod.field (n : ℕ) [nat.prime n] : field (zmod n) :=
sorry -- details omitted
```

Unfortunately, instances for `nat.prime` do not work well in their own right: it is impractical to check that a term `n` is a prime number by recursion on the term structure of `n`. In particular, `n` may contain free variables or be too large to reduce to a unary numeral. Splitting between a predicate `def nat.prime` whose proofs are passed as explicit parameters and the same predicate as a class declaration `class nat.prime_class` whose proofs are passed as instance parameters is not satisfying either, due to the large amount of duplication this would entail.

Instead mathlib provides a mechanism for ad hoc typeclass creation, by supplying a proposition to the `fact` class:

```
def nat.prime (n : ℕ) : Prop := 2 ≤ n ∧ (∀ m | n, m = 1 ∨ m = n)

class fact (p : Prop) : Prop := (out : p)

instance zmod.field (n : ℕ) [fact (nat.prime n)] : field (zmod n) :=
sorry -- details omitted
```

In a similar way, the `fact` class is used for the assumption `x < y` when showing that the interval $[x, y] \subset \mathbb{R}$ is a manifold with boundary, to provide the assumption that a polynomial f splits in a field K when defining the natural inclusion of the splitting field of f into K, and to provide non-negativity or positivity assumptions in various contexts.

Along with the ad hoc class pattern provided by `fact`, there is an ad hoc instance pattern provided by the tactic `letI`. Instance synthesis considers declarations marked as `instance` and parameters to the current declaration, caching these before elaborating the type and body of the declaration. The `letI` tactic inserts new instances into this cache, providing this instance in the current proof context.

In addition, `letI` can resolve the dangerous instance issue of Section 5.1 in some cases: in a proof context where the ring of scalars `R` remains fixed, we can use `letI` to safely make `module.add_comm_monoid_to_add_comm_group` an instance within this context.

10 Performance and bundling

The pervasive use of typeclasses in mathlib means instance synthesis accounts for 10 to 25 percent of the build time of a typical mathlib file. Beyond the time taken for synthesis, the use of typeclasses has performance impacts on the entirety of the compilation process. For example, typeclasses tend to produce larger terms compared to those generated by canonical structure: Another factor complicating direct comparison with other mechanisms is the trade-off between upfront and repeating costs. For instance, although activating a locale in Isabelle is rather costly since it requires processing all declarations in the locale's dependency graph [3], declarations can be structured such that this cost only needs to be paid once for a group of declarations; instance synthesis is performed for each instance parameter in each term. Performance of typeclasses in Lean remains acceptable, though, thanks to instance caching, Lean's efficient synthesis implementation in C++ and mathlib's design patterns.

First of all, the mathlib algebraic hierarchy avoids unbundled subclasses that express superclass constraints through instance parameters, since these lead to exponential blowup of term sizes. An example of exponential blowup is discussed in detail in Ralf Jung's blog post [16]. This example concerns product type instances of an unbundled class such as the following modification to the `comm_monoid` class:

```
class semigroup (G : Type*) [has_mul G] := ...
class mul_one_class (M : Type*) [has_one M] [has_mul M] := ...
class comm_semigroup (G : Type*) [semigroup G] := ...
class monoid (M : Type*) [semigroup M] [mul_one_class M].
class comm_monoid (M : Type*) [monoid M] [comm_semigroup M].
```

Providing an instance for the natural numbers is straightforward, although it now involves instantiating each step in the hierarchy separately:

```
instance : semigroup ℕ := sorry -- details omitted
instance : mul_one_class ℕ := sorry -- details omitted
instance : comm_semigroup ℕ := sorry -- details omitted
instance : monoid ℕ := sorry -- details omitted
instance : comm_monoid ℕ := sorry -- details omitted
```

When we want to instantiate the commutative monoid structure on the product of two commutative monoids, we see that the length of types starts to grow noticeably:

```
instance prod.has_mul [has_mul G] [has_mul H] : has_mul (G × H) :=
{ mul := λ a b, (a.1 * b.1, a.2 * b.2) }
instance prod.semigroup [has_mul G] [has_mul H]
  [semigroup G] [semigroup H] : semigroup (G × H) :=
sorry -- details omitted
...
instance prod.comm_monoid
  [has_one M] [has_one N] [has_mul M] [has_mul N]
  [semigroup M] [semigroup N] [mul_one_class M] [mul_one_class N]
  [monoid M] [monoid N] [comm_semigroup M] [comm_semigroup N]
  [comm_monoid M] [comm_monoid N] :
  comm_monoid (M × N) :=
sorry -- details omitted
```

The linear growth in the types translates to an exponential growth in the term size of concrete instances, since each instance parameter implicit in `comm_monoid (ℕ × ⋯ × ℕ)` is filled with a term that has itself the same number of instance arguments.

The performance issue of unbundled classes is well known in the Coq community since Coq has a similar implementation of classes to Lean, and the first Coq library using classes for its algebraic hierarchy suffered from slowdowns due to this design issue [25]. The *packed classes* design pattern used for performant canonical structures [13] translates to bundled subclassing: in packed classes, the substructure relation is expressed by declaring the superstructure as an instance, instead of a parameter to the record. Similarly, mathlib prefers bundled classes, expressing the subclass relation through incorporating the superclass as an instance, instead of a parameter on the class's type.

In addition, the deprecation of the `old_structure_cmd` option results in improved performance for unification of instances in the presence of large inheritance chains. Since equality of structures is determined field-wise, incorporating a parent as a field means instances deriving from the same parent instances can be immediately verified to be equal, compared to the `old_structure_cmd` situation where this comparison has to be performed on the union of all fields of all ancestor structures, unfolding all intermediate projections.

An important source of slowdowns is failing instance searches since the entire search space has to be exhaustively checked before failing. If a user omits a hypothesis by mistake, the error message should not be a timeout but instead point out that the instance was not found. Thus, all instances in mathlib are checked by a `fails_quickly` linter, that checks that within an acceptable time (configurable in the linter) synthesis fails to synthesize a given instance when arguments are missing. The `fails_quickly` linter can also detect timeouts caused by looping or diverging synthesis, for example the loop `nonempty` → `has_bot` → `nonempty` in the following code:

```
-- ‘has_bot.bot‘ is notation for the minimum element of ‘α‘
class has_bot (α : Type*) := (bot : α)

instance has_bot_nonempty (α : Type*) [has_bot α] : nonempty α :=
⟨has_bot.bot⟩

-- The natural numbers are well-ordered.
instance nat.subtype.has_bot (s : set ℕ) [decidable_pred (∈ s)]
  [h : nonempty s] : has_bot s := sorry -- proof omitted
```

On the other hand, sometimes failing to synthesize instances should not cause an error, especially in tactics which can handle synthesis failures by switching to a less powerful procedure. In particular, the simplification procedure used instance synthesis to determine whether the type of a subterm had a `subsingleton` instance, which would allow more powerful rewriting in the presence of dependencies between subterms. This meant the simplification tactic searched for `subsingleton` instances for each subterm that another subterm depended on, in the worst case spending more than half its time on failing instance synthesis calls. Modifying the simplification procedure to instead rely on user-supplied congruence lemmas[2] resulted in a speedup of approximately 15 percent over the entirety of mathlib.

[2] `https://github.com/leanprover-community/lean/pull/665`

The depth-first nature of instance synthesis means it is advantageous to try instances that succeed or fail fast before ones that require traversing a full tree before determining their success. The `priority` attribute of instances controls the order in which instances of the same class are considered: higher priorities are tried before lower priorities. The rule of thumb used in mathlib states to assign a low priority to *blanket instances*: those where all explicit parameters to the class are free variables. In particular, all subclass instances are automatically assigned a lower priority. A subtler case involved unification of quotient types: the instance `con.quotient.decidable_eq` states equality is decidable on the quotient of a type `M` by any decidable multiplicative congruence relation of type `con M`.

```
instance con.quotient.decidable_eq {M : Type*} [has_mul M] (c : con M)
  [∀ (a b : M), decidable (c a b)] : decidable_eq (quotient c) :=
sorry -- proof omitted
```

A value for the instance parameter `[has_mul M]` has to be synthesized when `con.quotient.decidable_eq` is considered as candidate instance, meaning this instance will cause a search through all instances of all subclasses of `has_mul`. Since mathlib makes extensive use of other quotient types such as `multiset` α, the quotient of `list` α modulo permutations, specialized instances such as `multiset.decidable_eq : decidable_eq` $\alpha \to$ `decidable_eq (multiset` α`)` are assigned higher priority than the expensive instance `con.quotient.decidable_eq`.

While the above design guidelines have allowed the growth of mathlib's class hierarchy, the fact they often need to be verified and applied manually shows that performance is a key consideration in the further growth of the library.

11 Conclusion

The pervasive use of class-based patterns throughout mathlib demonstrates that the instance parameter mechanism scales to a large interconnected library of mathematical structures. In addition to algebraic, order and topological hierarchies, classes have proved useful for representing morphisms and subobjects. Still, the choice between bundled and unbundled subclassing and the duplication in the additive and multiplicative hierarchy are drawbacks of the use of typeclasses that do not appear in canonical structures or locales. For newcomers, notation for classes such as vector spaces is surprising. Even worse, dangerous instances, definitional equality and divergence are regular sources of errors that require a good understanding of the synthesis mechanism to resolve, and keeping the whole system performant is a permanent source of concern.

Lean 4 brings tabled instance synthesis to improve performance, and linters are able to report both dangerous instances and divergence. Future work should build on these improvements by providing a user friendly way of resolving definitional equality issues, be it a linter or a way to incorporate equality into the synthesis mechanism. In addition, macros that transform binder lists can address some of the unfamiliar notations. Another avenue to address the drawbacks of typeclasses is the integration of classes with another hierarchy mechanism, such as the combination of classes and locales in Isabelle.

References

1 Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, and Kazuhiko Sakaguchi. Competing inheritance paths in dependent type theory: A case study in functional analysis. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *IJCAR 2020*, volume 12167 of *LNCS*, pages 3–20. Springer, 2020. doi:10.1007/978-3-030-51054-1_1.

2 Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in unification. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs 2009*, volume 5674 of *LNCS*, pages 84–98. Springer, 2009. doi:10.1007/978-3-642-03359-9_8.

3 Clemens Ballarin. Locales: A module system for mathematical theories. *J. Autom. Reason.*, 52(2):123–153, 2014. doi:10.1007/s10817-013-9284-7.

4 Clemens Ballarin. Exploring the structure of an algebra text with locales. *J. Autom. Reason.*, 64(6):1093–1121, 2020. doi:10.1007/s10817-019-09537-9.

5 Edwin C. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013. doi:10.1017/S095679681300018X.

6 Kevin Buzzard, Johan Commelin, and Patrick Massot. Formalising perfectoid spaces. In *CPP '20*, Certified Programs and Proofs, pages 299–312. ACM, 2020. doi:10.1145/3372885.3373830.

7 Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy builder: Algebraic hierarchies made easy in Coq with Elpi (system description). In Zena M. Ariola, editor, *FSCD 2020*, volume 167 of *LIPIcs*, pages 34:1–34:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.FSCD.2020.34.

8 Johan Commelin and Robert Y. Lewis. Formalizing the ring of Witt vectors. In Catalin Hriţcu and Andrei Popescu, editors, *CPP '21*, Certified Programs and Proofs, pages 264–277. ACM, 2021. doi:10.1145/3437992.3439919.

9 L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean theorem prover (system description). In A. P. Felty and A. Middeldorp, editors, *Automated Deduction - CADE-25*, volume 9195 of *LNCS*, pages 378–388. Springer, 2015. doi:10.1007/978-3-319-21401-6_26.

10 Leonardo de Moura, Jeremy Avigad, Soonho Kong, and Cody Roux. Elaboration in dependent type theory. *CoRR*, abs/1505.04324, 2015. arXiv:1505.04324.

11 Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE-28*, volume 12699 of *LNCS*, pages 625–635. Springer, 2021. doi:10.1007/978-3-030-79876-5_37.

12 Dominique Devriese and Frank Piessens. On the bright side of type classes: Instance arguments in Agda. *SIGPLAN Not.*, 46(9):143–155, September 2011. doi:10.1145/2034574.2034796.

13 François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs 2009*, volume 5674 of *LNCS*, pages 327–342. Springer, 2009. doi:10.1007/978-3-642-03359-9_23.

14 Adam Grabowski, Artur Kornilowicz, and Christoph Schwarzweller. On algebraic hierarchies in mathematical repository of Mizar. In Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki, editors, *FedCSIS 2016*, volume 8 of *Annals of Computer Science and Information Systems*, pages 363–371. IEEE, 2016. doi:10.15439/2016F520.

15 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, *ITP 2013*, volume 7998 of *LNCS*, pages 279–294. Springer, 2013. doi:10.1007/978-3-642-39634-2_21.

16 Ralf Jung. Exponential blowup when using unbundled typeclasses to model algebraic hierarchies, 2019. Accessed 2022-02-01. URL: https://www.ralfj.de/blog/2019/05/15/typeclasses-exponential-blowup.html.

17 Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. Locales - A sectioning concept for Isabelle. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin-Mohring, and Laurent Théry, editors, *TPHOLs'99*, volume 1690 of *LNCS*, pages 149–166. Springer, 1999. doi:10.1007/3-540-48256-3_11.

18 A. Mahboubi and E. Tassi. *The Mathematical Components Libraries*. Zenodo, Genève, Switzerland, 2017. doi:10.5281/zenodo.4457887.

19 Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. *SIGPLAN Not.*, 45(10):341–360, October 2010. doi:10.1145/1932682.1869489.

20 The Rust team. The Rust reference 1.57.0, 2021. Accessed 2021-12-22. URL: https://doc.rust-lang.org/1.57.0/reference/index.html.

21 Amokrane Saïbi. Typing algorithm in type theory with inheritance. In *Principles of Programming Languages*, POPL '97, pages 292–301. ACM, 1997. doi:10.1145/263699.263742.

22 Kazuhiko Sakaguchi. Validating mathematical structures. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *IJCAR 2020*, volume 12167 of *LNCS*, pages 138–157. Springer, 2020. doi:10.1007/978-3-030-51054-1_8.

23 Daniel Selsam, Sebastian Ullrich, and Leonardo de Moura. Tabled typeclass resolution. *CoRR*, abs/2001.04301, 2020. arXiv:2001.04301.

24 Matthieu Sozeau and Nicolas Oury. First-class type classes. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *TPHOLs 2008*, volume 5170 of *LNCS*, pages 278–293. Springer, 2008. doi:10.1007/978-3-540-71067-7_23.

25 Bas Spitters and Eelis van der Weegen. Developing the algebraic hierarchy with type classes in Coq. In Matt Kaufmann and Lawrence C. Paulson, editors, *ITP 2010*, volume 6172 of *LNCS*, pages 490–493. Springer, 2010. doi:10.1007/978-3-642-14052-5_35.

26 The mathlib Community. The Lean mathematical library. In J. Blanchette and C. Hriţcu, editors, *CPP 2020*, Certified Programs and Proofs, pages 367–381. ACM, 2020. doi:10.1145/3372885.3373824.

27 Floris van Doorn, Gabriel Ebner, and Robert Y. Lewis. Maintaining a library of formal mathematics. In Christoph Benzmüller and Bruce R. Miller, editors, *CICM 2020*, volume 12236 of *LNCS*, pages 251–267. Springer, 2020. doi:10.1007/978-3-030-53518-6_16.

28 P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Principles of Programming Languages*, POPL '89, pages 60–76. ACM, 1989. doi:10.1145/75277.75283.

29 Markus Wenzel. Type classes and overloading in higher-order logic. In Elsa L. Gunter and Amy P. Felty, editors, *TPHOLs'97*, volume 1275 of *LNCS*, pages 307–322. Springer, 1997. doi:10.1007/BFb0028402.

30 Eric Wieser. Scalar actions in Lean's mathlib. *CoRR*, abs/2108.10700, 2021. arXiv:2108.10700.

A Complete, Mechanically-Verified Proof of the Banach-Tarski Theorem in ACL2(R)

Jagadish Bapanapally
Department of Computer Science, University of Wyoming, Laramie, WY, USA

Ruben Gamboa
Department of Computer Science, University of Wyoming, Laramie, WY, USA

Abstract

This paper presents a formal proof of the Banach-Tarski theorem in ACL2(r). The Banach-Tarski theorem states that a unit ball can be partitioned into a finite number of pieces that can be rotated to form two identical copies of the ball. We have formalized 3D rotations and generated a free group of 3D rotations of rank 2. In prior work, the non-denumerability of the reals was proved in ACL2 (r), and a version of the Axiom of Choice that can consistently select a representative element from an equivalence class was introduced in ACL2 version 3.1. Using the free group of rotations, and this prior work, we show that the unit sphere can be decomposed into two sets, each equivalent to the original sphere. Then we show that the unit ball except for the origin can be decomposed into two sets each equivalent to the original ball by mapping the points of the unit ball to the points on the sphere. Finally, we handle the origin by rotating the unit ball around an axis such that the origin falls inside the sphere. Seemingly paradoxically, the construction results in two copies of the unit ball.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases ACL2(r), Banach-Tarski, Rotations

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.5

Supplementary Material *Software (Source Code)*:
`https://github.com/acl2/acl2/tree/master/books/nonstd/nsa/Banach-Tarski`
archived at `swh:1:dir:964ff589fe9739a46201bbeac71a6733aa4f274b`

Funding *Jagadish Bapanapally*: The research presented in this paper was partially supported by a grant from IOG Singapore Pte. Ltd.

Acknowledgements We want to thank professor John Cowles at the University of Wyoming for assisting us in the proof verifying the denumerability of the poles.

1 Introduction

The Banach-Tarski theorem [11] states that we can break the unit ball into a finite number of sets, then rotate the sets to form two identical copies of the unit ball. This seems impossible because it breaks our intuition that when we partition the ball into finite sets, the total volume of the pieces must be the same as the volume of the original ball. This would be the case if all the pieces had a well-defined volume. The Banach-Tarski theorem is possible because the construction breaks the ball into non-measurable sets [7], which means they don't have a well-defined volume. Such a partition of the unit ball is obviously subtle, and the entire construction depends on the Axiom of Choice [7] and the non-denumerability of reals [4]. Many properties of matrix algebra [5], modular arithmetic [1] and trigonometric functions [3] that are needed for the proof have already been formalized in ACL2(r).

In this paper we present a complete proof of the Banach-Tarski theorem in ACL2(r) [6], a variant of ACL2 that offers support for the real numbers by the way of non-standard analysis. The ACL2(r) source files for this proof are in the ACL2 community books under the directory nonstd/nsa/Banach-Tarski/. We begin in Section 2, with a free group of reduced words using

lists. The lists in this group are decomposed in various ways, resulting in multiple ways of reconstructing the free group. This is, in fact, a key step in the Banach-Tarski paradox. Then, in Section 3, we present a free group of rotations that correspond to the free group of reduced words, as shown in Section 4. We also formalize 3D rotations and prove some crucial properties of rotations.

Using the free group of rotations, in Section 5 we show how almost all of the unit sphere can be partitioned into sets, such that the sets can be rotated and rearranged to form two copies of the unit sphere, almost. This works for all points on the sphere, except the points that lie on the axis of rotation of one of the rotations in the free group. The set of such points is countable, and this is shown by proving some basic facts about constructing countable sets. This establishes the Hausdorff's Paradox. The proof then proceeds by adding some extra rotations that essentially wipe the poles of the rotations. This proves the Banach-Tarski paradox on the unit sphere.

Finally, in Section 6 the construction is extended to the unit ball. Except for the origin, this is done by projecting each point inside the unit ball onto the unit sphere, and using the decomposition of the unit sphere constructed in Section 5. The final step is to account for the origin by introducing a rotation around a point close enough to the origin that the origin is always mapped to a point inside the unit sphere.

Throughout, the proof of the Banach-Tarski theorem involves proving a lot of equivalences between various sets. In our proof of the Banach-Tarski theorem we use predicates and Skolem functions to represent various sets, then prove equivalence between these predicates.

2 A Free Group of Reduced Words

In this section, we introduce the free group over the letters a and b. This group contains all words that can be formed from a, b, a^{-1}, and b^{-1} such that no letter and its inverse appear together. For example, $abba$ is a member of this free group but $abb^{-1}a$ is not.

We use lists in ACL2(r) to represent words. A weak word is an empty list or a list that has characters a or a^{-1} or b or b^{-1}. e.g., $'(a\ b\ b^{-1}\ a^{-1})$ is a weak word. The single quote in the example means that it is a list, which would otherwise become a function call. In the ACL2(r) source files, we have defined the functions `wa`, `wa-inv`, `wb` and `wb-inv` which return the ACL2(r) characters $\#\backslash a$, $\#\backslash b$, $\#\backslash c$, and $\#\backslash d$ respectively. e.g., (`wa`)=$\#\backslash a$. We use the ACL2(r) characters $\#\backslash a$, $\#\backslash b$, $\#\backslash c$, and $\#\backslash d$ to represent a, a^{-1}, b, and b^{-1} respectively, but in this paper we will simply refer to a, a^{-1}, b, and b^{-1} to avoid confusion. The predicate *weak-wordp* recognizes elements of the set of weak words, as shown in Listing 1. Since ACL2(r) does not have support for infinite sets, such as the set of weak words, we represent these sets implicitly using recognizers for their elements.

A reduced word is a weak word such that character a^{-1} does not appear beside the character a and character b^{-1} does not appear beside the character b in the list. e.g., $'(a\ b\ a^{-1})$ is a reduced word and $'(a\ a^{-1}\ b)$ is not a reduced word. The predicates *a-wordp*, *a-inv-wordp*, *b-wordp*, and *b-inv-wordp* represent the set of reduced words that start with characters a, a^{-1}, b, and b^{-1} respectively. The predicate *reducedwordp*, shown in Listing 2, represents the set of all reduced words. *reducedwordp* returns true if the argument belongs to the set *a-wordp* or *a-inv-wordp* or *b-wordp* or *b-inv-wordp* or if it is an empty list.

The function *word-inverse* finds the inverse of a reduced word. If the argument is a weak word, *word-inverse* flips each character in the list to its inverse and then reverses the list. e.g., *word-inverse*$('(a\ a^{-1}\ b^{-1})) = '(b\ a\ a^{-1})$. Listing 3 shows the definition of the flip function and the inverse function.

Listing 1 Definition of the set of weak words.

```
(defun weak-wordp (w)
  (cond ((atom w) (equal w nil))
        (t (and (or (equal (first w) (wa))
                    (equal (first w) (wa-inv))
                    (equal (first w) (wb))
                    (equal (first w) (wb-inv)))
                (weak-wordp (rest w))))))
```

Listing 2 Definition of the set of reduced words.

```
(defun reducedwordp (x)
  (or (a-wordp x)
      (a-inv-wordp x)
      (b-wordp x)
      (b-inv-wordp x)
      (equal x '())))
```

The group operation *compose* takes two arguments. If the arguments are weak words, then the *compose* function first appends the two lists and then "fixes" the result by deleting any letter and its inverse that appear beside each other. Thus, the final result of *compose* is always a reduced word. e.g., *compose*('(*a b b*), '(b^{-1})) = '(*a b*). Listing 4 shows the definition of the fixing function and the group operation *compose*.

If $w1$ and $w2$ are reduced words, then (append $w1$ $w2$) is a weak word. If x is a weak word, then *word-fix*(x) returns a reduced word. So, *compose*($w1, w2$) = *word-fix*(append $w1$ $w2$) is a reduced word. This establishes that *compose* is closed over the set of reduced words. In fact, *compose* is a group operator over reduced words, as suggested earlier. A key lemma required to prove that it satisfies the associative property and the inverse property is that if x is a reduced word, then *word-fix*(rev(x)) = (rev(*word-fix*(x))), which we proved by induction on x. This proves that, with the group operation *compose* and considering the empty list as the identity element, the set of reduced words is a free group. Listing 5 shows the group properties of this set.

Listing 3 Definition of the Inverse operation.

```
;; Definition of the flip function
(defun word-flip (x)
  (cond ((atom x) nil)
        ((equal (car x) (wa)) (cons (wa-inv) (word-flip (cdr x))))
        ((equal (car x) (wa-inv)) (cons (wa) (word-flip (cdr x))))
        ((equal (car x) (wb)) (cons (wb-inv) (word-flip (cdr x))))
        ((equal (car x) (wb-inv)) (cons (wb) (word-flip (cdr x))))))

;; Definition of the Inverse operation
(defun word-inverse (x)
  (rev (word-flip x)))
```

Listing 4 Definition of the group operation *compose*.

```
;; Definition of the fixing function
(defun word-fix (w)
  (if (atom w)
      nil
    (let ((fixword (word-fix (cdr w))))
      (let ((w (cons (car w) fixword)))
        (cond ((equal fixword nil)
               (list (car w)))
              ((equal (car (cdr w)) (wa))
               (if (equal (car w) (wa-inv))
                   (cdr (cdr w))
                   w))
              ((equal (car (cdr w)) (wa-inv))
               (if (equal (car w) (wa))
                   (cdr (cdr w))
                   w))
              ((equal (car (cdr w)) (wb))
               (if (equal (car w) (wb-inv))
                   (cdr (cdr w))
                   w))
              ((equal (car (cdr w)) (wb-inv))
               (if (equal (car w) (wb))
                   (cdr (cdr w))
                   w)))))))

;; Definition of the group operation
(defun compose (x y)
  (word-fix (append x y)))
```

Listing 5 Group properties of the set of reduced words.

```
;; Closure property
(defthmd closure-prop
  (implies (and (reducedwordp x)
                (reducedwordp y))
           (reducedwordp (compose x y)))
  :hints ...)

;; Associative property
(defthmd assoc-prop
  (implies (and (reducedwordp x)
                (reducedwordp y)
                (reducedwordp z))
           (equal (compose (compose x y) z)
                  (compose x (compose y z))))
  :hints ...)

;; Inverse property
(defthmd reduced-inverse
  (implies (reducedwordp x)
           (equal (compose x (word-inverse x))
                  '()))
  :hints ...)
```

Denote the set of reduced words by $W(a,b)$, the set of reduced words starting with character a by $W(a)$, and similarly for $W(a^{-1})$, $W(b)$, and $W(b^{-1})$. Then clearly $W(a,b) = () \bigsqcup W(a) \bigsqcup W(a^{-1}) \bigsqcup W(b) \bigsqcup W(b^{-1})$, where $\bigsqcup$ denotes the union of *disjoint* sets. In addition to this we can show two other equivalences of the set of reduced words:

- $W(a,b) = a^{-1}W(a) \bigsqcup W(a^{-1})$ and
- $W(a,b) = b^{-1}W(b) \bigsqcup W(b^{-1})$,

where $xW(y) = \{xw \mid w \in W(y)\}$.

As we mentioned previously, we use recognizers to represent sets, and since ACL2(r) supports quantifiers via Skolem functions, we represent the set $a^{-1}W(a) = \{x \mid \exists w \in W(a) \text{ s.t. } x = compose('(a^{-1}), w)\}$ and $b^{-1}W(b) = \{x \mid \exists w \in W(b) \text{ s.t. } x = compose('(b^{-1}), w)\}$. The formal proof follows the proof of the two equivalences given in [11]. If an element x belongs to $W(a)$ then $compose('(a^{-1}), x) \subset () \bigsqcup W(a) \bigsqcup W(b) \bigsqcup W(b^{-1})$, because the *compose* function appends $'(a^{-1})$ and x and then deletes the first two characters a^{-1} and a in the appended list as they are inverses of each other. Moreover, the first character of $a^{-1}x$ is the second character of x, so it cannot be a^{-1}, since x is a reduced word that starts with a.

Likewise, if an element x belongs to $W(a)$ or $W(b)$ or $W(b^{-1})$ then there exists an element *word-a* that belongs to the set $W(a)$ such that x equals to $(compose('(a^{-1}), word\text{-}a)$, namely the element $a^{-1}x$. So, we have $a^{-1}W(a) = () \bigsqcup W(a) \bigsqcup W(b) \bigsqcup W(b^{-1})$. The same way we prove $b^{-1}W(b) = () \bigsqcup W(a) \bigsqcup W(a^{-1}) \bigsqcup W(b)$. With these two equivalences of the sets $a^{-1}W(a)$ and $b^{-1}W(b)$ we get two corollaries Corollary 2 and Corollary 3 which we use to prove the Banach-Tarski theorem on S^2.

▶ **Corollary 1.** $W(a,b) = () \bigsqcup W(a) \bigsqcup W(a^{-1}) \bigsqcup W(b) \bigsqcup W(b^{-1})$

▶ **Corollary 2.** $W(a,b) = a^{-1}W(a) \bigsqcup W(a^{-1})$

▶ **Corollary 3.** $W(a,b) = b^{-1}W(b) \bigsqcup W(b^{-1})$

Notice that these corollaries, while being about lists, already contain the key to the Banach-Tarski paradox. The set $W(a,b)$ is decomposed into five disjoint subsets, then it can be reconstructed in two different ways by taking the union of two of those subsets after prepending a letter to one of the subsets. In the same way, the sphere can be deconstructed into a number of sets, which can then be rotated and reassembled in two different ways to reconstruct a unit sphere.

3 A Free Group of 3D Matrices

Matrices in ACL2 are represented with the data structure *array2p*. We define a predicate *r3-matrixp* that recognizes the set of 3D matrices: *r3-matrixp* returns true if the argument is of type *array2p* and if its dimensions are 3×3, and if each element of the matrix is a real number.

We define now the four matrices A^+, A^-, B^+, and B^- as

$$A^{\pm} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{3} & \mp\frac{2\sqrt{2}}{3} \\ 0 & \pm\frac{2\sqrt{2}}{3} & \frac{1}{3} \end{bmatrix} \quad B^{\pm} = \begin{bmatrix} \frac{1}{3} & \mp\frac{2\sqrt{2}}{3} & 0 \\ \pm\frac{2\sqrt{2}}{3} & \frac{1}{3} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and we associate these matrices with the letters a, a^{-1}, b, and b^{-1} from the free group respectively. Moreover, we associate a list $(x_1, x_2, \ldots, x_n) \in W(a,b)$ with the matrix $X_1 \times X_2 \times \cdots \times X_n$, where $\times$ denotes matrix multiplication, and X_i is the matrix associated with letter x_i. We have defined a recursive function *rotation* that performs this mapping

from words in the free group to rotations. Denote the resulting set $R(a,b)$. i.e., $R(a,b) = \{\mathit{rotation}(w) \mid w \in W(a,b)\}$. By induction, it is easy to verify that every element of the set $R(a,b)$ belongs to *r3-matrixp*.

To show the set $R(a,b)$ is a free group isomorphic to $W(a,b)$, we show that if $w \in W(a,b)$ and w is not the empty list, then $\mathit{rotation}(w)$ is not equal to I, the identity matrix. Equivalently, we show that $(\mathit{rotation}(w))(0,1,0) \neq (0,1,0)$ unless w is the empty list.

To do this, suppose that $w \in R(a,b)$, and consider the rotation $R(w)$. In particular, suppose that $R(w)$ rotates the point $(0,1,0)$ to (x',y',z'). Define (x,y,z) as

$$(x,y,z) = 3^n \left(\frac{x'}{\sqrt{2}}, y', \frac{z'}{\sqrt{2}}\right)$$

where $n = |w|$. Using induction, we showed that x, y, and z are integers.

So now, suppose that $(\mathit{rotation}(w))(0,1,0) = (0,1,0)$ for some non-empty word w. It follows that $(x,y,z) = (0,3^n,0)$, where $n = |w| > 0$, thus $x \equiv y \equiv z \equiv 0 \pmod 3$. But this cannot be the case. If $|w| = 1$, then $\mathit{rotation}(w)$ is one of $A^{\pm}$ or $B^{\pm}$, and considering each of the four cases by brute force, it is clear that $(x,y,z) \not\equiv (0,0,0) \pmod 3$. Using induction, there are 16 cases to consider, but in all of these cases we again conclude that $(x,y,z) \not\equiv (0,0,0) \pmod 3$. This shows that if $|w| > 0$, then $\mathit{rotation}(w)$ is not the identity matrix.

Using this fact, the group properties of $W(a,b)$, and the associativity of the matrix multiplication, we then showed that there is a one-to-one relation between the set $R(a,b)$ and the set $W(a,b)$. So defining $R(a) = \{\mathit{rotation}(w) \mid w \in W(a)\}$, $R(a^{-1}) = \{\mathit{rotation}(w) \mid w \in W(a^{-1})\}$, $R(b) = \{\mathit{rotation}(w) \mid w \in W(b)\}$, and $R(b^{-1}) = \{\mathit{rotation}(w) \mid w \in W(b^{-1})\}$, then the set of rotations $R(a,b)$ can be partitioned as

$$R(a,b) = I \bigsqcup R(a) \bigsqcup R(a^{-1}) \bigsqcup R(b) \bigsqcup R(b^{-1}).$$

That is, the paradoxical partition of the free words $W(a,b)$ from Section 2 can be reproduced in the set of rotations $R(a,b)$.

4 A Free Group of Rotations of Rank 2

Before proceeeding directly into the Banach-Tarski construction, we need to prove some basic facts from matrix algebra. As discussed previously, the matrix transpose operation (*m-trans*) was formalized in prior work [5], and as part of that, it was shown that $(A \times B)^T = B^T \times A^T$.

We extended that formalization by introducing the function *r3-m-determinant* that computes the determinant of a matrix, the function *r3-m-inverse* that computes the inverse of a 3D matrix (when possible). Using these functions, we defined the predicate *r3-rotationp* that recognizes rotations in $\mathbb{R}^3$.

▶ **Definition 4.** *A matrix M is a rotation matrix if it satisfies these conditions [10]:*

- *M is a 3D matrix,*
- *$M^{-1} = M^T$, and*
- $\det(M) = 1$.

Another important detail is that every element of $R(a,b)$ must be a rotation of $\mathbb{R}^3$. Given the correspondence between $R(a,b)$ and $W(a,b)$ established in Section 3, what we need to show is that for any $w \in W(a,b)$, $\mathit{rotation}(w)$ satisfies the axioms in Definition 4. This was done using induction on the list w. It is easy to verify that the base cases are rotations; i.e., I, A^+, A^-, B and B^- are all rotation matrices. For the induction to go through, the lemma we need to prove $\mathit{rotation}(xw)$ is a rotation in $\mathbb{R}^3$ given that $\mathit{rotation}(w)$ is a rotation, is that the product of two rotation matrices M_1 and M_2 is also a rotation matrix.

The final lemma from matrix algebra that we needed was to show that every rotation matrix preserves distances [9]; i.e., that $||Mx|| = ||x||$ whenever M is a rotation matrix and x is a vector. Since the focus of this project was on the Banach-Tarski paradox and not matrix algebra, we proceeded to prove these results as directly as possible, without using deeper results from linear algebra, such as the geometric meaning of determinants. In the end, we proceeded using the roadmap suggested by the following lemmas, all proved in ACL2(r):

▶ **Lemma 5.** $r3\text{-}matrixp(m_1) \wedge r3\text{-}matrixp(m_2) \implies r3\text{-}matrixp(m_1 \times m_2)$

▶ **Lemma 6.** $r3\text{-}matrixp(m_1) \wedge r3\text{-}matrixp(m_2) \implies \det(m_1 \times m_2) = \det(m_1) \cdot det(m_2)$

▶ **Lemma 7.** $r3\text{-}matrixp(m) \implies m \times I = I \times m = m$

▶ **Lemma 8.** $r3\text{-}matrixp(m) \wedge \det(m) \neq 0 \implies m \times m^{-1} = m^{-1} \times m = I$

▶ **Lemma 9.** $r3\text{-}matrixp(m_1) \wedge \det(m_1) \neq 0 \wedge r3\text{-}matrixp(m_2) \wedge \det(m_2) \neq 0$
$\implies (m_1 \times m_2)^{-1} = m_2^{-1} \times m_1^{-1}$

▶ **Lemma 10.** $r3\text{-}rotationp(m_1) \wedge r3\text{-}rotationp(m_2) \implies r3\text{-}rotationp(m_1 \times m_2)$

▶ **Lemma 11.** $r3\text{-}rotationp(m) \implies r3\text{-}rotationp(m^{-1})$

▶ **Lemma 12.** *Rotations preserve distances.*

Proof. Let $p_1 = (x_1, y_1, z_1)$ and R be a rotation matrix, and consider $p_2 = Rp_1 = (x_2, y_2, z_2)$. Using the previous lemmas,

$$\begin{aligned} x_1^2 + y_1^2 + z_1^2 &= p_1^T \times p_1 \\ &= p_1^T \times (I \times p_1) \\ &= p_1^T \times ((R^{-1} \times R) \times p_1) \\ &= p_1^T \times ((R^T \times R) \times p_1) \\ &= (p_1^T \times R^T) \times (R \times p_1) \\ &= (R \times p_1)^T \times (R \times p_1) \\ &= p_2^T \times p_2 \\ &= x_2^2 + y_2^2 + z_2^2. \end{aligned}$$

◀

5 Banach-Tarski Theorem on the Unit Sphere

Before finishing the proof of the Banach-Tarski theorem for the unit sphere, we want to mention two key lemmas needed to carry out the proof. First, if $w_1, w_2 \in W(a, b)$, then by the definition of *rotation* and *compose*, $rotation(w_1) \times rotation(w_2) = rotation(compose(w_1, w_2))$. Second, if $r \in R(a, b)$, then $\exists w \in W(a, b)$ such that $r = rotation(w)$, and by the previous lemma $r^{-1} = rotation(w^{-1})$. Moreover, since $w^{-1} \in W(a, b)$, $r^{-1} \in R(a, b)$.

Returning to the main proof, let D be the set of poles of all of the rotations belonging to the set $R(a, b) - I$; i.e., $D = \{p \in S^2 \mid \exists r.\, r \in R(a, b) \wedge r \neq I \wedge r(p) = p\}$.

Now, consider a point $p \in S^2 - D$ and $r \in R(a, b)$. It follows that $r(p) \in S^2 - D$ as well. Otherwise, $r(p) \in D$ and by the definition of D there exists a witness $r_w \in R(a, b) - I$ such that $r_w(r(p)) = r(p)$. But then $r^{-1}(r_w(r(p))) = p$. By the previous lemmas, $r^{-1} r_w r \neq I \implies p \in D$. This proves if $r \in R(a, b)$ and $p \in S^2 - D$, then $r(p) \in S^2 - D$.

Define the orbit of a point $p \in S^2 - D$ as $\{r(p) \mid r \in R(a,b)\}$. Using the Axiom of Choice, implemented as `defchoose` in ACL2, we can choose one representative of each of these orbits. Let M be the set of all of the chosen points from each of the orbits. In Section 5.1 we will show how we used the Axiom of Choice in our proof and how we decomposed $S^2 - D$ into two sets each equivalent to $S^2 - D$. Then in Section 5.2 we'll show the set D is countable; i.e., we will show all the poles of rotations belonging to $R(a,b)$ can be enumerated. Since S^2 is not countable, there exists a point $P_{s2} \in S^2 - D$. Then in Section 5.3 we find an angle $a_{s2} \in [0, 2\pi)$ such that the rotation of any point in D around the axis from the origin to P_{s2} by an angle that is a multiple of a_{s2}, the resulting point does not lie in the set D. The remainder of the proof decomposes S^2 into two sets each equivalent to S^2 by proving equivalences between different sets as suggested in Section 1.

5.1 Decomposing the Unit Sphere minus the Set of Poles

ACL2 supports existential quantification by the way of the `defun-sk` event [8]. We have defined the orbit of a point $point = \{o\text{-}point \mid \exists w.\, w \in W(a,b) \wedge o\text{-}point = rotation(w) \times point\}$ as a Skolem function using defun-sk as shown below.

```
(defun-sk orbit-point-p-q (o-point point)
  (exists w
          (and (reducedwordp w)
               (m-= (m-* (rotation w (acl2-sqrt 2))
                         point)
                    o-point))))
```

The function *orbit-point-p-q* returns true if the point *o-point* belongs to the orbit of *point* and it chooses a witness reduced word w such that $o\text{-}point = rotation(w) \times point$.

Now using the Axiom of Choice we want to choose one representative from each of the orbits of the points in the set $S^2 - D$. The Axiom of Choice in ACL2 is implemented using `defchoose` which was previously used in the proof of the Vitali's theorem [2]. So the choice set M is defined as follows:

```
(defchoose choice-set-s2-d-p (c-point) (p)
  (and (point-in-r3 c-point)
       (orbit-point-p-q c-point p))
  :strengthen t)
```

In the definition of *choice-set-s2-d-p*, *point-in-r3* is the predicate that recognizes points in $\mathbb{R}^3$. If $p \in S^2$, then *choice-set-s2-d-p*(p) picks a point *c-point* in $\mathbb{R}^3$ that is in the orbit of the point p. The *strengthen* option in the choice function ensures that the same canonical witness is chosen for any other point *p1* in the same equivalence class as p.

Since M contains one representative from each of the orbits of the points belonging to the set $S^2 - D$, $S^2 - D = R(a,b)M$. For example, below is how we define the set $R(a,b)M = \{p \mid \exists p_1.\, p_1 \in S^2 - D \wedge c_{p1}$ is the chosen point from the orbit of $p_1 \wedge \exists r \in R(a,b).\, r \times c_{p1} = p\}$. Similarly, we define the sets M, $R(a)M$, $R(a^{-1})M$, $R(b)M$, $R(b^{-1})M$, $a^{-1}R(a)M$, and $b^{-1}R(b)M$.

```
(defun-sk diff-s2-d-p-q-1 (cp1 p)
  (exists w
          (and (reducedwordp w)
               (m-= (m-* (rotation w (acl2-sqrt 2)) cp1) p))))

(defun-sk diff-s2-d-p-q (p)
  (exists p1
          (and (s2-d-p p1)
               (diff-s2-d-p-q-1 (choice-set-s2-d-p p1)
                                p))))

;; Definition of the set R(a,b)M
(defun diff-s2-d-p (p)
  (and (point-in-r3 p)
       (diff-s2-d-p-q p)))
```

If the sets M, $R(a)M$, $R(a^{-1})M$, $R(b)M$, $R(b^{-1})M$ are disjoint, and the sets $a^{-1}R(a)M$, $R(a^{-1})M$ are disjoint, and $b^{-1}R(b)M$, $R(b^{-1})M$ are disjoint, then we have our decomposition of $S^2 - D$. Suppose that $R(a)M$ and $R(b)M$ are not disjoint. To simplify the discussion, define $\hat{p}$ as the point chosen for the orbit of p, and R_w as the rotation matrix of the word w.

Now, let p be a point in the intersection, i.e., $p \in R(a)M$ and $p \in R(b)M$. Then $\exists p_a \in S^2 - D$ and $\exists w_a \in W(a)$ such that $R_{w_a} \times \hat{p_a} = p$ and $\exists p_b \in S^2 - D$ and $\exists w_b \in W(b)$ such that $R_{w_b} \times \hat{p_b} = p$. Since $\hat{p_a}$ lies in the orbit of p_a, $\exists w_{pa} \in W(a,b)$ such that $R_{w_{pa}} \times p_a = \hat{p_a}$ and $\exists w_{pb} \in W(a,b)$ such that $R_{w_{pb}} \times p_b = \hat{p_b}$. So, $R_{w_a} \times R_{w_{pa}} \times p_a = p = R_{w_b} \times R_{w_{pb}} \times p_b$, which implies p_a and p_b belong to the same orbit. In other words, $\hat{p_a} = \hat{p_b}$, since those are the representatives points for their (one) orbit. Since, $R_{w_a} \times \hat{p_a} = R_{w_b} \times \hat{p_b}$, we have that $R_{w_b^{-1}w_a} \times \hat{p_a} = R_{w_b^{-1}} \times R_{w_a} \times \hat{p_a} = \hat{p_b} = \hat{p_a}$. Notice that $compose(w_b^{-1}, w_a) \neq$ '(), since w_b^{-1} ends with b^{-1} and w_a starts with a. Thus, $R_{w_b^{-1}w_a} \neq I$ which implies that $\hat{p_a} \in D$. But this is a contradiction since $\hat{p_a}$ is in the orbit of p_a. So, the sets $R(a)M$ and $R(b)M$ must be disjoint. Similar arguments show that the other sets are also disjoint. By the definition of $R(a,b)$ and by Corollary 2, Corollary 3 we can transfer the decomposition of $W(a,b)$ into the following decompositions of the set $S^2 - D$. Thus, the set $S^2 - D$ can be decomposed into two disjoint copies of itself. Listing 6 shows the proof of these decompositions of $S^2 - D$ in ACL2(r) where *s2-d-p* is the recognizer for the set $S^2 - D$, *diff-n-s2-d-p* is the recognizer for the set M, *diff-a-s2-d-p* is the recognizer for the set $R(a)M$, *diff-a-inv–s2-d-p* is the recognizer for the set $R(a^{-1})M$, *diff-b-s2-d-p* is the recognizer for the set $R(b)M$, *diff-b-inv-s2-d-p* is the recognizer for the set $R(b^{-1})M$, *a-inv-diff-a-s2-d-p* is the recognizer for the set $a^{-1}R(a)M$, and *b-inv-diff-b-s2-d-p* is the recognizer for the set $b^{-1}R(b)M$.

$$S^2 - D = R(a,b)M = M \bigsqcup R(a)M \bigsqcup R(a^{-1})M \bigsqcup R(b)M \bigsqcup R(b^{-1})M$$
$$S^2 - D = a^{-1}R(a)M \bigsqcup R(a^{-1})M$$
$$S^2 - D = b^{-1}R(b)M \bigsqcup R(b^{-1})M$$

5.2 The Set of Poles is Countable

We have seen that $S^2 - D$ can be decomposed, and that the pieces can be recombined in two different ways to create two copies of $S^2 - D$. We now want to show set D is countable.

Listing 6 Decompositions of the set $S^2 - D$ in ACL2(r).

```
;; Unit sphere minus the set of poles broken down into 5 sets
(defthmd s2-d-p-equivalence-1
  (iff (s2-d-p p)
       (or (diff-n-s2-d-p p)
           (diff-a-s2-d-p p)
           (diff-a-inv-s2-d-p p)
           (diff-b-s2-d-p p)
           (diff-b-inv-s2-d-p p)))
  :hints ...)

;; A copy of the unit sphere minus the set of poles
(defthmd s2-d-p-equivalence-2
  (iff (s2-d-p p)
       (or (a-inv-diff-a-s2-d-p p)
           (diff-a-inv-s2-d-p p)))
  :hints ...)

;; Another copy of the unit sphere minus the set of poles
(defthmd s2-d-p-equivalence-3
  (iff (s2-d-p p)
       (or (b-inv-diff-b-s2-d-p p)
           (diff-b-inv-s2-d-p p)))
  :hints ...)
```

Let p be a point in D. Then, there exists a non-empty word $w \in W(a,b)$ such that $R_w p = p$. R_w is a rotation matrix, so it has the form

$$R_w = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}.$$

If R_w is symmetric, then $R_w = R_w^T = R_w^{-1}$. But since $R_w^{-1} = R_{w^{-1}}$, we have that $R_{w^{-1}} = R_w$. But this is not possible, since we have also shown that the mapping from $W(a,b)$ to $R(a,b)$ is one-to-one. So R_w can not be symmetric, thus at least one of $m_{32} \neq m_{23}$, $m_{13} \neq m_{31}$, or $m_{21} \neq m_{12}$ must be true.

Let $K = \sqrt{(m_{32} - m_{23})^2 + (m_{13} - m_{31})^2 + (m_{21} - m_{12})^2}$. Since R_w is not symmetric, $K \neq 0$. So consider the point

$$f_p = \frac{1}{K}(m_{32} - m_{23}, m_{13} - m_{31}, m_{21} - m_{12}).$$

By computation, it is easy to verify that $R_w f_p = f_p$ and $R_w(-f_p) = -f_p$. So indeed, f_p and $-f_p$ are poles of R_w. Now we show these are the only poles of the rotation R_w; i.e., we show the original point $p \in D$ is either equal to f_p or $-f_p$. By construction, $R_w p = p$, and this implies that $p = R_w^{-1} p$, hence $p = R_w^T p$. This means that p satisfies the equations $R_w x = R_w^T x$ and $||x|| = 1$. Geometrically, the solutions to the first equation lie on a line through the origin, and the solutions to the second equation lie on the unit sphere, so the intersection of these results in two points. Algebraically, we proved that the only solutions to these equations are $x = f_p$ and $x = -f_p$.

Using this fact, we can now define an enumeration of all the poles; i.e., we define a sequence that contains all the poles of any rotation R_w corresponding to a non-empty reduced word $w \in W(a,b)$.

The first step is to enumerate all the words in $W(a, b)$. We do this by defining the function *generate-words-main* that returns all the possible words (including weak words, like '$(a\ a^{-1}\ b)$) up to a given input length. It is straightforward to prove that all words in $W(a, b)$ eventually appear in this sequence. Using this enumeration, we then enumerate all the poles by replacing a word w in the sequence with its corresponding pair of poles. The function *poles-list* returns the n^{th} pole, and we proved that all poles appear somewhere in this sequence.

This establishes that the poles are countable, and since the points on the sphere are not, there is at least one point p on the sphere that is not a pole. In the next section, we will use this point to decompose the entire unit sphere.

5.3 Decomposing the Unit Sphere

Up to this point, we have been working with rotations of the form R_w where w is a reduced word. Now, we consider general rotations around a line that passes through the origin and an arbitrary point in the sphere S^2. We defined the function *rotation-3d* that takes an angle $\theta \in [0, 2\pi)$ and a point p in S^2 and returns the matrix corresponding to that general rotation. We will use $R_{p,\theta}$ to denote this general rotation matrix.

Recall from Section 5.2 that there is a point p that lies on the unit sphere but is not one of the poles; i.e., there is a point p such that $p \in S^2 - D$. We would like to choose $\theta \in [0, 2\pi)$ such that for any $p' \in D$, $R_{p,\theta}p' \notin D$; i.e., the rotation $R_{p,\theta}$ rotates D away from D. More than that, we want to choose θ so that even if we rotate multiple times by θ the result is still not in D; i.e., for any $p' \in D$, $R^n_{p,\theta}p' \notin D$ for any $n > 1$. Before finding this θ, we observe (and proved formally in ACL2(r)) that $R_{p,\theta_1} \times R_{p,\theta_2} = R_{p,\theta_1+\theta_2}$ and by induction $R^n_{p,\theta} = R_{p,n\theta}$.

So consider the set of all angles α such that they rotate some element of D to an element of D, perhaps by rotating multiple times; i.e., consider the set

$$A = \left\{\alpha \mid \alpha \in [0, 2\pi) \wedge n \in \mathbb{Z}^+ \wedge \exists p'.\, p' \in D \wedge R_{p,n\alpha}p' \in D\right\}.$$

Now, any angle γ can be written uniquely as $\gamma = 2\pi k + \beta$, where k is an integer and $\beta \in [0, 2\pi)$. In particular, we showed in ACL2(r) that for the positive angle $n\alpha$, there is a unique non-negative integer k and an angle $\beta \in [0, 2\pi)$ such that $n\alpha = 2\pi k + \beta$.

Moreover, suppose p_1 and p_2 are in D, and that there is an angle $\alpha \in [0, 2\pi)$ and a positive integer n such that $R_{p,n\alpha}p_1 = p_2$. As observed, $n\alpha$ can be written uniquely as $n\alpha = 2\pi k + \beta$, which means that the angle α itself can be written as $\frac{2\pi k+\beta}{n}$. Moreover, the angle β is uniquely determined by $n\alpha$, and α is uniquely determined by the choice of p_1, p_2, and n. So enumerating the possible values of n (positive integer) and k (non-negative integer) will also enumerate all the possible values of $\alpha \in A$.

We formalized the proof in ACL2(r) that the Cartesian product of two countable sets is also countable, and we used this result to show that the set A is countable since the sets of possible n and k values as well as pairs (p_1, p_2) are countable. As before, since A is a countable set of angles in $[0, 2\pi)$ there must be some angle $\theta \in [0, 2\pi)$ that is not in A. This angle θ satisfies the desired condition, namely that for any $p' \in D$ and $n \geq 1$, $R^n_{p,\theta}p' = R_{p,n\theta}p' \notin D$.

What we have at this point is a rotation matrix $R_{p,\theta}$ that maps the set of poles P to somewhere in $S^2 - D$. It is easy to verify that if $m \neq n$, then $R_{p,n\theta} \neq R_{p,m\theta}$. Now, consider the set $E = D \bigsqcup R_{p,\theta}D \bigsqcup R_{p,2\theta}D \bigsqcup R_{p,3\theta}D \bigsqcup \cdots$. From the definition of E, it follows easily that $R_{p,\theta}E = E - D$. Thus, the set $S^2 - D$ can be decomposed as

$$S^2 - D = (S^2 - E) \bigsqcup (E - D) = (S^2 - E) \bigsqcup R_{p,\theta}E.$$

With a bit of tedious algebraic manipulation, this formula can be used to find a disjoint decomposition of the entire surface S^2:

$$S^2 = \big((S^2 - D) \cap (S^2 - E)\big) \bigsqcup R_{p,-\theta}\big((S^2 - D) \cap E\big).$$

$S^2 - E$ is just a set, so this equality has the form

$$S^2 = \big((S^2 - D) \cap F\big) \bigsqcup R_{p,-\theta}\big((S^2 - D) \cap E\big).$$

In Section 5.1, we showed how $S^2 - D$ could be decomposed into disjoint sets such that the pieces could be rotated and recombined to create two copies of $S^2 - D$. Replacing $(S^2 - D)$ in the equality above with those two decompositions of $S^2 - D$ results in a similar decomposition of S^2 (but with many more terms). This establishes the Banach-Tarski theorem for the entire sphere S^2. Listing 7 shows the proof of the Banach-Tarski theorem on S^2 in ACL2(r) where *s2-def-p* is the recognizer for the set S^2.

Listing 7 Decompositions of S^2 in ACL2(r).

```
;; Unit sphere broken down into 14 sets
(defthmd s2-equiv-1
  (iff (s2-def-p p)
       (or (set-a1 p)
           (set-a2 p)
           (set-a3 p)
           (set-a4 p)
           (set-a5 p)
           (set-a6 p)
           (set-a7 p)
           (set-a8 p)
           (set-a9 p)
           (set-a10 p)
           (set-a11 p)
           (set-a12 p)
           (set-a13 p)
           (set-a14 p)))
  :hints ...)

;; A copy of the unit sphere
(defthmd s2-equiv-2
  (iff (s2-def-p p)
       (or (set-a-inv-a3 p)
           (set-a-inv-r-a4 p)
           (set-r-1-a-inv-a5 p)
           (set-r-1-a-inv-r-a6 p)
           (set-a7 p)
           (set-a8 p)))
  :hints ...)

;; Another copy of the unit sphere
(defthmd s2-equiv-3
  (iff (s2-def-p p)
       (or (set-b-inv-a9 p)
           (set-b-inv-r-a10 p)
           (set-r-1-b-inv-a11 p)
           (set-r-1-b-inv-r-a12 p)
           (set-a13 p)
           (set-a14 p)))
  :hints ...)
```

6 Banach-Tarski Theorem on the Unit Ball

In Section 5, we showed how the unit sphere S^2 can be decomposed into a finite collection of disjoint sets such that the subsets can be rotated and recombined to construct two copies of S^2. In this section, we use that fact to define a similar construction for the unit ball B^3.

First, we will decompose the unit ball except the origin. Suppose that $p \in B^3 - \{0\}$, and let $r = ||p||$. Define the point $p' = p/r$. It is easy to show that $p' \in S^2$. Geometrically, it is obvious that if we rotate a point on S^2, then all the points along the line from the origin to that point will be rotated by the same angle and direction. We proved this fact algebraically in ACL2(r). Using this fact, it is trivial to generalize the Banach-Tarski decomposition of S^2 into a similar decomposition of $B^3 - \{0\}$.

Generalizing the decomposition to the entire unit ball B^3 is conceptually similar to the way the decomposition of $S^2 - D$ was extended to cover all of S^2. The trick, then, was to find a rotation that would essentially erase the points in D, and this was possible because D is countable. The origin is just a single point, so the same strategy of rotating the origin away should work. The major complication is that any rotation with an axis that passes through the origin will map the origin to itself. So we need to consider rotations along arbitrary axes, and these are not linear transformations, so they cannot be simply encoded as matrices.

Besides the linear transformation of rotation, we also need translation. We defined the function, *rotation-about-arbitrary-line* that accepts an arbitrary point p, an angle θ, and an axis of rotation l (defined using two points), and returns the result of rotating p around l by θ. We proved that this operation satisfies the expected properties of rotations in 3D.

Four of these properties were needed to complete the proof. First, the result of rotating a point p by an angle θ around an axis l is always a point in $\mathbb{R}^3$. Second, if $\theta = 0$, rotation around any axis l by θ is the identity transformation. Third, rotating a point p by an angle θ_1 about an axis l and then rotating the result by an angle θ_2 about the same axis l, is the same as rotating the point p by $\theta_1 + \theta_2$ around l. Finally, the result of rotating the origin around a specific axis l that is close to the origin results in a point that is inside the unit ball B^3. The last two properties combine to show that repeatedly rotating the origin around this specific axis by θ will always yield a result that is inside B^3.

The rest follows the same strategy presented in Section 5.3. Fix the axis of rotation l as above, so that origin is always mapped to some point inside B^3. Now we want to find an angle $\alpha \in [0, 2\pi)$ such that if we rotate the origin by an angle $n\alpha$ around l, the result is never the origin; i.e., let $R_{l,\theta}p$ be the result of rotating p around l by θ. Then $R_{l,\theta}0, R_{l,2\theta}0, R_{l,3\theta}0, \ldots$ is a countably infinite sequence of points that are all inside B^3.

We find a suitable α by partially solving the equation $R_{l,n\theta}0 = 0$. In particular, we showed that this requires that $\cos(n\theta) = 1$, and this means that θ must have the form $\theta = \frac{2\pi k}{n}$, where n is a positive integer and $k \in \mathbb{Z}$. Similar to the construction in Section 5.3, the set of possible θ can be enumerated, so there must be at least one angle $\alpha \in [0, 2\pi)$ that is *not* one of the θ. Thus, $R_{l,n\alpha}0 \neq 0$ for any positive integer n.

Exactly as before, let $Z = \{R_{l,n\alpha}0 \mid n \in \mathbb{N}\}$. Then $B^3 - \{0\} = (B^3 - Z) \bigsqcup R_{l,\alpha}Z$. This is then used to show that

$$B^3 = \big((B^3 - \{0\}) \cap (B^3 - Z)\big) \bigsqcup R_{l,-\alpha}\big((B^3 - \{0\}) \cap Z\big).$$

And just as before, replacing $(B^3 - \{0\})$ with the decompositions found above yields a decomposition of all of B^3 that satisfies the Banach-Tarski paradox. Listing 8 shows the decompositions of B^3 in ACL2(r) where *b3* is the recognizer of the set B^3.

Listing 8 Decompositions of B^3 in ACL2(r).

```
;; Unit ball broken down into 52 sets
(defthmd b3-equiv-1
  (iff (b3 p)
       (or (b3-00 p)
           (b3-01 p)
           ...
           ...
           (b14-00 p)
           (b14-01 p)
           (set-b20 p)
           (set-b10 p)
           (rota-1-b3-10 p)
           ...
           ...
           (rota-1-b14-11 p)
           (rota-1-b21 p)
           (rota-1-b11 p)))
  :hints ...)

;; A copy of the unit ball
(defthmd b3-equiv-2
  (iff (b3 p)
       (or (rot-3-b3-00 p)
           (rot-3-b3-10 p)
           ...
           ...
           (rot-8-b8-00 p)
           (rot-8-b8-10 p)
           (rota-1-rot-3-b3-11 p)
           (rota-1-rot-3-b3-01 p)
           ...
           ...
           (rota-1-rot-8-b8-11 p)
           (rota-1-rot-8-b8-01 p)))
  :hints ...)

;; Another copy of the unit ball
(defthmd b3-equiv-3
  (iff (b3 p)
       (or (rot-9-b9-00 p)
           (rot-9-b9-10 p)
           ...
           ...
           (rot-14-b14-00 p)
           (rot-14-b14-10 p)
           (rota-1-rot-9-b9-11 p)
           (rota-1-rot-9-b9-01 p)
           ...
           ...
           (rota-1-rot-14-b14-11 p)
           (rota-1-rot-14-b14-01 p)))
  :hints ...)
```

7 Conclusion

In this paper we have presented a formalization of the Banach-Tarski theorem in ACL2(r). Although ACL2(r) may not be the obvious choice to formalize such an abstract theorem, it turns out that the key step in the proof is reasoning about free groups, and since this is tantamount to reasoning about lists, it is perfectly natural for theorem provers in the Boyer-Moore family of provers, like ACL2(r). Moreover, even though there is very limited support for quantification in ACL2(r), we have shown that we can define complex structures and prove properties about them. The proof also makes use of 3D rotations, and we formalized these rotations and proved many key properties about them. We have formalized the proof for the Cartesian product of two countable sets is countable, and used this proof in the decomposition of S^2 and B^3. The proven properties of 3D rotations from section 4 and countable sets are readily available to use by anyone who chooses to do so, and these are available in rotations.lisp and countable-sets.lisp in the ACL2(r) source files. We have used many properties of modular arithmetic and trigonometric functions, and these were previously formalized in ACL2(r). Also critical in a few steps was the fact that certain sets can be enumerated, but that no non-trivial interval of reals can be – and this had also been proved in prior work. The end result is a proof of the Banach-Tarski paradox: The unit ball B^3 in $\mathbb{R}^3$ can be decomposed into finitely many pieces that can be rotated and reassembled to form two copies of B^3.

References

1 Piergiorgio Bertoli and Paolo Traverso. *Design Verification of a Safety-Critical Embedded Verifier*, pages 233–245. Kluwer Academic Publishers, USA, 2000.

2 John Cowles and Ruben Gamboa. Using a first order logic to verify that some set of reals has no lesbegue measure. In *International Conference on Interactive Theorem Proving*, pages 25–34. Springer, 2010.

3 Ruben Gamboa. *Mechanically Verifying Real-Valued Algorithms in ACL2*. PhD thesis, Citeseer, 1999.

4 Ruben Gamboa and John Cowles. A Cantor Trio: Denumerability, the Reals, and the Real Algebraic Numbers. In *International Conference on Interactive Theorem Proving*, pages 51–66. Springer, 2012.

5 Ruben Gamboa, John Cowles, and JV Baalen. Using ACL2 Arrays to Formalize Matrix Algebra. In *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2'03)*, volume 1, 2003.

6 Ruben A. Gamboa and Matt Kaufmann. Nonstandard Analysis in ACL2. *J. Autom. Reason.*, 27(4):323–351, November 2001. doi:10.1023/A:1011908113514.

7 Thomas J Jech. *The Axiom of Choice*. Courier Corporation, 2008.

8 J. Strother Moore. Milestones from the pure lisp theorem prover to acl2. *Form. Asp. Comput.*, 31(6):699–732, December 2019. doi:10.1007/s00165-019-00490-3.

9 Rotation matrix. Rotation matrix – Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Rotation_matrix, 2021. Online; Accessed: 2022-02-04.

10 Madeline Tremblay. The Banach-Tarski Paradox. unpublished, 2017.

11 Tom Weston. The Banach-Tarski Paradox. *Citado*, 2:15, 2016.

Dandelion: Certified Approximations of Elementary Functions

Heiko Becker
MPI-SWS, Saarland Informatics Campus (SIC), Germany

Mohit Tekriwal
University of Michigan, Ann Arbor, MI, USA

Eva Darulova
Uppsala University, Sweden

Anastasia Volkova
Nantes Université, France

Jean-Baptiste Jeannin
University of Michigan, Ann Arbor, MI, USA

Abstract

Elementary function operations such as sin and exp cannot in general be computed exactly on today's digital computers, and thus have to be approximated. The standard approximations in library functions typically provide only a limited set of precisions, and are too inefficient for many applications. Polynomial approximations that are customized to a limited input domain and output accuracy can provide superior performance. In fact, the Remez algorithm computes the best possible approximation for a given polynomial degree, but has so far not been formally verified.

This paper presents *Dandelion*, an automated certificate checker for polynomial approximations of elementary functions computed with Remez-like algorithms that is fully verified in the HOL4 theorem prover. Dandelion checks whether the difference between a polynomial approximation and its target reference elementary function remains below a given error bound for all inputs in a given constraint. By extracting a verified binary with the CakeML compiler, Dandelion can validate certificates within a reasonable time, fully automating previous manually verified approximations.

2012 ACM Subject Classification Software and its engineering → Formal software verification

Keywords and phrases elementary functions, approximation, certificate checking

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.6

Supplementary Material *Software (Source Code)*: `https://github.com/HeikoBecker/Dandelion` archived at `swh:1:dir:2098ee54478f916e1898fb4732b0b4156f2d94d8`

Funding This work was supported in part by the University of Michigan.

Acknowledgements The authors would like to thank John Harrison for the insightful discussion and for providing the source code for his paper that inspired the Dandelion work. Further, we thank Magnus Myreen and Michael Norrish for their help with improving the HOL4 implementation of Dandelion. We also thank Samuel Coward for helping us with the MetiTarski evaluation. Finally, we thank the anonymous ITP reviewers for their feedback on the paper.

1 Introduction

Exact computation in real-number arithmetic is in general too inefficient for most applications [5], and is thus typically replaced by finite-precision (floating-point or fixed-point) arithmetic. While arithmetic operations such as addition and multiplication are well-supported and efficient, real-world code often also needs to support elementary functions such as sin and exp. Such functions cannot be computed exactly on today's digital hardware and thus necessarily have to be approximated. For floating-point arithmetic, libraries provide

general-purpose approximations for a limited set of formats, e.g. correctly rounded to single or double precision [27, 13]. However, these can be inefficient for applications that do not need quite as much accuracy, or that only need to work for a limited set of inputs [14, 25]. Furthermore, many applications, especially in the embedded systems domain, operate with fixed-point arithmetic, for which efficient library approximations do not exist [24].

When library function implementations are suboptimal, a possible solution is to generate approximations of elementary functions on demand with custom accuracy – exactly the accuracy that is needed by the application and its context. Indeed, automated algorithms exist for generating polynomial approximations of elementary functions with a given polynomial degree or given bound on the *approximation error*. For instance, Remez-like algorithms [36] generate the best polynomial approximation, i.e. the one with the smallest approximation error for a given polynomial degree. We can, for example, approximate exp on $[0, 0.5]$ by $0.999 + 1.001 \times x + 0.484 \times x^2 + 0.215 \times x^3$ with an approximation error of 2.63×10^{-5}.

Remez-like algorithms are used in several automated tools for generating custom approximations [8, 25], however, these implementations are not formally verified. This is especially problematic because the algorithms are generally tricky to get right [32][1].

In this paper, we implement and prove correct *Dandelion*, a fully automated and formally verified certificate checker for polynomial approximations computed by Remez-like algorithms. Dandelion is implemented and fully verified inside the HOL4 theorem prover [37]. A certificate for Dandelion consists of an elementary function f, an input interval I, and a polynomial approximation p and an approximation error ϵ returned by a Remez-like algorithm. We prove once and for all the correctness theorem that if Dandelion returns true for a certificate, the encoded error is a true upper bound to the difference between the elementary function and the encoded polynomial: $\max_{x \in I} |f(x) - p(x)| \leq \epsilon$.

Dandelion's certificates are minimal, requiring only inputs and outputs of the approximation algorithm to be recorded. Additionally, Dandelion certifies the known best possible approximations of Remez-like algorithms, together, making it widely applicable. Previous work focused on manual proofs [18]; certifies only results of Chebyshev approximations, which are not as accurate as those computed by Remez-like algorithms [6]; or their verification-technique is mainly based on interval arithmetic [28]. Dandelion is the first tool that automates the approach of Harrison [18], and thus the key challenge that Dandelion solves is *automation*; Dandelion requires no user interaction, making it the first fully automated validator for results of Remez-like algorithms based on polynomial zero finding.

One may think that verifying an implementation of a Remez-like algorithm should be favored over validating each run separately. However, correctness proofs for one implementation generally do not apply to other implementations, and thus would have to be re-done with every change. In contrast, by certifying only the end-result, Dandelion is indifferent to the implementation choices and thus immediately more widely applicable.

Harrison [18] has manually verified a polynomial approximation of the exponential function in HOL-Light [20]. The methodology presented is general, but was never automated. Dandelion borrows the high-level approach from Harrison's manual proof, automating the key ideas to validate results of Remez-like algorithms.

While the idea of automating an existing development may seem simple, we faced two major challenges to make automated certification practical. First, computations in theorem provers are generally slower than those in unverified tools, making certain designs impractical. Second, some definitions of Harrison use non-computable functions and thus cannot be

[1] Muller warns: "[..] even if the outlines of the [Remez] algorithm are reasonably simple, making sure that an implementation will always return a valid result is sometimes quite difficult." ([32], page 52).

used in an automated approach. To speed-up the computations, we extract Dandelion as a verified binary using the CakeML compiler [39]. The extracted binary enjoys the same correctness guarantees as our in-logic implementation, and makes checking a certificate fast: a single certificate is checked on average within 6 minutes. We overcome the problem of non-computable functions by identifying computable versions and proving equivalence between the computable and non-computable functions.

Dandelion can be used as a verifier for any Remez-like algorithm. In our evaluation, we use Dandelion to certify a number of approximations generated from FPBench [12] and the work by Izycheva et al. [24]. Our evaluation shows that certificate checking in Dandelion is fast, and that Dandelion certifies, for an elementary function f and polynomial p, approximation errors on the same order of magnitude as the infinity norm ($\max_{x \in I(x)} |f(x) - p(x)|$). We also encode the original proof-goal of Harrison as a certificate – Dandelion reduces its proof to a single line of code.

Contributions

In summary, this paper provides the following contributions:

- a HOL4 implementation of Dandelion[2], a verified certificate checker for polynomial approximations
- a verified binary extracted using CakeML to make certificate checking fast, and
- an evaluation of Dandelion's performance on a set of benchmarks, comparing it with the state-of-the-art.

2 Overview

Before we dive into the technical details of Dandelion, we give an overview of our toolchain and the proofs that Dandelion performs automatically using the example in Figure 1. The starting point is the code in Figure 1a that converts polar to cartesian coordinates, and returns the resulting x component. This code could for example be part of an autonomous car or a drone, and inaccuracies in the conversion of coordinates may have catastrophic effects [33]. Chip sizes and energy budgets in these devices are usually small, and thus using a fully-fledged floating-point unit is not always possible. As an alternative, code is often implemented in fixed-point arithmetic, which, however, does not come with standard and efficient library implementations of elementary functions [24]. Hence, an engineer may approximate the function cos on line 8 in Figure 1a with a custom polynomial approximation shown in Figure 1b, for instance using the state-of-the-art synthesis tool Daisy [24][3].

Daisy internally calls a Remez-like algorithm to generate the polynomial approximation of cos, but the approximation algorithm and Daisy itself are not (formally) verified. With Dandelion, we can straight-forwardly instrument Daisy to generate the certificate shown in Figure 1c that encodes the elementary function to be approximated (`f`), the approximating polynomial (`p`), the approximation error (ε), and the range on which the approximation is supposed to be valid (`I`), and an additional parameter `n` which we explain later. Note that the input interval `I` recorded in the certificate captures the direct inputs to the elementary function cos and is thus different from the input interval in the `require` clause that captures inputs to the overall function `polToCart_x`. Dandelion validates this certificate in 31 seconds and proves the HOL4 theorem

[2] The source code of Dandelion is publicly available at `https://github.com/HeikoBecker/Dandelion`.
[3] Daisy can also synthesize suitable finite-precision types for the polynomial (not shown here), and generate certificates that formally verify the *roundoff* error bound of this polynomial implementation [4].

```
def polToCart_x(radius: Real,
  theta: Real): Real = {
 require(((1.0 <= radius) &&
  (radius <= 10.0) && (0.0 <= theta) &&
  (theta <= 360.0)))
 val pi = 3.14159265359
 val radiant = (theta * (pi / 180.0))
 (radius * cos(radiant))
}
```

(a) Example kernel using a elementary function.

```
def polToCart_x(radius: Fixed,
 theta: Fixed): Fixed = {
 val pi = 3.14159265359
 val radiant = (theta * (pi / 180.0))
 val _tmp = (1.3056366443634033 +
  (radiant * (-1.2732393741607666 +
  (radiant * (0.2026423215866089 +
  (radiant * 3.3222216089257017e-09))))))
 (radius * _tmp)
  }
```

(b) Example with polynomial approximation for cos.

```
cos_cert = <| f := Fun Cos (Var "radiant"); n := 32;
  (* p (x) ~ 1.305 - 1.273 * x + 0.202 * x^2 + 3.322 * 10^-9 * x^3 *)
  p := [5476237/4194304; -5340353/4194304; 1699887/8388608; 3740489/1125899906842624];
  ε := 766133524584849981160987377038947873961143126798 7/(25 * 10^48); (* ~0.306 *)
  I := [("radiant", (0, 314159265359/50000000000))]; (* ~ x in [0, 6.284] *) |>
```

(c) Certificate for the approximation of cos in the example.

Figure 1 Example kernel using a elementary function (top-left), the kernel with a polynomial approximation (top-right), and the certificate for Dandelion (bottom).

▶ **Theorem 1.** $\forall x.\ x \in I(x) \Rightarrow |\cos(x) - p(x)| \leq \varepsilon$

If the approximation error had not been correct, the binary would emit an error message, explaining which part of the validation failed.

The certificate in Figure 1c uses only a single elementary function. In general, Dandelion supports more complicated elementary function expressions, like $\exp(x \times \frac{1}{2})$, and $\sin(x - 1) + \cos(x + 1)$. Like Remez-like algorithms, we only require the functions to be univariate, i.e. the certificate can only have a single free variable. Any approximation tool that can generate these certificates can be used to generate inputs for Dandelion, and Dandelion can be used independently of a particular approximation algorithm implementation.

The approach used by Dandelion has been laid out previously in a manual proof for the exponential function by Harrison [18] (Subsection 2.1 overviews the main theorems and ideas). The presented high-level approach is general, but a key challenge that Dandelion solves is to automate each step and extend them to more complex expressions (Subsection 2.2).

2.1 Manual Proof by Harrison

Harrison has manually verified an approximation by Tang [40] of the exponential function, showing that: $\forall x.x \in [-0.010831, 0.010831] \Rightarrow |((e^x - 1) - p(x)| \leq 2^{-33.2}$. The manual verification by Harrison is split into two steps. First, Harrison simplifies the overall proof goal to a proof about polynomials, by replacing $e^x - 1$ with a high-accuracy truncated Taylor series q. By truncating the series after the 7th term, the approximation error of the series becomes 2^{-58} and the overall proof-goal is reduced from $|((e^x - 1) - p(x)| \leq 2^{-33.2}$ with the triangle inequality to

$$|q(x) - p(x)| \leq 2^{-33.2} - 2^{-58}. \tag{1}$$

The difference between $q(x)$ and $p(x)$ itself is a polynomial $h(x)$, and thus this first step reduces the overall proof goal to proving an upper bound on the polynomial h. As h represents the difference between two polynomial approximations, the points where h attains its maximum value (i.e. its extremal points) are those where the approximation error is the largest. It thus suffices to reason about the extremal points of h for proving the inequality.

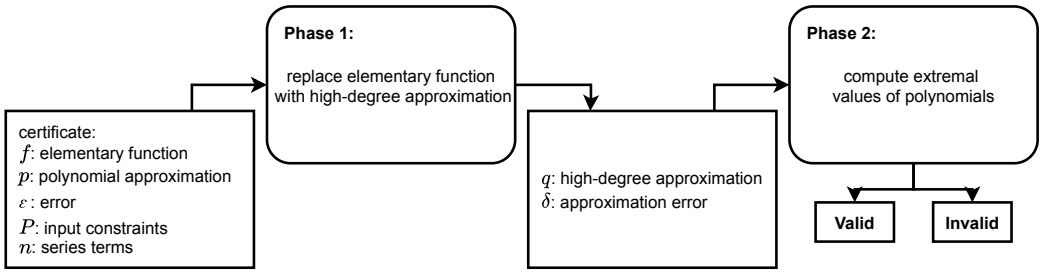


Figure 2 Overview of Dandelion toolchain.

To prove the polynomial inequality, Harrison proved two well-known mathematical theorems in HOL-Light. The first theorem proves that polynomial p on the closed interval $[a, b]$ attains its extremal values either at the outer points or at the points where the first derivative is zero:

▶ **Theorem 2.** *Let p a differentiable, univariate polynomial, defined on $[a, b]$ and M a real number, then*

$$|\,p(a)\,| \leq M \wedge |\,p(b)\,| \leq M \wedge$$
$$(\forall x.\ a \leq x \leq b\ \wedge\ p'(x) = 0 \Rightarrow |\,p(x)\,| \leq M) \Rightarrow$$
$$(\forall x.\ a \leq x \leq b \Rightarrow |\,p(x)\,| \leq M).$$

The second theorem is called *Sturm's theorem* and proves that the exact number of zeros of a polynomial can be computed from the so-called Sturm sequence of polynomials:

▶ **Theorem 3.** *Let p a differentiable, univariate polynomial, defined on $[a, b]$. If p has non-zero values on both a and b, and its derivative is not the constant zero function, then we call Sturm ss the sturm sequence for p, and the set of zeros of p has size $V(a, ss) - V(b, ss)$.*

The Sturm sequence ss of polynomial p is defined recursively as

$$ss_0 = p \qquad ss_1 = p' \qquad ss_{i+1} = -\mathrm{rem}\,(ss_{i-1}, ss_i)$$

where rem computes the remainder of the polynomial division $\frac{ss_{i-1}}{ss_i}$. Computation stops once the remainder becomes the constant 0 polynomial. Function $V(a, ss)$ in Theorem 3 computes the number of sign changes when evaluating the polynomials in the list ss on value a.

To prove the final inequality, Harrison computes unverified guesses for both the Sturm sequence of $h'(x)$ and the zeros of $h'(x)$ using Maple, and manually validates them in HOL4 using Theorem 3. By knowing the number of zeros, and their values, Harrison then provably derives an upper bound on the extremal values of polynomial h using Theorem 2.

2.2 Automated Proofs in Dandelion

As in Harrison's approach, Dandelion splits the proof into two parts. In the first phase, Dandelion replaces elementary functions in the certificate by high-accuracy approximations computed inside HOL4. The second phase then proves that the approximation error is a correct upper bound on the extremal points of the resulting polynomial by finding zeros of the derivative and bounding the number of zeros with Sturm sequences. The key differences between Dandelion and the proof by Harrison is that Dandelion supports more elementary functions, i.e. $\exp$, $\sin$, $\cos$, $\ln$, and $\tan^{-1}$, and that its certificate checking is fully automated

and does not require any user interaction or additional proofs. Figure 2 gives an overview of the automatic computations done by Dandelion. We explain them at a high-level for our example from Figure 1.

To prove the overall correctness theorem (Theorem 1), Dandelion first computes a computable high-accuracy polynomial approximation for cos, denoted by q, using a truncated Taylor series of degree `n`, with an approximation error of `3.77e-3`. Generally, the certificate parameter `n` defines the number of series terms computed for the truncated Taylor series in Dandelion. Exactly as for Harrison's manual proof, Dandelion proves an upper bound on the difference between q and the target polynomial approximation p (Equation 1). Coming up with a general approach for computing accurate truncated Taylor series of arbitrary elementary functions was a major challenge for Dandelion – we implemented a library of general purpose Taylor series for the supported elementary functions. Given a target degree, Dandelion automatically computes a polynomial implementation and proves an approximation error for the truncated series. We explain the first phase in more detail in Section 3.

In the second phase, Dandelion computes an upper bound on the polynomial $h(x) = q(x) - p(x)$ exactly like Harrison, by reasoning about the zeros of its derivative h', using Sturm sequences to bound the number of zeros of h'. Based on the number of zeros, Dandelion uses an (unverified) oracle to automatically come up with a list of zeros of h'.

To prove the final bound on h, Dandelion checks for each zero of h' that the value of h at this point is smaller than or equal to the residual error $\varepsilon -$ `3.77e-3`. Harrison's definition of Sturm sequences is defined as a non-computable predicate, involving an existentially quantified definition of polynomial division. Key to Dandelion is the implementation of a computable version of polynomial division, as well as Sturm sequences, in combination with equivalence proofs relating them to Harrison's predicates. We explain how Dandelion computes the Sturm sequences automatically and how Dandelion estimates zeros of polynomials in more detail in Section 4.

Computing the Sturm sequence is the most computationally expensive part of Dandelion, which we found to be impractical to do in logic. Thus we extract a verified binary using the CakeML compiler for the second phase of Dandelion only. For the extraction to work, we translated the HOL4 definitions of the second phase into CakeML source code via CakeML's proof-producing synthesis tool [1]. We explain the extraction with CakeML in Section 5.

3 Automatic Computation of Truncated Taylor Series

As in Harrison's manual proof, Dandelion replaces in a first step the elementary function in the certificate by a high-accuracy polynomial approximation. The crucial difference is that Dandelion automates all of the manual steps, which we explain next.

When checking a certificate for function f, with input range I, Dandelion automatically replaces every occurrence of an elementary function in f with a truncated Taylor series $t_{f,n}$. Below, we take f to be a single elementary function and will discuss the extension to more complicated elementary function expressions later. The parameter n of $t_{f,n}$ is part of the certificate and specifies the number of terms computed for the truncated series, i.e. if n is 32, Dandelion truncates the Taylor series of f after the 32nd term. The final result of the phase is a high-accuracy polynomial approximation of function f, $q_{f,n}$, and an overall approximation error $\delta_{f,n}$. For the simple case where f is a single elementary function, $q_{f,n}$ and $t_{f,n}$ are the same. Once we extend Dandelion to more complicated expressions, Dandelion combines different instances of $t_{f,n}$, into the final $q_{f,n}$, as we explain later. We implement the first phase in a HOL4 function `approxAsPoly` and prove soundness of `approxAsPoly` once and for all in HOL4:

▶ **Theorem 4** (First Phase Soundness).

$$\texttt{approxAsPoly}\ f\ I\ n = \mathit{Some}(q_{f,n}, \delta_{f,n}) \Rightarrow (\forall x.\ x \in I \Rightarrow |\, f(x) - q_{f,n}(x)\,| \leq \delta_{f,n})$$

The theorem states that if `approxAsPoly` succeeds and returns $q_{f,n}$ and $\delta_{f,n}$, then the approximation error on input range I between $f(x)$ and $q_{f,n}(x)$ is upper bounded by $\delta_{f,n}$.

In the rest of this section, we first explain how Dandelion automatically computes truncated Taylor series for elementary functions like sin and exp, then we explain how Dandelion extends this approach in `approxAsPoly` to compute a single polynomial approximations of more complicated elementary function expressions like $\exp(x * \frac{1}{2})$ via interval analysis and propagation of polynomial errors. Throughout this section, we use f to refer to the elementary function from the certificate, $t_{f,n}$ as truncated Taylor series, $\delta_{t,n}$ as the approximation error of the series, $q_{f,n}$ as polynomial approximation, and $\delta_{f,n}$ as the overall approximation error.

3.1 Truncated Taylor Series for Single Elementary Functions

Both $t_{f,n}$ and $\delta_{t,n}$ depend on the approximated elementary function f, as well as the number of series terms n from the certificate. Overall, Dandelion automatically computes a truncated Taylor series for the elementary functions sin, cos, exp, $\tan^{-1}$, and ln[4]; the series expansions for exp and ln already existed in HOL4 prior to Dandelion and we port the series for $\tan^{-1}$ from HOL-Light. For sin and cos we prove series based on textbook descriptions. Formally, Dandelion proves a truncated Taylor series for each elementary function once and for all as

▶ **Theorem 5.** $\forall x\ n.\ Pre(x) \Rightarrow f(x) = \sum_{i=0}^{n}(\frac{f^i\ 0}{i!} * x^i) + \delta_{t,n}(x)$

Here, f^i is the i-th derivative of f, and the approximation error $\delta_{f,n}(x)$ is soundly bounded from the remainder term of Taylor's theorem for input value x. Predicate Pre is a precondition constraining the interval on which function f can be approximated by the truncated series. When approximating an elementary function f by its truncated series, Dandelion always ensures that this precondition Pre is true: The series for exp requires inputs to be non-negative, the series for ln requires arguments greater than 1, and $\tan^{-1}$ requires arguments in $(-1, 1)$. The series for sin and cos have no preconditions.

At certificate checking time Dandelion automatically computes an upper bound to the approximation error $\delta_{t,n}(x)$. Further, the second phase of Dandelion operates on polynomials following Harrison's formalization. Therefore, we prove once and for all that the truncated series from Theorem 5 can be implemented in Harrison's polynomial datatype:

▶ **Theorem 6.** $\forall n. \sum_{i=0}^{n}(\frac{f^i\ 0}{i!} * x^i) = t_{f,n}\ (x)$

Theorem 6 proves that t_n implements the truncated Taylor series on the left-hand side for an arbitrary number of approximation steps n. We prove versions of Theorem 6 for each elementary function supported by Dandelion. Finally, the proof of First Phase Soundness (Theorem 4) for a single elementary function is a simple combination of Theorems 5 and 6.

3.2 Approximations of More Complicated Expressions

Next, we explain how Dandelion uses truncated Taylor series to approximate more complicated elementary function expressions, using $\exp(y \times \frac{1}{2}) - 1$ on the interval $[1, 2]$ as an example[5]. In general, a Remez-like algorithm can return an approximation for a compound function or an

[4] Dandelion currently does not support *tan*, as a straight-forward reduction to sin(x)/cos(x) did not work out. We plan to incorporate a more direct series from HOL-Light in the future.

[5] Currently, Dandelion does not generally support divisions, hence we represent $\frac{y}{2}$ as $y \times \frac{1}{2}$ explicitly.

expression, as long as it stays univariate. Compared to approximating individual functions, e.g. exp, an overall expression approximation can be more accurate, sometimes avoiding undesirable effects such as cancellation. Hence, Dandelion should also be able to certify those.

From Theorems 5 and 6 Dandelion knows how to automatically compute a polynomial approximation $t_{\exp,n}$ and an approximation error $\delta_{\exp,n}(x)$ for the exponential function for a given input range on the argument. In our example, the input argument is $y \times \frac{1}{2}$, and thus the value of $\delta_{\exp,n}(x)$ depends on the range of this expression, which Dandelion computes automatically using interval arithmetic [30].

As interval analysis, we reuse an existing HOL4 formalization [4], and extend it with range bounds for elementary functions. For our example Dandelion also needs to compute a range bound for $\exp(y \times \frac{1}{2})$. In general, because elementary functions are defined non-computably in HOL4, we have to rely on a trick to compute interval bounds. To compute interval bounds for elementary functions, Dandelion reuses our formalized truncated Taylor series. From Theorem 5 and Theorem 6, we derive for f that

$$|f(x) - t_{t,n}(x)| \leq \delta_{t,n}. \tag{2}$$

From this inequality, we derive a bound on $f(x)$ in the interval $[a, b]$

$$t_{t,n}(a) \leq f(x) \leq t_{t,n}(b) + \delta_{t,n}. \tag{3}$$

Equation 3 holds for monotone f only, and thus we cannot apply it to sin and cos as they are periodic. For both functions, interval analysis returns the closed interval $[-1, 1]$. Dandelion's interval analysis is proven sound once and for all in HOL4.

With the interval analysis, we can soundly compute a polynomial approximation for exp, $t_{\exp,n}$ on the range of $y \times \frac{1}{2}$. Dandelion automatically composes the polynomial $y \times \frac{1}{2}$ with $t_{\exp,n}$ to obtain a polynomial $q_{\exp(y\times\frac{1}{2}),n}$ with approximation error $\delta_{\exp,n}(x)$. However, we still need to come up with a polynomial approximation p and an approximation error for the full function $\exp(y \times \frac{1}{2}) - 1$. In our example, Dandelion treats the constant 1 as a polynomial returning 1, and automatically computes the polynomial difference of $q_{\exp(\ldots),n}$ and $q_{1,n}$. The global approximation error δ for the difference of $q_{\exp(\ldots),n}$ and $q_{1,n}$ depends on the approximation errors accumulated in both polynomials. In a final step, Dandelion automatically computes an upper bound on the global approximation error by propagating accumulated errors through the subtraction operation.

Generally, Dandelion implements an automatic approximation error analysis inside function `approxAsPoly` that propagates accumulated approximation errors. The propagation is implemented for basic arithmetic, and elementary functions to support e.g. expressions like $\exp(x) + \sin(x - 1)$.

Computing Propagation Errors for Sin and Cos

To accurately propagate approximation errors through sin and cos, our soundness proof assumes that the accumulated approximation error is contained in the interval $[0, \frac{\pi}{2}]$. This does not pose a true limitation of Dandelion as errors larger than $\pi/2$ would anyway be undesirable and impractical. For the correctness proof of `approxAsPoly` (Theorem 4), however, Dandelion must automatically prove that accumulated errors are less than or equal to $\pi/2$. This poses a challenge as in HOL4 π is defined non-computably using Hilbert's choice operator: if $0 \leq x \leq 2$ and $\cos(x) = 0$, then π is $2 \times x$. To solve this problem, we reuse the truncated Taylor series of $\tan^{-1}$ and the fact that $\tan^{-1}(1) = \pi/4$ to compute a lower bound r in HOL4, where $r \leq \pi$. At certificate checking time, when propagating the error $\delta_{f,n}$ through sin and cos, Dandelion checks $\delta_{f,n} \leq \frac{r}{2}$, which by transitivity proves that $\delta_{f,n} \leq \frac{\pi}{2}$.

3.3 Extending Dandelion's First Phase

All of the truncated Taylor series proven in Dandelion are for single applications of an elementary function. For a particular application it may be beneficial to add special cases to compute a single, more accurate, truncated Taylor series of an elementary function like $\exp(\sin(x))$ instead of computing a truncated series for each function separately.

In Harrison's original proof this would require manually redoing a large chunk of the proof work whereas for Dandelion such an extension amounts to 4 steps: Proving the truncated Taylor series as in Theorem 5, implementing and proving correct the polynomial $t_{f,n}$ as in Theorem 6, extending `approxAsPoly` with the special case for the new elementary function, and finally using the theorems proven for the first two steps to extend First Phase Soundness (Theorem 4) with a correctness proof for the new case. Complexity of the proofs only depends on the complexity of the series approximation. Dandelion then automatically uses the new series approximation whenever the approximated function is encountered in a certificate, and the global soundness result of Dandelion still holds without any required changes. The second phase directly benefits from adding additional approximations as more accurate Taylor series decrease the approximation error of the first phase.

4 Validating Polynomial Errors

For a certificate consisting of an elementary function `f`, polynomial approximation `p`, approximation error ε, input constraints `I`, and truncation steps `n`, the first phase of Dandelion computes a truncated Taylor series $q_{f,n}$ and an approximation error $\delta_{f,n}$, which is sound by Theorem 4. Both $q_{f,n}$ and `p` are polynomials, and following Harrison's terminology, we refer to their difference $q_{f,n}(x) - \mathtt{p}(x)$ as the error polynomial $h(x)$. In the second phase, Dandelion automatically finds an upper bound to the extremal values of $h(x)$ and compares this upper bound to the residual approximation error $\varepsilon - \delta_{f,n}$, which we refer to as γ. We prove soundness of the second phase once and for all as a HOL4 theorem:

▶ **Theorem 7** (Second Phase Soundness).

$$\forall x.\ x \in \mathtt{I}(x) \Rightarrow |\, q_{f,n}(x) - \mathtt{p}(x) \,| \leq \gamma$$

Before going into the details of how Dandelion automatically validates the residual error γ, we quickly recall the key real analysis result which we rely on in this phase: on a closed interval $[a, b]$, a differentiable polynomial p can reach its extremal values at the outer points $p(a)$, $p(b)$, and the zeros of p's first derivative p' (Theorem 2). To find the extremal values of $h(x)$, Dandelion thus needs to automatically find *all* zeros of $h'(x)$.

Dandelion splits finding the extremal values and validating γ into three automated steps:

1. Compute the number of zeros using Sturm's theorem (Theorem 3 in Section 2)
2. Validate a guess of the zeros computed by an unverified, external oracle
3. Compute an upper bound on extremal values (using the validated zeros) and compare with γ

Conceptually, the second phase automates the main part of Harrison's manual proof, and the key step is computing Sturm sequences automatically in the first step. Next, we explain the ideas behind automating each of the steps.

```
sturm_seq (p, q, n) =
  if n = 0 then
    if (rm (p, (1/q[deg q]) * q) = 0 ∧ q <> 0) then SOME []
    else None
  else let g = - (rm (p, (1/q[deg q]) * q)) in
   if g = 0 ∧ ~ q = 0 then Some []
   else if (q = 0 ∨ (deg q < 3)) then None
   else case sturm_seq (q, g, n-1) of
     None => None
     |Some ss => Some (g::ss)
```

Figure 3 HOL4 definition of Sturm sequence computation.

4.1 Bounding the Number of Zeros of a Polynomial

Dandelion bounds the number of zeros of a polynomial using Sturm's theorem (Theorem 3). A key challenge in developing this part of Dandelion was ensuring that the Sturm sequence is computable inside HOL4. In HOL-Light, Harrison defines Sturm sequences as a non-computable predicate `STURM` that existentially quantifies results, and thus can only be used to validate results in a manual proof.

In Figure 3, we show how Dandelion computes Sturm sequences. Function `rm (p,q)` computes the remainder of the polynomial division of `p` by `q`, `deg p` is the degree of polynomial `p`, and `q[n]` is the extraction of the `n`-th coefficient of `q`. As each polynomial division operation decreases the degree of the result by at least 1, the Sturm sequence for a polynomial p has a maximum length of $\deg(p) - 1$, as computation starts with p and its first derivative p'. Function `sturm_seq` is therefore initially run on polynomial $h'(x)$, $h''(x)$, and $\deg(h') - 1$.

If `sturm_seq(deg h'-1, h', h'')` returns list `sseq`, the complete Sturm sequence is `h'::h''::sseq`, and Dandelion computes the number of zeros of e' as its variation on the input range, based on Theorem 3.

We have proven once and for all that the results obtained from `sturm_seq(n, p', p'')` satisfy Harrison's non-computable predicate `STURM`. Thus we can reuse Harrison's proof of Sturm's theorem (Theorem 3). Harrison's Sturm sequences also use on a non-computable predicate for defining the result of polynomial division, and we prove it equivalent to a computable version in Dandelion, inspired by the one provided by Isabelle/HOL [23]. Ultimately, Dandelion uses these two equivalence proofs to reuse Harrison's proof of Sturm's theorem which we ported from HOL-Light.

4.2 Finding Zeros of Polynomials

Given the numbers of zeros nz for $h'(x)$, Dandelion next finds their values. As zero-finding is highly complicated even in non-verified settings, Dandelion uses an external oracle to come up with an initial guess of the zeros. These initial guesses are presented as a list of confidence intervals $[a, b]$, where $h'(x)$ has a zero between a and b. Further, the algorithm computing the confidence intervals need not be verified, as the result can easily be validated by Dandelion. To validate a list of guesses Z, Dandelion again relies on a result from real-number analysis, proven by Harrison: For a confidence interval $[a, b]$, function f has a zero in the interval, if its first derivative f' changes sign in the interval $[a, b]$.

Dandelion validates the confidence intervals Z using a computable function that checks automatically for each element $[a, b]$ in Z, that $h''(a) \times h''(b) \leq 0$, which is equivalent to a sign-change in the interval. If the number of zeros found is nz, Dandelion checks that this sign change occurs at least nz times in Z.

While we do prove our approach for finding zeros of polynomials sound, Dandelion is necessarily incomplete. One known source for incompleteness are so-called multiple roots as they occur e.g. in $p(x) = (x-1)^2$. Harrison's formalization implicitly relies on the polynomial being squarefree and Dandelion inherits this limitation. This issue could potentially be addressed using the approach of Li et al. [26] though it would require reproving Sturm's theorem for non-squarefree polynomials.

4.3 Computing Extremal Values

In the final step, Dandelion uses the validated confidence intervals Z which contain all zeros of $h'(x)$ to compute an upper bound to the extremal values of $h(x)$. For interval $[a, b]$, Dandelion would ideally bound the error of $h(x)$ in $[a, b]$ as the maximum of $h(a)$, $h(b)$, and $h(y)$, where y is a zero of $h'(x)$. However, we have only confidence intervals for the zeros, and not their exact values available. Therefore, Dandelion's computation of an upper bound to $e(x)$ is more involved, and we base it on a theorem of Harrison. Harrison's theorem is a generalization of Theorem 2 for polynomial p, with derivative p', on interval $[a, b]$:

▶ **Theorem 8.**

(1) $(\forall x.\ a \le x \le b \land f'(x) = 0 \Rightarrow$

(2) $(\forall x.\ a \le x \le b \Rightarrow |\, p'(x)\,| \le B) \land$

(3) $(\forall [u, v].\ [u, v] \in Z \Rightarrow a \le u \land v \le b \land |\, u - v\,| \le e \land |\, f(u)\,| \le K) \Rightarrow$
$\forall x.\ a \le x \le b \Rightarrow |\, p(x)\,| \le \max(|\, f(a)\,|, |\, f(b)\,|, K + B \times e)$

The theorem can be used to prove an upper bound on the error polynomial $h(x)$ which then can be compared to the residual error γ. For Dandelion, we automatically computed the values described by the assumptions to compute an overall bound on $h(x)$. We implement this computations in a function `validateErr`, and explain each of its computation steps on a high-level, based on the assumptions of Theorem 8.

The first assumption (1) from Theorem 8 states that the confidence intervals in Z contain only valid zeros and has been established automatically by the previous step. Based on assumption (2), Dandelion computes B by evaluating $|h'(x)|$ on $\max(|a|, |b|)$. Following assumption (3), Dandelion computes K as the maximum value of evaluating the error polynomial h on the lower bounds of the confidence intervals in Z, and a value e as the maximum value of $|u - v|$ for each $[u, v]$ in Z. Dandelion then computes the overall bound on the error polynomial h as $\max(h(a), h(b), K + B \times e)$. To validate the residual error γ, it then suffices to check $\max(h(a), h(b), K + B \times e) \le \gamma$.

Overall, we prove once and for all soundness of Dandelion as

▶ **Theorem 9** (Dandelion Soundness)**.**

$$\texttt{Dandelion(f,p,I,}\varepsilon\texttt{,n)} = \texttt{true} \ \Rightarrow\ (\forall x.\ x \in \texttt{I} \Rightarrow |\texttt{f}(x) - \texttt{p}(x)| \le \varepsilon)$$

The proof of Theorem 9 uses the triangle inequality to combine the theorem First Phase Soundness (Theorem 4) with the theorem Second Phase Soundness (Theorem 7).

5 Extracting a Verified Binary with CakeML

Computations performed in interactive theorem provers are known to be slower than those in unverified languages. To alleviate this performance problem, the proof-producing synthesis [1] implemented in the CakeML verified compiler [39] translates HOL4 functions into their

CakeML counterpart, with an equivalence proof. These translated CakeML functions are compiled into machine code with the CakeML compiler, and as CakeML is fully verified, the machine code enjoys the same correctness guarantees as its HOL4 version.

During an initial test run we noticed that the Sturm sequence computations in the second phase are the most computationally expensive task of Dandelion. HOL4 represents real-numbers as (reduced) fractions during computation, and we noticed that in Dandelion their size still grew quite large, leading to a single multiplication taking up to 6 hours. Therefore, we use CakeML's proof-producing synthesis to extract a verified binary for the computations described in Section 4. To communicate results of the first phase with the binary, we implemented an (albeit unverified) lexer and parser that reads-in results from the first phase.

6 Evaluation

We have described how Dandelion automatically validates polynomial approximations from Remez-like algorithms. Next, we demonstrate Dandelion's usefulness with three separate experiments, by demonstrating that Dandelion fully automatically

1. validates certificates generated with an off-the-shelf Remez-like algorithm (Subsection 6.1)
2. validates certificates for more complicated elementary function expressions (Subsection 6.2)
3. validates certificates for less-accurate techniques (Subsection 6.3)

All the results we report in this section where gathered on a machine running Ubuntu 20.04, with an 2.7GHz i7 core and 16 GB of RAM. All runnning times are measured using the UNIX `time` command as elapsed wall-clock time in seconds.

6.1 Validating Certificates of a Remez-like Algorithm

In our first evaluation, we show that Dandelion certifies accurate approximation errors from an off-the-shelf Remez-like algorithm. We generate certificates by combining the Daisy static analyzer with the Sollya approximation tool [8], and extend Daisy with a simple pass that replaces calls to elementary functions with an approximation computed by Sollya. As a Remez-like algorithm, we use the `fpminimax` [7] function in Sollya. Our pipeline is benchmarked on numerical kernels taken from the FPBench benchmark suite [12], and the benchmarks used by an unverified extension of Daisy with approximations for elementary functions [24]. These benchmarks represent kernels as they occur in e.g. embedded systems, and thus they benefit from custom polynomial approximations. The original work by Izycheva et al. [24] synthesizes polynomial approximations whose target error bounds are usually larger than those inferred by Sollya. Hence, Dandelion could validate Daisy's bounds as well, but for the sake of the evaluation we choose more challenging, tighter bounds. For each benchmark, Daisy creates a certificate for each approximated elementary function, amounting to a total of 96 generated certificates.

Sollya's implementation of `fpminimax` can be configured to use different degrees for the generated approximation and different formats for the coefficients of the approximation. In our evaluation we approximate elementary functions with a degree 5 polynomial, storing the coefficients with a precision of 53 bits. All input ranges used in the certificates are computed by Daisy without modifying them, except for exp, where we disallow negative intervals, i.e. if Daisy wanted to approximate on $[-x, y]$, we change it to $[0, y]$ as Dandelion currently does not support negative exponentials. This can be fixed by straight-forward range reductions that are independent to the approximations computed by Remez-like algorithms. For each such approximation, Daisy creates a certificate to be checked by Dandelion.

Table 1 Overview of certificates validated with Dandelion, MetiTarski, and CoqInterval.

Function	#	Dandelion			MetiTarski		CoqInterval	
		Verified	HOL4(s)	Binary(s)	Verified	Time(s)	Verified	Time(s)
atan	2	1	11.62	20.36	2	6.80	2	1.74
cos	28	25	202.35	251.33	25	3.15	26	1.91
exp	21	18	39.58	212.54	10	5.35	20	1.58
log	8	0	0	0	5	4.99	8	1.68
sin	31	27	17.83	295.66	25	6.32	31	1.78
Total	90	71			67		87	

The MetiTarski automated theorem prover [3] is a tool that provides the same level of automation as Dandelion, but relies on a different technology for proving inequalities. MetiTarski is based on the Metis theorem prover [22] and can output proofs in TSTP format [38]. It relies on an external decision procedure to discharge some goals, and for our experiments we used Mathematica. In general, MetiTarski checks real-number inequalities that may contain elementary functions, thus we compare the number of certificates validated by Dandelion to those that can be checked by MetiTarski.

Further, the CoqInterval package [28] proves inequalities about elementary functions in the Coq theorem prover [10]. In contrast to Dandelion's technique based on high-accuracy Taylor polynomials and Sturm sequences, CoqInterval is based on interval arithmetic with optional interval bisections and high-accuracy Taylor polynomials. Further, Dandelion is verified in HOL4, while CoqInterval is implemented in Coq. To compare the two approaches, our evaluation also includes the CoqInterval package. Our evaluation excludes a similar approach formalized in Isabelle/HOL [21] because we could not come up with a straightforward translation of our certificates as inputs to the tool. We have manually tested some of our benchmarks and expect it to produce results similar to CoqInterval.

Our results are given in Table 1. The left-most column of Table 1, contains the name of the elementary function approximated by Daisy, and the second column, labeled with a # contains the number of certificates generated for the elementary function, with unique input ranges. The next three columns, headed "Dandelion", contain the number of certificates validated by Dandelion, the average HOL4 running time for the first phase in seconds, and the average running time of the binary for the second phase in seconds. The next two columns, headed "MetiTarski", contain the number of certificates validated by MetiTarski, and the average running time in seconds. The final two columns, headed "CoqInterval", contain the number of certificates validated by CoqInterval, and the average running time in seconds.

Our evaluation truncates Taylor series after 32 terms in `approxAsPoly`. In general, we found six times the degree of the computed approximation to be a good estimate for when to truncate Taylor series in Dandelion. Our use of 32 instead of 30 is a technical detail, as some Taylor series require both the number of series terms n, as well as $\frac{n}{2}$ to be even. In general, the number of series terms has to be significantly higher than the degree of the approximated polynomial, to make the approximation error of the first phase almost negligible.

Overall, we notice that each of the tools in our evaluation certifies a slightly different set of approximations. In total, CoqInterval certifies most of the benchmarks, but Dandelion successfully checks one cosine certificate that CoqInterval fails to check. While Dandelion validates more certificates than MetiTarski, both MetiTarski and CoqInterval validate certificates that are currently out of reach for Dandelion. This is mostly due to the first

phase of Dandelion. Even though we used general, widely known truncated Taylor series for all supported elementary functions, Dandelion fails to validate certificates for the ln function. We have inspected the generated certificates, and Dandelion cannot compute a high-accuracy polynomial approximation for 5 of them because they do not satisfy the precondition of Dandelion's Taylor series. For the remaining 3 certificates, we ran into issues with Sollya's computation of the confidence intervals for the zeros. On a high-level, the problem originates from the derivative of the error polynomial $h(x)$ being very close to 0, leading to a huge number of zeros being found, i.e. computation not terminating within a reasonable amount of time. To demonstrate that Dandelion still certifies errors for the ln function, we add an example in Subsection 6.2.

While Dandelion cannot certify errors for the ln function in this part of the evalution, this is not a conceptual limitation, as polynomial approximations are commonly paired with an argument reduction strategy. While verification of these strategies is orthogonal to validating results of an Remez-like algorithm, they could be used to reduce the input range of the approximated elementary function into a range that Dandelion can certify. More generally, we have done the heavy lifting of automating the computations and implementing the general framework, such that adding more accurate Taylor series to Dandelion amounts to mere proof engineering, modulo coming up with Taylor series in the first place. For the certificates for $\tan^{-1}$, cos, sin, and exp, we notice that the average running time is in the order of minutes, making certificate checking with Dandelion's verified binary feasible.

We compare the approximation errors recorded in the certificates with the infinity norm computed by Sollya, which is the most-accurate estimate of the approximation error [9]. Overall, Dandelion certifies an approximation error in the same order of magnitude as the infinity norm for 61 certificates. For the remaining 10, the error is a sound upper bound. In general, infinity norm-based estimates are known to be the most accurate and their verification requires more elaborate techniques than Sturm sequences [9]. Consequently we would not expect Dandelion to be able to always certify infinity norms.

We ran the evaluation for an approximation degree of 3, with precisions of 53 and 23 each to measure the influence of those parameters. Overall, the running time significantly decreases when decreasing from degree 5 to 3, going from average running times of minutes to average running times of seconds. Decreasing the precision of the coefficients further speeds up evaluation, though not as significant as decreasing the degree did. This suggests that higher coefficient accuracies can easily be used for generating polynomials with Remez-like algorithms, and lower degree polynomials should be preferred for fast validation.

6.2 Validating Certificates for Elementary Function Expressions

Next, we show that Dandelion can also certify approximation errors for complicated elementary function expressions. We validate with Dandelion approximation errors for random examples involving elementary functions and arithmetic. Polynomial approximations are again generated by Sollya.

An overview of our results is given in Table 2. The table shows the approximatied function, then Sollya's parameters (the input range, the target degree (Deg.), the target precision (Prec.)), and then the infinity-norm (∞-norm) of the approximation. The final columns summarize the Dandelion results, giving the certified approximation error, and the running time in seconds of the first phase (HOL4) and the second phase (Binary).

Overall, we notice that the certified approximation error is on the same order of magnitude as the infinity norm for all examples. We also notice that performance for both phases varies across the different examples. For the first phase this is often due to how the input ranges are encoded in the certificate. We noticed that HOL4 is very sensitive to how the fractions representing real numbers are encoded when performing computations. Similarly,

Table 2 Functions approximated with Sollya using `fpminimax` and certified with Dandelion.

Function	Range	Deg.	Prec.	∞-norm	Error	HOL4	Binary
$\cos(x+1)$	$[0, 2.14]$	5	53	3.06E-5	3.06E-5	169	63
$\sin(x-2)$	$[-1, 3.00]$	5	53	2.05E-3	2.91E-3	93	68
$\ln(x+\frac{1}{10})$	$[1.001, 1.1]$	3	32	1.08E-7	1.08E-7	2775	1773
$\exp(x\times\frac{1}{2})+\cos(x\times\frac{1}{2})$	$[0.1, 1.00]$	5	53	2.03E-9	4.45E-9	711	5
$\arctan(x)-\cos(\frac{3}{4}\times x)$	$[-0.5, 0.5]$	5	53	1.18E-5	1.18E-5	24	2308

performance of the second phase greatly varies depending on the complexity of the error polynomial computed by the first phase. We observe that perfomance improves with both smaller degree polynomials, and smaller representations of the polynomial coefficients.

The results in Table 2 exclude examples where two elementary functions are composed with each other, e.g. as in $\exp(\sin(x-1))$. This is because Dandelion computes the global high-accuracy approximation in the first phase via polynomial composition of an approximation for exp and sin. While this is theoretically supported by Dandelion, we found that the polynomial composition leads to an exponential blow-up in the degree of the error polynomial. Even for the innocuously looking example $\exp(\sin(x-1))$, the second phase could not validate a polynomial approximation within 24 hours. This clearly motivates the use of more elaborate Taylor series if compound elementary functions need to be certified by Dandelion. In general, settings where elementary functions like those in Table 2 are used could potentially be made more accurate and be validated faster with custom Taylor series.

6.3 Validating Certificates for Simpler Approximation Algorithms

Remez-like algorithms are known to be the most accurate approximation algorithms. However, less accurate approaches are still in use today, and as such interesting targets for verification. Bréhard et al. [6] certify Chebyshev approximations in the Coq theorem prover, where their approach requires some manual proofs. We demonstrate that Dandelion also certifies Chebyshev approximations on some random examples by computing Chebyshev approximations with Sollya's function `chebyshevform`. The results are shown in Table 3.

Again, we first give Sollya's parameter and the infinity norm, then we give the error certified by Dandelion, and the execution times for the first and second phase.

We also include the approximation certified by Harrison [18], labeled with a *. The polynomial has degree 3, but we leave the precision empty as it is not generated by Sollya, and we do not provide an infinity norm. The only difference to the proof from Harrison is that we prove the bound only for positive x, as Dandelion currently does not handle exponentials on negative values. The lower bound of the range is 0.003, instead of 0 to rule out a 0 on the lower bound (as $\exp(x)-1=0$ for $x=0$), which we must exclude by Theorem 8. Harrison's manual proof of the polynomial approximation then reduces to a single line running Dandelion on the encoding.

7 Related Work

Throughout the paper, we have already hinted at the immediate related work. Next, we explain the key conceptual differences between Dandelion and the immediate related work and put Dandelion into the greater context. In general, Dandelion touches upon two key research areas in interactive and automated theorem proving: techniques for approximating elementary functions and techniques for proving theorems involving real-numbered functions.

Table 3 Chebyshev approximations certified with Dandelion.

Function	Range	Deg.	Prec.	∞-norm	Error	HOL4	Binary
$\cos(x)$	$[0, 2.14]$	5	53	3.17E-7	3.22E-7	169	63
$\sin(x+2)$	$[-1.5, 1.5]$	5	53	4.47E-4	7.60E-4	142	138
$\sin(3 \times x) + \exp(x \times \frac{1}{2})$	$[0, 1]$	3	53	2.45E-2	2.48E-2	54	1897
$\exp(x) - 1^*$	$[0.003, 0.01]$	3			$2^{-33.2}$	133	<1

Approximating Elementary Functions. The work on approximating elementary functions can be distinguished among two axes: whether or not the work provides rigorous machine-checked proofs, and whether the work is fully automated or requires user intervention.

Fully automated, rigorous machine-checked proofs, similar to Dandelion are provided by the work by Bréhard et al. [6]. They develop a framework for proving correct Chebychev approximations of real number functions in the Coq theorem prover [10]. Also in Coq, Martin-Dorel and Melquiond [29] verify polynomial approximations using the CoqInterval [28] package. They develop a fully automated tactic for proving approximations inside floating-point mathematical libraries correct. A key difference between Dandelion and both these tools is that they cannot certify approximations computed by Remez-like algorithms, which can in general provide more accurate approximations.

For manual proofs, versions of the exponential function have been verified by Harrison [17], and Akbarpour et al. [2]. The manual proof by Harrison [18] layed out the foundations for Dandelion. The work has also been extended by Chevillard et al. [9]. Instead of verifying approximation errors for polynomials, they use so-called sum-of-squares decompositions [19] to certify infinity norm computations. A major limiting factor for their work was finding accurate enough Taylor polynomials which we found to not be a major issue for our approach.

Coward et al. [11] use the MetiTarski automated theorem prover [3] to verify accuracy of hardware finite-precision implementations of elementary functions. MetiTarski provides proofs in machine-readable form using the TSTP format [38] instead of being developed inside an interactive theorem prover like HOL4. A major conceptual difference is that the verification done by Coward et al. reasons about bit-level accuracy of the hardware implementation, while Dandelion reasons about real-number functions and polynomials. Together with a verified roundoff error analysis like FloVer [4], Dandelion could be extended to verify finite-precision implementations of elementary functions, and together with Daisy [24] verification could possibly be lifted to entire arithmetic kernels.

A different style of unverified approximations is provided by Lim et al. [27]. Instead of computing specialized polynomial approximations, they focus on correctly-rounded, general purpose approximations. These approximations are not build for specific use-cases, but should rather be seen as replacements for the functions provided in mathematical libraries. At the time of writing, their approach is not formally verified, but they do provide a pen-and-paper correctness argument for their code generation. The CR-libm [13] library also provides unverified alternatives of correctly rounded mathematical libraries, and Muller [32] gives a general overview of the techniques for implementing elementary functions.

(Automated) Real-Number Theorem Proving. Dandelion heavily relies on HOL4's support for real-number theorem proving. Below we list some alternatives for proving properties of real-numbers in both interactive and automated theorem proving systems. In the HOL-family of ITP systems, Harrison [19] has formalized sum-of-squares certificates for the HOL-Light [20] theorem prover. His approach relies on semidefinite programming to find a decomposition of a polynomial into a sum-of-squares polynomial. Both Isabelle/HOL and PVS have been

independently extended with implementations of Sturm sequences [15, 34, 35]. Their main focus is not on verification of polynomial approximations, they rather use Sturm sequences to prove properties about roots of polynomials, non-negativity, and monotonicity.

Previously we have already mentioned the MetiTarski automated theorem prover [3], as an example of an automated theorem prover for real-numbered functions. However, MetiTarski is not the only automated prover for real-numbered functions. Real-numbered functions are also supported by e.g. dReal [16], and z3's SMT theory for real-numbers [31].

8 Conclusion

We have presented Dandelion, a verified and fully automated certificate checker for polynomial approximations of elementary functions computed with Remez-like algorithms. Dandelion splits the validation task into two clearly separated phases: The first phase replaces elementary functions by high-accuray Taylor series, and the second phase uses Sturm's theorem and an external oracle to validate the approximation error. Our evaluation has shown that Dandelion certifies approximation errors computed by an off-the-shelf Remez-like algorithm, and Dandelion also certifies approximation errors for Chebyshev approximations.

References

1 Oskar Abrahamsson, Son Ho, Hrutvik Kanabar, Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Yong Kiam Tan. Proof-Producing Synthesis of CakeML from Monadic HOL Functions. *Journal of Automated Reasoning (JAR)*, 64(7), 2020. doi:10.1007/s10817-020-09559-8.

2 Behzad Akbarpour, Amr Abdel-Hamid, Sofiène Tahar, and John Harrison. Verifying a Synthesized Implementation of IEEE-754 Floating-Point Exponential Function using HOL. *The Computer Journal*, 53:465–488, May 2010. doi:10.1093/comjnl/bxp023.

3 Behzad Akbarpour and Lawrence Charles Paulson. MetiTarski: An Automatic Theorem Prover for Real-Valued Special Functions. *Journal of Automated Reasoning (JAR)*, 44(3):175–205, 2010. doi:10.1007/s10817-009-9149-2.

4 Heiko Becker, Nikita Zyuzin, Raphaël Monat, Eva Darulova, Magnus O Myreen, and Anthony Fox. A Verified Certificate Checker for Finite-Precision Error Bounds in Coq and HOL4. In *Formal Methods in Computer Aided Design (FMCAD)*, 2018. doi:10.23919/FMCAD.2018.8603019.

5 Hans-J. Boehm. Towards an API for the Real Numbers. In *Programming Language Design and Implementation (PLDI)*, 2020. doi:10.1145/3385412.3386037.

6 Florent Bréhard, Assia Mahboubi, and Damien Pous. A Certificate-Based Approach to Formally Verified Approximations. In *Interactive Theorem Proving (ITP)*, 2019. doi:10.4230/LIPIcs.ITP.2019.8.

7 Nicolas Brisebarre and Sylvain Chevillard. Efficient polynomial L-approximations. In *IEEE Symposium on Computer Arithmetic (ARITH)*, 2007. doi:10.1109/ARITH.2007.17.

8 S. Chevillard, M. Joldeş, and C. Lauter. Sollya: An Environment for the Development of Numerical Codes. In *International Congress on Mathematical Software (ICMS)*, 2010. doi:10.1007/978-3-642-15582-6_5.

9 Sylvain Chevillard, John Harrison, Mioara Joldeş, and Ch Lauter. Efficient and accurate computation of upper bounds of approximation errors. *Theoretical Computer Science*, 412(16):1523–1543, 2011. doi:10.1016/j.tcs.2010.11.052.

10 The Coq Proof Assistant. URL: https://coq.inria.fr.

11 Samuel Coward, Lawrence Paulson, Theo Drane, and Emiliano Morini. Formal Verification of Transcendental Fixed and Floating Point Algorithms using an Automatic Theorem Prover. *Formal Aspects of Computing (in press)*, 2022.

12 Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Chen Qiu, Alexander Sanchez-Stern, and Zachary Tatlock. Toward a Standard Benchmark Format and Suite for Floating-Point Analysis. In *Numerical Software Verification (NSV)*, 2016. doi:10.1007/978-3-319-54292-8_6.

13 Catherine Daramy, David Defour, Florent de Dinechin, and Jean-Michel Muller. CR-LIBM: a correctly rounded elementary function library. In *Advanced Signal Processing Algorithms, Architectures, and Implementations*, volume 5205, pages 458–464. International Society for Optics and Photonics, 2003.

14 Eva Darulova and Anastasia Volkova. Sound Approximation of Programs with Elementary Functions. In *Computer Aided Verification (CAV)*, 2019. doi:10.1007/978-3-030-25543-5_11.

15 Manuel Eberl. A Decision Procedure for Univariate Real Polynomials in Isabelle/HOL. In *Certified Programs and Proofs (CPP)*, 2015. doi:10.1145/2676724.2693166.

16 Sicun Gao, Soonho Kong, and Edmund M Clarke. dReal: An SMT solver for nonlinear theories over the reals. In *Conference on Automated Deduction (CADE)*, 2013. doi:10.1007/978-3-642-38574-2_14.

17 John Harrison. "floating point verification in hol light: The exponential function". In *Algebraic Methodology and Software Technology (AMAST)*, 1997. doi:10.1007/BFb0000475.

18 John Harrison. Verifying the accuracy of polynomial approximations in HOL. In *Theorem Proving in Higher Order Logics (TPHOLs)*, 1997. doi:10.1007/BFb0028391.

19 John Harrison. Verifying nonlinear real formulas via sums of squares. In *Theorem Proving in Higher Order Logics (TPHOLs)*, 2007. doi:10.1007/978-3-540-74591-4_9.

20 The HOL-Light Proof Assistant. URL: https://www.cl.cam.ac.uk/~jrh13/hol-light/.

21 Johannes Hölzl. Proving inequalities over reals with computation in isabelle/hol. In *Programming Languages for Mechanized Mathematics Systems*, 2009.

22 Joe Hurd. First-Order Proof Tactics in Higher-Order Logic Theorem Provers. In *Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports*, 2003.

23 The Isabelle/HOL Proof Assistant. URL: https://isabelle.in.tum.de/.

24 Anastasiia Izycheva, Eva Darulova, and Helmut Seidl. Synthesizing Efficient Low-Precision Kernels. In *Automated Technology for Verification and Analysis (ATVA)*, 2019. doi:10.1007/978-3-030-31784-3_17.

25 Olga Kupriianova and Christoph Lauter. Metalibm: A Mathematical Functions Code Generator. In *International Congress on Mathematical Software (ICMS)*, 2014. doi:10.1007/978-3-662-44199-2_106.

26 Wenda Li, Grant Olney Passmore, and Lawrence C Paulson. Deciding univariate polynomial problems using untrusted certificates in isabelle/hol. *Journal of Automated Reasoning*, 62(1):69–91, 2019. doi:10.1007/s10817-017-9424-6.

27 Jay P Lim and Santosh Nagarakatte. One Polynomial Approximation to Produce Correctly Rounded Results of an Elementary Function for Multiple Representations and Rounding Modes. *Principles of Programming Languages (POPL)*, 2022. doi:10.1145/3498664.

28 Érik Martin-Dorel and Guillaume Melquiond. CoqInterval: A Toolbox for Proving Nonlinear Univariate Inequalities in Coq. In *Conference on Effective Analysis: Foundations, Implementations, Certification*, 2016.

29 Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions with elementary functions in Coq. *Journal of Automated Reasoning (JAR)*, 57(3):187–217, 2016. doi:10.1007/s10817-015-9350-4.

30 R.E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

31 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008. doi:10.1007/978-3-540-78800-3_24.

32 Jean-Michel Muller. *Elementary Functions*. Springer, 2006.

33 César Muñoz, Anthony Narkawicz, George Hagen, Jason Upchurch, Aaron Dutle, María Consiglio, and James Chamberlain. DAIDALUS: detect and avoid alerting logic for unmanned systems. In *Digital Avionics Systems Conference (DASC)*, 2015.

34 Anthony Narkawicz, César Munoz, and Aaron Dutle. Formally-Verified Decision Procedures for Univariate Polynomial Computation Based on Sturm's and Tarski's theorems. *Journal of Automated Reasoning (JAR)*, 54(4):285–326, 2015. doi:10.1007/s10817-015-9320-x.

35 Anthony Narkawicz, Cesar Munoz, and Aaron Dutle. A decision procedure for univariate polynomial systems based on root counting and interval subdivision. *Journal of Formalized Reasoning*, 11(1):19, 2018. doi:10.6092/issn.1972-5787/8212.

36 Ricardo Pachón and Lloyd N Trefethen. Barycentric-Remez Algorithms for Best Polynomial Approximation in the Chebfun System. *BIT Numerical Mathematics*, 49(4):721, 2009.

37 Konrad Slind and Michael Norrish. A Brief Overview of HOL4. In *Theorem Proving in Higher Order Logics (TPHOLs)*, 2008. doi:10.1007/978-3-540-71067-7_6.

38 Geoff Sutcliffe, Jürgen Zimmer, and Stephan Schulz. Tstp data-exchange formats for automated theorem proving tools. *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, 112:201–215, 2004.

39 Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. The Verified CakeML Compiler Backend. *Journal of Functional Programming (JFP)*, 29, 2019. doi:10.1017/S0956796818000229.

40 Ping-Tak Peter Tang. Table-driven implementation of the exponential function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software (TOMS)*, 15(2):144–157, 1989. doi:10.1145/63522.214389.

The Zoo of Lambda-Calculus Reduction Strategies, And Coq

Małgorzata Biernacka
University of Wrocław, Faculty of Mathematics and Computer Science, Poland

Witold Charatonik
University of Wrocław, Faculty of Mathematics and Computer Science, Poland

Tomasz Drab
University of Wrocław, Faculty of Mathematics and Computer Science, Poland

Abstract

We present a generic framework for the specification and reasoning about reduction strategies in the lambda calculus, representable as sets of term decompositions. It is provided as a Coq formalization that features a novel format of *phased* strategies. It facilitates concise description and algebraic reasoning about properties of reduction strategies. The formalization accommodates many well-known strategies, both weak and strong, such as call by name, call by value, head reduction, normal order, full β-reduction, etc. We illustrate the use of the framework as a tool to inspect and categorize the "zoo" of existing strategies, as well as to discover and study new ones with particular properties.

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

Keywords and phrases Lambda calculus, Reduction strategies, Coq

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.7

Supplementary Material *Software (Source Code)*: `https://bitbucket.org/pl-uwr/strategies` archived at `swh:1:dir:a1628e852357d77fe8d678b7adbbb536baa331ab`

Funding This research is supported by the National Science Centre of Poland, under grant number 2019/33/B/ST6/00289.

Acknowledgements We thank the anonymous reviewers for their comments and references to the literature. The last author thanks his wife, Monika, for taking care of everything around during proof completion.

1 Introduction

The behavior of the lambda calculus, be it as a computation model or a prototype programming language, is based on the notion of β-reduction that constitutes a basic computation step. In its general definition β-reduction is non-deterministic and unrestricted – a term can be reduced in different ways, and the consequences of the different reduction choices can vary tremendously, as for the term $K\,x\,\Omega$: it can normalize in two steps in call by name, reduce indefinitely in call by value [27], or it can be stuck in a variant of call by value with abstractions as the only values. In many applications it is useful or even necessary to trim down the full generality of β-reduction by following a specific (not necessarily deterministic) *reduction strategy*. In general terms, as Barendregt put it, "a reduction strategy provides a way of choosing how to reduce a term" [5].

When considering lambda calculus as a programming language or a computation model, Barendregt's specification is too general, and we typically impose quite strong restrictions for strategies to be considered useful or efficient. Computation with the lambda calculus consists in reducing terms until a normal form (i.e., an irreducible term) is found. Therefore, one can adopt the following properties as practically relevant characteristics of reduction

strategies: termination (does the strategy always terminate for a term that has a normal form?), the number of steps it performs to reach a normal form, or its effectiveness (i.e., how easy it is to determine at each step what the reduced term should be).

The concept of effectiveness of the strategy poses the following problem to consider: how do we define reduction strategies, and in particular how do we define effective strategies? Fortunately, this question has many useful answers coming from the programming languages community, where various formats of operational semantics have been developed. The popular approaches include: big-step semantics that describes the relation between the term and its final value [18], and small-step semantics that describes single steps of computation, and is typically given in the format of structural operational semantics [28], or of reduction semantics with explicit representation of reduction contexts [12]. Effectiveness of these formats (and in consequence, of the strategies defined by them) relies on the fact that their semantic rules are typically defined inductively, and in such a way that the cost of applying a rule can be bounded by some low factor (ideally, constant) related to the inspection of the data structures involved. There exist numerous reduction strategies introduced to model desired properties of the lambda calculus that can be described in a common semantic format, there exist methodologies and tools to implement and test them [10, 11, 34], but how can we compare strategies, and how do we find new, useful ones?

In this work, we make an attempt at characterizing and categorizing effective strategies in the pure lambda calculus, as they pertain to the order of β-reductions in a reduction sequence, and we use Coq as our implementation platform [33]. A source of inspiration for our work was Sestoft's survey of reduction strategies in the lambda calculus [31] that collected existing strategies and their ML implementations, as a didactic tool. However, unlike Sestoft, we base our work on the small-step approach. We exploit the fact that in general small-step semantics provides more fine-grained control over computation, for example it makes explicit the order of evaluation (as in left-to-right vs. right-to-left), whereas big-step semantics may leave it unspecified. Therefore, we choose to take as a foundation the view of strategies as *decompositions* of terms into reduction contexts and redices, as offered by the format of reduction semantics. The framework that we built on these assumptions delineates the space where most of existing strategies live, and new ones can be discovered, systematically. One notable omission is the call-by-need strategy – it seems that this strategy should fit in our picture, but it is harder to characterize, and therefore still remains in the wild.

Related Work. The work closest to ours is Sestoft's survey of lambda-calculus reduction strategies. It is accompanied by implementation, but it does not mechanize the properties of the strategies. Recently, with the rising popularity of proof assistants numerous mechanizations of concrete programming languages have been done, but we are not aware of any comprehensive study of various reduction strategies.

Earlier formalizations of the λ-calculus theory include Shankar's mechanization of the Church-Rosser theorem in Boyer-Moore theorem prover [32] and Pfenning's mechanization in Elf [25], Huet's formalization of the residual theory of λ-calculus in Coq [17], McKinna & Pollack's formalization of Church-Rosser theorem, standardization theorem, and the basic theory of Pure Type Systems in LEGO Proof Development System [22], Norrish's formalization of basic λ-calculus theory in HOL [23]. Pierce et al. provide a series of textbooks on software verification including usage of operational semantics in the form of Coq proof scripts [26]. Biernacka et al. formalize derivations of refocusing abstract machines for various lambda-calculus reduction strategies in Coq [8]. Most recently, Forster et al. formalized in Coq reasonability for time and space of weak call by value (shortened *lcbv* and *cbv* here) [13, 14] and essential theorems for it as a model of computation [15].

Contributions and outline. The contributions of our work are the following:

1. We provide a generic, simple and intuitive formalization of lambda-calculus reduction strategies in Coq. The formalization is discussed in Section 2.
2. We collect and categorize common existing strategies studied in the functional programming-language community. We formalize and prove some of their properties, such as inclusion, determinism, and uniformity. This systematization is presented in Section 3.
3. We propose a novel semantic format to define *phased* reduction strategies, formalized in Coq, that facilitates concise specification and algebraic reasoning about strategies. It is introduced in Section 4, where some of its benefits and advantages over existing formats are also discussed.

2 Basic concepts and Coq formalization

We aimed at a simple, concise and intuitive formalization. It turns out we only need two `Inductive` definitions in the whole development: one for λ-terms, and one for context frames. Our formalization is also minimalistic in the sense that we do not introduce any more concepts than we need. In particular, we do not need substitution to talk about strategies, and so we do not include it in our formalization. This is not a limitation, since the framework is open to extensions and substitution can be added in order to prove dynamic properties of the lambda calculus under any definable strategy. Many notions are expressed in terms of basic set theory as presented next.

2.1 Sets

Sets of elements of type `A`, denoted with $\mathcal{P}$ `A`, are represented as functions of type `A` $\to$ `Prop`. This is similar to `Coq.Sets.Ensembles` but we do not use the axiom of extensionality. The definitions are standard, and excerpts are shown in Listing 1.[1]

Listing 1 Basic relations on sets.

```
Notation "'𝒫' A"    := (A → Prop) (at level 55, only parsing).
Notation "x '∈' A" := (A x)       (at level 70, only parsing).

Definition   subset {A:Type} (s t:𝒫 A) : Prop := ∀ x, x ∈ s → x ∈ t.
Definition   set_eq {A:Type} (s t:𝒫 A) : Prop := ∀ x, x ∈ s ↔ x ∈ t.
Definition disjoint {A:Type} (s t:𝒫 A) := ∀ x, x ∈ s → x ∈ t → False.

Infix    "⊆" := subset             (at level 70).
Infix   "==" := set_eq             (at level 70).
Notation "∅" := empty_set.
Notation "●" := full_set.
Infix    "∪" := union              (at level 65).
Notation "⋃" := family_union.
Infix    "×" := cartesian_product (at level 50).
```

Using the typeclass mechanism of Coq, we show that `set_eq` is an equivalence relation, that `union` is a properly defined operation on sets, and that cartesian product is monotone (see Listing 2).

[1] It is worth noting that `set_eq` is independent of notion of equality on type `A` if it has its own one as for example real numbers.

Listing 2 Example properties of sets.

```
Instance set_eq_equiv {A:Type} : Equivalence (@set_eq A).
Instance union_proper {A:Type} :
  Proper (set_eq ==> set_eq ==> set_eq) (@union A).
Instance cartesian_product_monotone {A B:Type} :
  Proper (subset ++> subset ++> subset) (@cartesian_product A B).
```

2.2 Families of terms

The grammar of lambda terms

$$t ::= x \mid \lambda x.t \mid t_1\, t_2$$

is formalized in a straightforward way, with strings used to represent variable names, as shown in Listing 3.

Listing 3 Inductive definition of terms.

```
Inductive term : Type :=
| var (x : string) : term
| lam (x : string) (s : term) : term
| app (s : term)   (t : term) : term.
```

When discussing reduction strategies, we will stumble upon various normal forms – they occur as term families expressible with simple grammars.

$$\begin{array}{rlrl} nf \ni n & ::= \lambda x.n \mid a & neu \ni a & ::= x \mid a\, n \\ wnf \ni w & ::= \lambda x.t \mid i & inert \ni i & ::= x \mid i\, w \\ hnf \ni h & ::= \lambda x.h \mid r & rigid \ni r & ::= x \mid r\, t \\ whnf \ni q & ::= \lambda x.t \mid r & & \\ abs \ni \_ & ::= \lambda x.t & abs^{\mathtt{C}} \ni \_ & ::= x \mid t\, t \end{array}$$

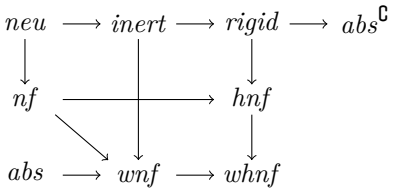


Figure 1 Term families and their inclusions.

In Figure 1, the grammars of common normal forms are shown, together with a diagram illustrating the relationships between them (the arrows represent inclusions between families). For example, *nf* is the family of full β-normal forms (resulting from full, unrestricted normalization), and is defined with the use of an auxiliary family of neutral terms *neu*. The family *wnf* of weak normal forms occurs as the final results of computation in the (weak) open call-by-value strategy, and is defined with the use of an auxiliary family of *inert* terms [2]. Weak-head normal forms *whnf* result from call-by-name reduction, and its subfamily of head normal forms *hnf* contains head-reduced terms. The name *rigid* for variable-headed application comes from Accattoli et al. [1].

In the implementation we define the various normal forms as sets of terms using `Fixpoints`, as shown for *nf* and *neu* in Listing 4. Such definitions contain superfluous information, e.g., the grammar of *neu* has to be repeated. Nevertheless, we show in lemmas `nf_grammar` and `neu_grammar` in Listing 4 that the fixpoint definitions coincide with the grammars in Figure 1, where `abs_of` and `app_of` are Coq versions of the constructors used in the grammars. Moreover, using simple set-based reasoning we can prove that neutral terms are exactly rigid normal terms (`neu_rigid_nf`), and we can prove all inclusions shown in Figure 1 (the example of $neu \subseteq nf$ is shown in the listing).

Listing 4 Definition, grammars and properties of normal and neutral terms.

```
Fixpoint neu (a:term) : Prop :=
  match a with
  | var x   => True
  | app a n => neu a ∧ nf n
  | _       => False
  end
with nf (n:term) : Prop :=
  match n with
  | lam x n => nf n
  (* _       => neu n *)
  | var x   => True
  | app a n => neu a ∧ nf n
  end.

Lemma   nf_grammar :  nf == abs_of nf ∪ neu.
Lemma  neu_grammar : neu ==  variable ∪ app_of neu nf.
Lemma neu_rigid_nf : neu == rigid ∩ nf.
Lemma    neu_is_nf : neu ⊆ nf.
```

2.3 Semantic formats

In this section we discuss the most popular approaches to defining computation in the lambda calculus in order to justify the choices made in our formalization. Our running example is the left-to-right open call-by-value strategy (shortened as *lcbw*) studied in detail in [2] but – as is often the case – occurring in the literature previously [24, 31].

Definitional interpreters

A natural way for a programmer to define a reduction is to write a definitional interpreter in some metalanguage [30]. This way *lcbw* is defined in Paulson's textbook [24] and we present his evaluator in Listing 5, translated to Coq syntax. The definition is clear and the deterministic order of reductions follows from the evaluation order of the metalanguage. However, Coq cannot accept this definition because of possible nontermination.

Listing 5 Pseudocode of Paulson's evaluator for left-to-right open call by value.

```
Parameter subst : string → term → term → term.

Fixpoint eval (t : term) :=
  match t with
  | app s t => match eval s with
               | lam x u => eval (subst x (eval t) u)
               |       u =>          app u (eval t)
               end
  |       t => t
  end.
```

A variant of this approach is to define a higher-order evaluator that represents some values as functions of the metalanguage, but this is also inapplicable in our situation because of nontermination.

Big-step operational semantics

The evaluator above can be intuitively translated to big-step operational semantics format. The big-step formulation of the *lcbw* strategy is given in Figure 2. It is isomorphic to the one given by Sestoft [31] under the name of "the call-by-value reduction" *bv*.

$$\frac{}{x \Downarrow x} \qquad \frac{}{\lambda x.t \Downarrow \lambda x.t} \qquad \frac{t_1 \Downarrow \lambda x.t \qquad t_2 \Downarrow t_2' \qquad t[x := t_2'] \Downarrow t'}{t_1\, t_2 \Downarrow t'} \qquad \frac{t_1 \Downarrow t_1' \not\equiv \lambda x.t \qquad t_2 \Downarrow t_2'}{t_1\, t_2 \Downarrow t_1'\, t_2'}$$

Figure 2 Sestoft's big-step operational semantics of open call by value.

The inductive definition of the relation $\Downarrow$ can be formulated in Coq. However, due to presentation of inference rules with multiple independent premises, the evaluation order happens to be only conventional. In order to discriminate between, e.g., left-to-right and right-to-left, we need more specific formats.

Another drawback shared by the big-step approaches is the fact that actual computation is welded with navigation in a term. Therefore, we consider these semantic formats as derived from a more fundamental, small-step semantics [29]. Interestingly, the specification of evaluation order can be restored in pretty-big-step semantics [9], which – being big-step in spirit – is convenient for certain types of reasoning.

Small-step structural operational semantics

In Figure 3, we show the specification of *lcbw* by a relation $\overset{lcbw}{\to}$ defined in the format of Plotkin's structural operational semantics [28]. This definition is directly expressible in Coq.

$$\frac{}{(\lambda x.t)\, w \overset{lcbw}{\to} t[x := w]}\ (\beta) \qquad \frac{t_1 \overset{lcbw}{\to} t_1'}{t_1\, t_2 \overset{lcbw}{\to} t_1'\, t_2}\ (\mu) \qquad \frac{t_2 \overset{lcbw}{\to} t_2'}{w_1\, t_2 \overset{lcbw}{\to} w_1\, t_2'}\ (\nu)$$

Figure 3 Structural operational semantics for left-to-right open call by value.

Notice that the small-step formulation demands more information about the reduction strategy. In the rule (β), we have to restrict substitution to accept only weak normal forms w, as defined in Figure 1. Without this condition the defined strategy would be nondeterministic and not call-by-value. Similarly, the rule (ν) states explicitly that the left component of the application has to be a weak normal form. Therefore, in contrast to the big-step approach, the small-step formulation requires a precise description of the grammar of terms allowed in designated positions in a term, for a rule to be applied.

Small-step reduction semantics

We can observe that that rules (μ) and (ν) in structural operational semantics play a different role than (β) because they simply propagate inner steps while (β) is responsible for the actual reduction. In reduction semantics [12] these propagation rules are materialized with an explicit data structure representing contexts, and the one-step reduction is expressed via contextual closure of the basic reduction rule called *contraction*. In Figure 4, we present a reduction semantics for *lcbw*, which coincides with the left-to-right variant of Accattoli and Guerrieri's definition of open call by value [2].

$$\frac{}{(\lambda x.t)\, w \rightharpoonup_{\beta_w} t[x := w]} \qquad \frac{t \rightharpoonup_{\beta_w} t'}{E[t] \overset{lcbw}{\rightarrow} E[t']} \qquad E ::= \Box \mid w\, E \mid E\, t$$

Figure 4 Reduction semantics for left-to-right open call by value.

The format of reduction semantics is more compact than the one of structural operational semantics. It is also more convenient in defining reduction strategies as it separates contraction (specifying how subterms are rewritten) from contexts (specifying where the contraction takes place). Therefore we choose this format to define strategies and to formalize in Coq. However, in the following, we will use the nonterminal ${}^{w}V^{t}$ instead of E for contexts of *lcbw* due to the shortage of capital letters in the latin alphabet.

2.4 Reduction contexts

Traditionally, contexts used in reduction semantics are defined by grammars. The grammar of general lambda-calculus contexts is the following[2]:

$$C ::= \Box \mid \lambda x.C \mid C\, t \mid t\, C$$

Every context can be seen as a term with exactly one free occurrence of a special variable $\Box$ called the hole. It is used to indicate a particular location in a term.

Alternatively, a context can be thought of and represented as a list of elementary contexts of the form $\lambda x.\Box$, $\Box\, t$, $t\, \Box$, called *frames*, on the path between the root of the term and the hole. This is the representation that we use (see Listing 6). We often think of this representation as a sequence of navigation steps made from the root of the term in order to find a redex to contract.

Listing 6 Inductive definition of contexts.

```
Inductive frame : Type :=
|  Lam : string → frame
| Rapp : term → frame
| Lapp : term → frame.

Definition context : Type := list frame.
```

Technically, a general context (C above) is a *folded* representation of a list of frames. Correspondingly, reduction semantics introduces a `plug` function (often left implicit) to reconstruct a term from a context and a term put in its hole. It is denoted as $C[t]$, where C is a context and t is a term.

$$\Box[s] = s \qquad (C\, t)[s] = C[s]\, t \qquad (t\, C)[s] = t\, C[s] \qquad (\lambda x.C)[s] = \lambda x.C[s]$$

Just as we need to define normal forms as term families to specify strategies, we also need to restrict the general grammar of contexts. We do it by defining specific context families, considered as sets of contexts, and therefore formalized as functions of type `context → Prop`.

[2] McBride showed that it can be derived through simple symbolic differentiation [21].

The common examples of reduction contexts are collected as a cheat sheet in Figure 5, where we use their familiar specification as terms with a hole. For example, CBN defines standard call-by-name contexts, that in the frames representation consist of just one type of frames: $\Box\, t$. For another example, Weak defines a family of contexts representing a non-deterministic weak strategy that does not reduce under lambda abstractions.

$$
\begin{array}{rl}
\text{CBN} \ni & Q ::= \Box \mid Q\, t \\
\text{Head} \ni & H ::= Q \mid \lambda x.H \\
\text{HS} \ni & J ::= \Box \mid J\, t \mid \lambda x.J \\
\text{LO} \ni & N ::= \overline{N} \mid \lambda x.N \\
\text{LS} \ni & M ::= J \mid \overline{M} \mid \lambda x.M \\
\text{LI} \ni & L ::= \Box \mid L\, t \mid n\, L \mid \lambda x.L \\
\text{Weak} \ni & W ::= \Box \mid t\, W \mid W\, t \\
\text{LCBV} \ni & {}^{\lambda}V^{t} ::= \Box \mid (\lambda x.t)\, {}^{\lambda}V^{t} \mid {}^{\lambda}V^{t}\, t \\
\text{LCBW} \ni & {}^{w}V^{t} ::= \Box \mid w\, {}^{w}V^{t} \mid {}^{w}V^{t}\, t \\
\text{SDET} \ni & V^{\lambda} ::= \Box \mid V^{\lambda}\, (\lambda x.t) \\
\text{LOW} \ni & {}^{i}V^{t} ::= \Box \mid i\, {}^{i}V^{t} \mid {}^{i}V^{t}\, t \\
\text{LLCBW} \ni & {}^{w}_{a}V^{t}_{w} ::= {}^{w}V^{t} \mid \overline{{}^{w}_{a}V^{t}_{w}} \mid \lambda x.{}^{w}_{a}V^{t}_{w} \\
\text{RLCBW} \ni & {}^{w}_{i}V^{t}_{n} ::= {}^{w}V^{t} \mid \overline{{}^{w}_{i}V^{t}_{n}} \mid \lambda x.{}^{w}_{i}V^{t}_{n} \\
\text{LRCBW} \ni & {}^{t}_{a}V^{w}_{w} ::= {}^{t}V^{w} \mid \overline{{}^{t}_{a}V^{w}_{w}} \mid \lambda x.{}^{t}_{a}V^{w}_{w} \\
\text{RRCBW} \ni & {}^{t}_{i}V^{w}_{n} ::= {}^{t}V^{w} \mid \overline{{}^{t}_{i}V^{w}_{n}} \mid \lambda x.{}^{t}_{i}V^{w}_{n} \\
\text{SCBW} \ni & {}^{t}_{i}V^{t}_{w} ::= W \mid \overline{{}^{t}_{i}V^{t}_{w}} \mid \lambda x.{}^{t}_{i}V^{t}_{w}
\end{array}
\qquad
\begin{array}{rl}
\text{Rigid} \ni & \overline{C} ::= \Box \mid \overline{C}\, t \mid r\, C \\
\overline{\text{LO}} \ni & \overline{N} ::= \Box \mid \overline{N}\, t \mid a\, N \\
\overline{\text{LS}} \ni & \overline{M} ::= \overline{M}\, t \mid a\, M \\
\text{RI} \ni & R ::= \Box \mid R\, n \mid t\, R \mid \lambda x.R \\
\text{RCBV} \ni & {}^{t}V^{\lambda} ::= \Box \mid t\, {}^{t}V^{\lambda} \mid {}^{t}V^{\lambda}\, (\lambda x.t) \\
\text{RCBW} \ni & {}^{t}V^{w} ::= \Box \mid t\, {}^{t}V^{w} \mid {}^{t}V^{w}\, w \\
 & \\
 & \\
 & \\
 & \\
 & \overline{{}^{w}_{a}V^{t}_{w}} ::= a\, {}^{w}_{a}V^{t}_{w} \mid \overline{{}^{w}_{a}V^{t}_{w}}\, w \\
 & \overline{{}^{w}_{i}V^{t}_{n}} ::= i\, {}^{w}_{i}V^{t}_{n} \mid \overline{{}^{w}_{i}V^{t}_{n}}\, n \\
 & \overline{{}^{t}_{a}V^{w}_{w}} ::= a\, {}^{t}_{a}V^{w}_{w} \mid \overline{{}^{t}_{a}V^{w}_{w}}\, w \\
 & \overline{{}^{t}_{i}V^{w}_{n}} ::= i\, {}^{t}_{i}V^{w}_{n} \mid \overline{{}^{t}_{i}V^{w}_{n}}\, n \\
 & \overline{{}^{t}_{i}V^{t}_{w}} ::= i\, {}^{t}_{i}V^{t}_{w} \mid \overline{{}^{t}_{i}V^{t}_{w}}\, w
\end{array}
$$

Figure 5 Context families cheat sheet.

Even though we use the frame representation, we can still prove that the two representations are equivalent, i.e., that they define the same context family. For example, our formalization of *lcbw*-contexts is shown in Listing 7 as the `L_CBW` function, where `Uniform` is a function checking that all frames in a context are proper *lcbw* frames. On the other hand, we can express the LCBW grammar of Figure 5, and prove the equivalence stated in lemma `L_CBW_grammar`. Let's spell out the meaning of the LCBW grammar in Figure 5 and of the `L_CBW_grammar` lemma: an LCBW context (denoted by ${}^{w}V^{t}$ and `L_CBW`) is a hole or an application of weak normal form (denoted by w and `wnf`) to an LCBW context, or an application of an LCBW context to any term (denoted by t and ●).

Apart from this, we can also formalize and prove other properties, such as those stated in the remaining lemmas in Listing 7: LCBW contexts are weak (`L_CBW_is_Weak`), CBN contexts are exactly the weak head contexts (`CBN_Weak_Head`), and the auxiliary nonterminal in the grammar of leftmost-outermost contexts (denoted LO) describes exactly the rigid leftmost-outermost contexts (`RLO_Rigid_LO`).

Overlined nonterminals inform that they are rigid versions of their regular counterparts, i.e., not headed by a lambda abstraction. Letters around V in nonterminals inform which application frames appear in such call-by-value context family. Superscripts concern the weak fragment and subscripts the remaining part.

Listing 7 Example properties of contexts.

```
Definition L_CBW_frame (f:frame) : Prop :=
  match f with
  | Lapp e => wnf e
  | Rapp e => True
  | _      => False
  end.

Definition L_CBW  : 𝒫 context := Uniform L_CBW_frame.

Definition Lapp_of (t1:𝒫 term) (T2:𝒫 context) (C:context) : Prop :=
  match C with Lapp e1 :: C' => t1 e1 ∧ T2 C' | _ => False end.
Definition Rapp_of (T1:𝒫 context) (t2:𝒫 term) (C:context) : Prop :=
  match C with Rapp e2 :: C' => t2 e2 ∧ T1 C' | _ => False end.

Lemma L_CBW_grammar :
  L_CBW == Hole ∪ Lapp_of wnf L_CBW ∪ Rapp_of L_CBW ●.
Lemma L_CBW_is_Weak : L_CBW ⊆ Weak.
Lemma CBN_Weak_Head : CBN == Weak ∩ Head.
Lemma RLO_Rigid_LO : RLO == Rigid ∩ LO.
```

2.5 Decompositions and Strategies

The purpose of an (effective) reduction strategy is to indicate locations in a term where contraction can occur. It can be done using *decompositions* of a term into a designated subterm and its surrounding context. This approach does not require substitution to prove any of the properties of strategies that we study in this paper.

Strategies are defined as sets of decompositions, as shown in Listing 8. Recomposition is the uncurried `plug` function. With these definitions, we can define that a term is in normal form with respect to a given strategy if there is no decomposition of this term accepted by the strategy (cf. `normal_form`). We can also define that a strategy is deterministic if for any term there exists at most one of its decompositions accepted by the given strategy (cf. `det_strategy`). Our definition of strategy is still quite general and makes it possible to talk about undecidable strategies, but they are out of the scope of our interest.

Listing 8 Definitions of decomposition, strategy, normal forms and determinism.

```
Definition decomposition : Type := context * term.
Definition strategy      : Type := decomposition → Prop.

Definition recompose : decomposition → term := uncurry plug.

Definition normal_form (s:strategy) (t:term) : Prop :=
  ¬ ∃ d, t = recompose d ∧ d ∈ s.

Definition ex_le1 {A:Type} (P : A → Prop) : Prop :=
  ∀ x x', P x → P x' → x = x'.
Notation "∃≤1 x , p" := (ex_le1 (λ x, p))
  (at level 200, right associativity) : type_scope.

Definition det_strategy (s:strategy) : Prop :=
  ∀ t, ∃≤1 d, t = recompose d ∧ d ∈ s.
```

The final aspect to consider is how to recognize that a subterm plugged in the hole of a context can be contracted according to the given strategy. Often, such a term is called a *redex*. In the code, we use the name *contrex* (contractible expression) to stress that it has fit not just reduction, but exactly contraction. For example, we said that in the *lcbw* strategy a redex has to have the form $(\lambda x.t)\, w$. On the other hand, any term of the form $(\lambda x.t_1)\, t_2$ is a redex in *cbn*.

We define β, β_λ, β_w, β_{wh}, ${}_h\beta$, ${}_n\beta_n$ term families denoting accepted subjects of contraction as follows (letter symbols are as in Figure 1: $q \in whnf$, $w \in wnf$, $h \in hnf$, $n \in nf$; later on, the symbols β, β_λ, β_{wh} will also denote strategies that can perform the appropriate contraction only in the empty context):

$$\begin{array}{lll} \beta \ni (\lambda x.t_1)\, t_2 & \beta_w \ni (\lambda x.t)\, w & {}_h\beta \ni (\lambda x.h)\, t \\ \beta_\lambda \ni (\lambda x_1.t_1)\, (\lambda x_2.t_2) & \beta_{wh} \ni (\lambda x.t)\, q & {}_n\beta_n \ni (\lambda x.n_1)\, n_2 \end{array}$$

With these ingredients, we can now define specific strategies, e.g., *cbn* as CBN $\times$ β and *lcbw* as LCBW $\times$ β_w. Example definitions are shown in Listing 9.

Listing 9 Example strategy definitions.

```
Definition    β_contrex : term → Prop := app_of abstraction ●.
Definition βwnf_contrex : term → Prop := app_of abstraction wnf.
Definition hnfβ_contrex : term → Prop := app_of (abs_of hnf) ●.

Definition only_β_contraction : strategy :=  Hole × β_contrex.
Definition                cbn : strategy :=   CBN × β_contrex.
Definition              l_cbw : strategy := L_CBW × βwnf_contrex.
```

3 A trip to the zoo

3.1 Zoo picture description

Figure 6 depicts β-reduction strategies as Cartesian products of context families (located in the topmost row in the figure) and term families (located in the leftmost column). For example, $cbv := \textsc{Weak} \times \beta_\lambda$. There are some crossings that define strategies we are not interested in, e.g., LOW $\times$ β_w. We mark them with "$\cdot$", and we omit dots in the empty rectangles. In general there exist strategies that are not Cartesian products, e.g. an interbreeding: $cbn \cup cbv$, but we have not found it interesting.

Some of the strategies are well known and can be identified by their reduction semantics from the literature: *cbn* as call by name, *no* as normal order, *lcbv* as left-to-right (closed) call by value [8]. Analogously, *rcbv* is right-to-left closed call by value and *cbv* is nondeterministic closed call by value. On the other hand, *cbw* can be identified as non-deterministic open call by value (a.k.a. fireball calculus) and *rcbw* as its right-to-left substrategy [3]. Analogously, *lcbw* is its left-to-right substrategy. In the following, we show that *llcbw* (and the three strategies written underneath it) are deterministic extensions of open call by value to strong call by value, and that *scbw* is their superstrategy, non-deterministic strong call by value.[3]

[3] Strong call by value is understood differently in [1]: it is not a superstrategy of open call by value, it is defined in a language with explicit substitutions and thus it does not fit directly into our zoo. Strong call by value presented here is a strategy studied in [7].

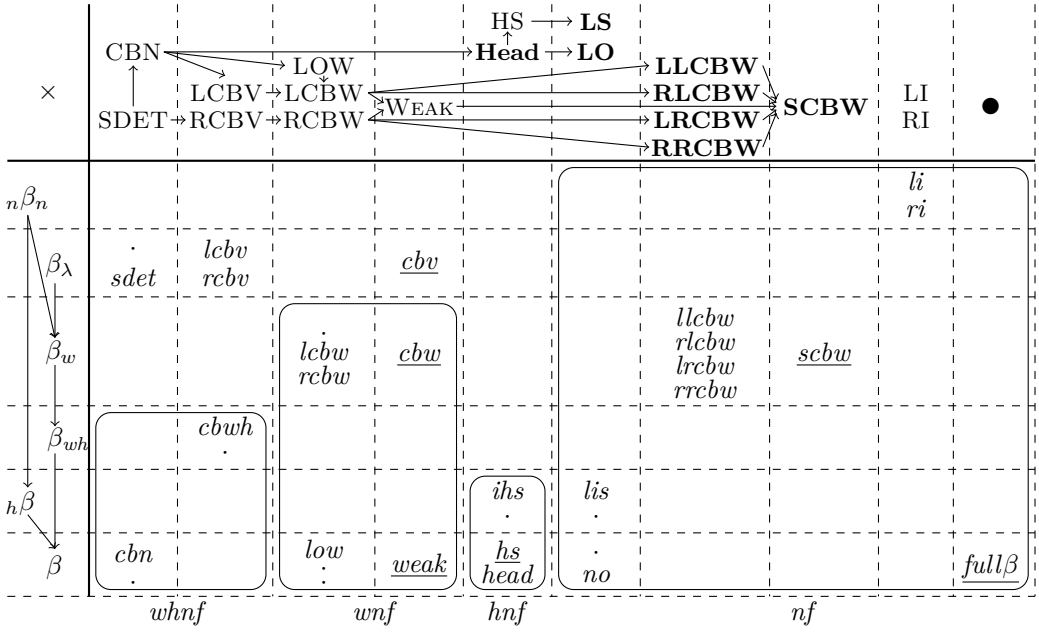


Figure 6 Zoo of strategies of the lambda calculus.

Furthermore, the strategies *weak*, *head*, *full*β, *li*, *ri* are the weak, head, full, leftmost-innermost and rightmost-innermost β-reductions, respectively. The remaining strategies (*sdet*, *cbwh*, *low*, *ihs* and *lis*) are discussed in Section 4.2.

The arrows shown in the topmost row and leftmost column represent family inclusions, such as `L_CBW_is_Weak` : LCBW $\subseteq$ WEAK. Thanks to the monotonicity of the Cartesian product, we can read inclusions of the strategies: *head* $\subseteq$ *no* because HEAD $\subseteq$ LO (and $\beta \subseteq \beta$), or *sdet* $\subseteq$ *lcbv* because SDET $\subseteq$ CBN $\subseteq$ LCBV $\subseteq$ LCBW and $\beta_\lambda \subseteq \beta_w$.

What is left to be demonstrated, is that the strategies living in the same pen (rounded rectangle) have the same normal forms, and these normal forms are written in the bottom of the figure under each pen. For example, normal forms of *head*, *ihs* and *hs* (but not of the dot for HEAD $\times\, {}_h\beta$) are exactly *hnf*. The strategies underlined in the figure are non-deterministic, and the non-underlined ones are deterministic. Context families in bold are non-uniform and the other ones are uniform (see definitions in Section 3.3).

3.2 Example reductions

In this section we show how strategies behave when we feed them with terms. To this end, we define some standard terms and their abbreviations[4]: $I := \lambda x.x$, $K := \lambda x.\lambda y.x$, $\omega := \lambda x.x\,x$, $\Omega := \omega\omega$, $(t_1, t_2) := \lambda f.f\,t_1\,t_2$, $(t,) := \lambda f.f\,t$, $\pi_1 := (K,)$.

The term $(I\,K)\,I$ reduces in *cbn*, but it is not contractible: $K\,I \overset{cbn}{\leftarrow} (I\,K)\,I \not\to_\beta$. In *cbn* II reduces to I which is its normal form ($II \overset{cbn}{\to} I \overset{cbn}{\not\to}$), because the decomposition of II into $\Box$ and II is a *cbn*-decomposition, while the decomposition of I into $\Box$ and I is not. The reason is that I is not contractible – it is not in the term family β.

[4] Notation $(t,)$ for 1-tuples comes from Python.

Call by name and call by value can go separate ways on the same term as shown by the following example: $\omega\,K \overset{cbn}{\leftarrow} I\,(\omega\,K) \overset{cbv}{\rightarrow} I\,(K\,K)$. Closed call by value can be stuck where the open is not: $\overset{lcbv}{\nleftarrow} x\,(I\,x) \overset{lcbw}{\rightarrow} x\,x$. Weak strategies do not allow reduction under lambda abstractions: $(\lambda x.x\,I)\,K \overset{weak}{\nleftarrow} (\lambda x.(\lambda y.x\,y)\,I)\,K \overset{weak}{\rightarrow} (\lambda y.K\,y)\,I$. For the same term, *head* and *ihs* can behave differently: $(\lambda y.K\,y)\,I \overset{head}{\leftarrow} (\lambda x.(\lambda y.x\,y)\,I)\,K \overset{ihs}{\rightarrow} (\lambda x.x\,I)\,K$. Normal-order reduction is known for avoiding nontermination whenever possible while leftmost-innermost and rightmost-innermost fall victim to it very easily: $I \overset{no}{\leftarrow} (\lambda x.I)\,(\Omega,) \overset{li,ri}{\rightarrow} (\lambda x.I)\,(\Omega,)$.

A very small strategy $sdet == cbn \cap rcbv$ is capable of performing pair projection: $\pi_1\,(K,I) \overset{sdet}{\rightarrow} (K,I)\,K \overset{sdet}{\rightarrow} K\,K\,I \overset{sdet}{\rightarrow} (\lambda y.K)\,I \overset{sdet}{\rightarrow} K$ and nontermination: $\Omega \overset{sdet}{\rightarrow} \Omega$.

3.3 Uniformity and determinism

There are some general properties that can be expressed and studied uniformly for any reduction strategy; by way of example we discuss two of such properties, illustrated already in Figure 6. Sestoft [31] observed that some strategies are uniform in the sense that the definition of such a strategy (in the big-step semantics) depends inductively only on that strategy itself, while other strategies are hybrid: they use some uniform ones as substrategies. Uniform and hybrid strategies were studied e.g. by García-Pérez and Nogueira [16]. Biernacka et al. [8] define a strategy to be uniform if its context family can be defined with a grammar with only one nonterminal symbol. That means that the shape of context frames can be uniformly checked for each frame separately, and it is reflected by our definition of `Uniform` (see Listing 10). To prove uniformity it is enough to define the given family in a uniform way. Disproving it is more complicated: one must show that every grammar with only one non-terminal symbol that generates all contexts in the family generates also some contexts that are not in this family.

Another property of interest concerns (non-)determinism. To refute determinism of a given strategy it is enough to show two different decompositions of the same term accepted by the strategy. Moreover, superstrategies of a non-deterministic strategy are non-deterministic (this property is reflected by `det_strategy_variance` in Listing 10 that says if s_1 is a substrategy of s_2 and s_2 is deterministic then s_1 is also deterministic). On the other hand, showing determinism is more demanding because it requires equating two decompositions of the same term accepted by a strategy. In order to prove it for more complex strategies, we will employ more sophisticated reasoning.

Listing 10 Uniformity and non-determinism.

```
Fixpoint Uniform (F:frame → Prop) (C:context) : Prop :=
  match C with
  | []     => True
  | f :: C => F f ∧ Uniform F C
  end.

Theorem uniform_strategies : ∀ C, In C [SDET; CBN; R_CBV; L_CBV;
  R_CBW; L_CBW; LOW; Weak; HS; RI; LI; ●] → ∃ F, C == Uniform F.
Theorem non_uniform_strategies : ∀ C, In C [Head; LO; LS;
  LL_CBW; LR_CBW; RL_CBW; RR_CBW; SCBW] → ¬ ∃ F, C == Uniform F.

Lemma det_strategy_variance : Proper (subset --> impl) det_strategy.

Theorem nondeterministic_strategies :
  ∀ s, In s [cbv; cbw; weak; hs; scbw; full_β] → ¬ det_strategy s.
```

4 Phased strategies

Reduction strategies are often implemented as procedures for locating the next redex to be contracted. For example, call by name can be defined as a strategy that reduces an application $t_1\,t_2$ by first reducing t_1 to weak-head normal form and then trying β-contraction. But then it is not clear if such a definition has anything in common with the product CBN $\times\,\beta$, as defined in our formalization. In this section we present a novel semantic format of phased strategies that allows concise description of strategies in this (operational) spirit and facilitates algebraic reasoning about their properties. In particular we show that the two definitions of *cbn* (and similar definitions of all the other strategies in the zoo) are equivalent.

The basic idea of a phased strategy is that it is a recursively defined sequence of *phases*, each phase being a description of one small step in the search for the next redex. The phases are easy to read and to implement. For example the phased version of *lcbw* is $\swarrow lcbw; \searrow lcbw; \beta$. It consists of three phases, each of which tries to reduce an application of the form $t_1\,t_2$ (the absence of the constructor $\downarrow$ shows that *lcbw* never reduces under lambda abstractions). The first phase is $\swarrow lcbw$, it says: search for the redex in the left term, t_1, using the *lcbw* strategy. The second phase $\searrow lcbw$ says: if the first phase finds no redex, then continue the search in the right term, t_2, using the *lcbw* strategy. The third phase tries β-contraction when the first two phases find no redex.

Apart from their simplicity, phased strategies have other advantages. They are extremely concise in presentation, and they support equational reasoning that can be used to identify equal strategies that admit different definitions. The sequencing of phases preserves determinism, which greatly simplifies reasoning about deterministic strategies. In contrast to small-step semantics from Section 2.3, defining a phased strategy does not require knowledge of normal forms in this strategy. And in contrast to big-step semantics, the phased format can distinguish between divergent and stuck terms.

4.1 Definitions

The first ingredient of phased strategies is strategy sequencing. It joins two strategies, called phases, into one strategy that either makes a step of the first phase unconditionally, or it makes a step of the second phase – if the term is already in the normal form with respect to the first phase. This definition is formalized in Listing 11. On paper, we use the semicolon to denote sequencing, but in Coq we use double semicolon because the single one is already reserved. Strategy sequencing seems to be quite natural as strategies with the sequencing operator form a monoid.

Listing 11 Strategy sequencing monoid.

```
Definition sequence_strategy (r s: strategy) : strategy :=
  λ d, r d ∨ (normal_form r (recompose d) ∧ s d).
Infix ";;" := sequence_strategy (at level 60, right associativity).

Lemma sequence_strategy_empty_r : ∀ s, s;; ∅ == s.
Lemma sequence_strategy_empty_l : ∀ s, ∅;; s == s.
Lemma sequence_strategy_assoc : ∀ q r s, q;; (r;; s) == (q;; r);; s.
```

We define five unary strategy operators that allow us to peel one top frame from the term and execute the given strategy beneath. For example, $\swarrow cbn$ works only on applications and it can make a *cbn*-step on t_1 in $t_1\,t_2$. Similarly, $\downarrow cbn$ works only on abstractions. Analogously, $\not\swarrow cbn$ can make a *cbn*-step on t_1 in $t_1\,t_2$ given that t_2 is already an abstraction. Formal definitions are given in Listing 12. We also use β, β_λ, β_{wh} to denote strategies that can only perform the contraction on the top term.

Listing 12 Phased strategies constructors.

```
Definition left_strategy (s : strategy) : strategy :=
  λ d, match d with (Rapp  _ :: C, c) => s (C, c) | _ => False end.
Definition right_strategy (s : strategy) : strategy :=
  λ d, match d with (Lapp  _ :: C, c) => s (C, c) | _ => False end.
Definition down_strategy (s : strategy) : strategy :=
  λ d, match d with (Lam   _ :: C, c) => s (C, c) | _ => False end.
Definition left_abs_strategy (s : strategy) : strategy :=
  λ d, match d with (Rapp t2 :: C, c) => abstraction t2 ∧ s (C, c)
                  | _                 => False end.
Definition right_abs_strategy (s : strategy) : strategy :=
  λ d, match d with (Lapp t1 :: C, c) => abstraction t1 ∧ s (C, c)
                  | _                 => False end.

Notation "↙"  := left_strategy.
Notation "↘"  := right_strategy.
Notation "↓"  := down_strategy.
Notation "↙λ" := left_abs_strategy.
Notation "↘λ" := right_abs_strategy.
Notation "'β'"   := only_β_contraction.
Notation "'βλ'"  := only_βλ_contraction.
Notation "'βwh'" := only_βwhnf_contraction.

Definition cbn_phased : strategy := ↙cbn;; β.
```

In Figure 7 we present formal definitions of 27 phased strategies, and at the same time we state 27 theorems formalizing how these strategies are equivalent to 24 strategies from the zoo of Figure 6. For example, the *cbn* strategy of Figure 6 can be shown to be equal to the phased strategy $\swarrow cbn; \beta$, and also to the phased strategy $\beta; \swarrow cbn$ (first line).

$$
\begin{aligned}
cbn &= \swarrow cbn; \beta = \beta; \swarrow cbn & \qquad weak &= \swarrow weak \cup \searrow weak \cup \beta \\
head &= (\beta; \swarrow head) \cup \downarrow head & \qquad lcbv &= \swarrow lcbv; \overset{\lambda}{\searrow} lcbv; \beta_\lambda \\
ihs &= (\swarrow ihs; \beta\ \ ) \cup \downarrow ihs & \qquad rcbv &= \searrow rcbv; \overset{\lambda}{\swarrow} rcbv; \beta_\lambda \\
hs &= \beta \cup \swarrow hs \cup \downarrow hs & \qquad cbv &= \swarrow cbv \cup \searrow cbv \cup \beta_\lambda \\
no &= \qquad (\beta; \swarrow no; \searrow no) \cup \downarrow no & \qquad lcbw &= \swarrow lcbw; \searrow lcbw; \beta \\
lis &= (\swarrow ihs; \beta; \swarrow lis; \searrow lis)\ \cup \downarrow lis & \qquad rcbw &= \searrow rcbw; \swarrow rcbw; \beta \\
full\beta &= \beta \cup \swarrow full\beta \cup \searrow full\beta \cup \downarrow full\beta & \qquad cbw &= (\swarrow cbw \cup \searrow cbw); \beta \\
li &= (\swarrow li; \searrow li; \beta) \cup \downarrow li & \qquad llcbw &= (lcbw;\ \swarrow llcbw;\ \searrow llcbw) \cup \downarrow llcbw \\
ri &= (\searrow ri; \swarrow ri; \beta) \cup \downarrow ri & \qquad rlcbw &= (lcbw;\ \searrow rlcbw; \swarrow rlcbw) \cup \downarrow rlcbw \\
cbwh &= \swarrow cbwh; \overset{\lambda}{\searrow} cbwh; \beta_{wh} & \qquad lrcbw &= (rcbw;\ \swarrow lrcbw;\ \searrow lrcbw) \cup \downarrow lrcbw \\
sdet &= \overset{\lambda}{\swarrow} sdet; \beta_\lambda = \beta_\lambda; \overset{\lambda}{\swarrow} sdet & \qquad rrcbw &= (rcbw;\ \searrow rrcbw; \swarrow rrcbw) \cup \downarrow rrcbw \\
low &= \beta; \swarrow low; \searrow low = \swarrow low; \beta; \searrow low & \qquad scbw &= (cbw; (\swarrow scbw \cup \searrow scbw)) \cup \downarrow scbw
\end{aligned}
$$

Figure 7 Phased forms.

Phased strategy operators exhibit some algebraic properties that ease reasoning about them. Some examples are shown in Listing 13. Normal forms of a sequenced strategy are exactly terms that are normal w.r.t. both phases (`normal_form_sequence_strategy`).

The left phase of a weak strategy on the left application branch commutes with a β-phase (`left_weak_strategy_β_contraction_commutative`). It is because β needs an abstraction on the left to contract, and weak strategies do not reduce abstractions. This is why *cbn* has two different phased forms. Unary phased operators are distributive over sequencing (`phase_distributive_over_sequencing`). If a strategy is sequenced with its superstrategy that is deterministic then the obtained strategy is equivalent to the superstrategy alone (`deterministic_extension`). To prove that constructively, we need the decidability of the smaller strategy to decide if the term is its normal form in the proof of the right-to-left inclusion.

Listing 13 Example properties of phased strategy operators.

```
Lemma normal_form_sequence_strategy : ∀ r s,
  normal_form (r;; s) == normal_form r ∩ normal_form s.
Lemma left_weak_strategy_β_contraction_commutative : ∀ w,
  w ⊆ weak →↙ w;; β == β;; ↙ w.
Theorem phase_distributive_over_sequencing : ∀ X,
  In X [↙;↘;↓;↙λ;↘λ] → ∀ s s', (X s;; X s') == X (s;; s').
Theorem deterministic_extension : ∀ s1 s2, (∀ x, {x ∈ s1} + {x ∉ s1})
  → det_strategy s2 → s1 ⊆ s2 → s1;; s2 == s2.
```

4.2 Benefits of phased strategies

In order to see the benefits of phased formulation of the strategies, we first establish the equalities from Figure 7. For many strategies, if we want to define their reduction semantics we need to discover what normal forms are with respect to the strategy we are defining. Most of the time, we prove the equality between reductive form and phased form by induction over terms, where the induction hypothesis is extended with the equality between a fixed family of normal forms and normal forms of the phased form. Then we get theorems about normal forms as corollaries. We collect them in Listing 14.

Listing 14 Normal forms and determinism theorems.

```
Theorem weak_head_normal_forms : ∀ s, In s [cbn; cbwh] →
  normal_form s == whnf.
Theorem weak_normal_forms : ∀ s, In s [l_cbw; r_cbw; cbw; low; weak]
  → normal_form s == wnf.
Theorem head_normal_forms : ∀ s, In s [head; ihs] →
  normal_form s == hnf.
Theorem full_normal_forms : ∀ s, In s [no; lis; ll_cbw; lr_cbw;
  rl_cbw; rr_cbw; scbw; li; ri; full_β] → normal_form s == nf.

Theorem deterministic_strategies : ∀ s, In s
  [sdet; cbn; l_cbv; r_cbv; cbwh; l_cbw; r_cbw; low; ihs; head;
   lis; no; ll_cbw; lr_cbw; rl_cbw; rr_cbw; li; ri] → det_strategy s.
```

As the phased form is affirmed, determinism of the strategies can be proven by simple, repetitive inductions over terms. This seems to be a more structured approach than ad-hoc proofs of determinism (cf. the proof of determinism of *rrcbw* in [7]).

Similarly, the reduction semantics of *lis* $=$ LS $\times$ ${}_h\beta$ (leftmost-innermost-spine) is quite involved (cf. Figure 5). Thanks to the phased formulation, it is clearer how it works and that the reduction semantics formulation is indeed correct (cf. Figure 7). We can see that *head* and *ihs* (innermost-head-spine) are two different deterministic, head-reducing strategies. Both are extended to two different full-reducing strategies: *lis* and *no*. The study of *ihs* and

lis, studied earlier by Barendregt et al. [6], is particularly interesting because they exhibit some call-by-need traits. For example, their decomposition function reaches the head variable before any reduction takes place.

The framework of phased strategies facilitates the study of new strategies. It is widely known that *cbn* performs weak-head reduction. We have formulated the *cbwh* strategy that performs weak-head reduction of arguments before they are substituted, and we have shown that it is also a weak-head-reducing strategy. Thus, it enjoys Accattoli and Guerrieri's harmony property [2].

By swapping the phases of *lcbw* and *rcbw*, we have discovered the *low* (leftmost-outermost-weak) strategy. It resembles *no* but does not go under lambda abstractions. It is a complete weak-reducing strategy, i.e., it always reaches a weak normal form if it exists.

We have formulated the *sdet* strategy that is a substrategy of all strategies mentioned in Figure 6, except for *ihs*, *lis*, *li*, and *ri*. This strategy is sufficient to run Dal Lago and Accattoli's simulation of Turing machines [19], and so all of its superstrategies are also sufficient.

Finally, in Figure 8 we show an example of algebraic reasoning, where we use some of the algebraic laws that we proved. This example demonstrates that *lrcbw* (right-to-left-to-right call by value), which is one of the four self-evident deterministic strong call-by-value evaluation orders, has a special phased form with left weak phase optimized away. It corresponds to a special optimization in its implementation by an abstract machine [7]. Similarly, we can show that *scbw* is a conservative extension of *cbw*, and formalize it as $scbw = cbw; scbw$. Thus *scbw* intuitively inherits the strong confluence (diamond property) of *cbw* [2].

$$
\begin{aligned}
lrcbw &= (rcbw; \swarrow lrcbw; \searrow lrcbw) \cup \downarrow lrcbw \overset{\texttt{sequence\_strategy\_assoc}}{=\!=} \\
&= (\searrow rcbw; \swarrow rcbw; \beta; \swarrow lrcbw; \searrow lrcbw) \cup \downarrow lrcbw \overset{\texttt{left\_weak\_strategy\_}\beta\texttt{\_...\_commutative}}{=\!=} \\
&= (\searrow rcbw; \beta; \swarrow rcbw; \swarrow lrcbw; \searrow lrcbw) \cup \downarrow lrcbw \overset{\texttt{phase\_distributive\_over\_sequencing}}{=\!=} \\
&= (\searrow rcbw; \beta; \swarrow (rcbw; lrcbw), \searrow lrcbw) \cup \downarrow lrcbw \overset{\texttt{deterministic\_extension}}{=\!=} \\
&= (\searrow rcbw; \beta; \swarrow lrcbw; \searrow lrcbw) \cup \downarrow lrcbw
\end{aligned}
$$

Figure 8 An equational reasoning on the *lrcbw* phased form.

4.3 Phased strategies on their own

Phased strategies as presented above depend on the prior definitions of strategies. Nevertheless, there are properties inviting to define them independently. Listing 15 shows that *cbn* is the strategy unique up to set equality that satisfies the equation $cbn = \swarrow cbn; \beta$. From this follows that *cbn* is the smallest and the largest of such strategies. However, these definitions (with $\bigcap$ and $\bigcup$) were again impractical because we could not prove the identity of normal forms without use of *cbn*.

Listing 15 Uniqueness of *cbn*.

```
Definition cbn_eqn (s : strategy) : Prop := s == ↙s;; β.
Lemma       cbn_unique : ∀ s, cbn_eqn s → s == cbn.
Theorem   cbn_ind_form : cbn == ⋂ cbn_eqn.
Theorem cbn_coind_form : cbn == ⋃ cbn_eqn.
```

Nonetheless, a standalone definition in Coq is possible. We can define counterparts of strategy operators working on decomposition functions (e.g., `sequence_decompose`). Then we can define a decomposition function that corresponds to a phased form, take it as a base of the strategy and prove the set equality of the strategies as presented in Listing 16.

Listing 16 Independent phased definition of *cbn*

```
Fixpoint cbn_decompose (t : term): option decomposition :=
  sequence_decompose
    (left_decompose cbn_decompose)
    contrex_decompose
  t.
Definition decompose_strategy f := λ d, f (recompose d) = Some d.
Definition cbn_decomposition := decompose_strategy cbn_decompose.
Theorem cbn_decompose_form : cbn == cbn_decomposition.
```

5 Conclusion

We have presented a minimalistic, concise formalization of a class of reduction strategies in the λ-calculus, that are representable as term decompositions. It can be extended to richer languages based on the lambda calculus, or adapted to other term-rewriting formalisms. We have collected and systematized existing strategies, and shown how some of their properties can be proved in our framework. Finally, we have introduced a novel semantic format of phased strategies that enables simple equational reasoning about strategies. As future work, we plan to accommodate further strategies within our framework, such as variants of call-by-need strategies, or optimal strategies [20, 4] and study abstract machines derived from phased forms.

References

1 Beniamino Accattoli, Andrea Condoluci, and Claudio Sacerdoti Coen. Strong call-by-value is reasonable, implosively. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*, pages 1–14. IEEE, 2021. doi:10.1109/LICS52264.2021.9470630.

2 Beniamino Accattoli and Giulio Guerrieri. Open call-by-value. In Atsushi Igarashi, editor, *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, volume 10017 of *Lecture Notes in Computer Science*, pages 206–226, 2016. doi:10.1007/978-3-319-47958-3_12.

3 Beniamino Accattoli and Giulio Guerrieri. Abstract machines for open call-by-value. *Sci. Comput. Program.*, 184, 2019. doi:10.1016/j.scico.2019.03.002.

4 Thibaut Balabonski. Weak optimality, and the meaning of sharing. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 263–274. ACM, 2013. doi:10.1145/2500365.2500606.

5 Hendrik Pieter Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985.

6 Hendrik Pieter Barendregt, Richard Kennaway, Jan Willem Klop, and M. Ronan Sleep. Needed reduction and spine strategies for the lambda calculus. *Inf. Comput.*, 75(3):191–231, 1987. doi:10.1016/0890-5401(87)90001-0.

7 Malgorzata Biernacka, Dariusz Biernacki, Witold Charatonik, and Tomasz Drab. An abstract machine for strong call by value. In Bruno C. d. S. Oliveira, editor, *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings*, volume 12470 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 2020. doi:10.1007/978-3-030-64437-6_8.

8 Malgorzata Biernacka, Witold Charatonik, and Klara Zielinska. Generalized refocusing: From hybrid strategies to abstract machines. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK*, volume 84 of *LIPIcs*, pages 10:1–10:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.FSCD.2017.10.

9 Arthur Charguéraud. Pretty-big-step semantics. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2013. doi:10.1007/978-3-642-37036-6_3.

10 Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics, 2004.

11 Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009. URL: http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11885.

12 Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992. doi:10.1016/0304-3975(92)90014-7.

13 Yannick Forster, Fabian Kunze, and Marc Roth. The weak call-by-value λ-calculus is reasonable for both time and space. *Proc. ACM Program. Lang.*, 4(POPL):27:1–27:23, 2020. doi:10.1145/3371095.

14 Yannick Forster, Fabian Kunze, Gert Smolka, and Maximilian Wuttke. A mechanised proof of the time invariance thesis for the weak call-by-value λ-calculus. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 19:1–19:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ITP.2021.19.

15 Yannick Forster and Gert Smolka. Call-by-value lambda calculus as a model of computation in coq. *J. Autom. Reason.*, 63(2):393–413, 2019. doi:10.1007/s10817-018-9484-2.

16 Álvaro García-Pérez and Pablo Nogueira. On the syntactic and functional correspondence between hybrid (or layered) normalisers and abstract machines. *Sci. Comput. Program.*, 95:176–199, 2014. doi:10.1016/j.scico.2014.05.011.

17 Gérard P. Huet. Residual theory in lambda-calculus: A formal development. *J. Funct. Program.*, 4(3):371–394, 1994. doi:10.1017/S0956796800001106.

18 Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987. doi:10.1007/BFb0039592.

19 Ugo Dal Lago and Beniamino Accattoli. Encoding turing machines into the deterministic lambda-calculus. *CoRR*, abs/1711.10078, 2017. arXiv:1711.10078.

20 Jean-Jacques Lévy. Optimal reductions in the lambda-calculus. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 160–191. Academic Press, 1980.

21 Conor McBride. The derivative of a regular type is its type of one-hole contexts (extended abstract), 2001.

22 James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *J. Autom. Reason.*, 23(3-4):373–409, 1999. doi:10.1023/A:1006294005493.

23 Michael Norrish. Mechanising lambda-calculus using a classical first order theory of terms with permutations. *High. Order Symb. Comput.*, 19(2-3):169–195, 2006. doi:10.1007/s10990-006-8745-7.

24 Lawrence C. Paulson. *ML for the working programmer (2. ed.)*. Cambridge University Press, 1996.

25 Frank Pfenning. A proof of the church-rosser theorem and its representation in a logical framework, 1992. URL: `https://www.cs.cmu.edu/~fp/papers/cr92.pdf`.

26 Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations*. Software Foundations series, volume 2. Electronic textbook, May 2018. Version 5.5.

27 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`.

28 Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebraic Methods Program.*, 60-61:17–139, 2004.

29 Casper Bach Poulsen and Peter D. Mosses. Deriving pretty-big-step semantics from small-step semantics. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 270–289. Springer, 2014. `doi:10.1007/978-3-642-54833-8_15`.

30 John C. Reynolds. Definitional interpreters for higher-order programming languages. *High. Order Symb. Comput.*, 11(4):363–397, 1998. `doi:10.1023/A:1010027404223`.

31 Peter Sestoft. Demonstrating lambda calculus reduction. In Torben Æ. Mogensen, David A. Schmidt, and Ivan Hal Sudborough, editors, *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th birthday]*, volume 2566 of *Lecture Notes in Computer Science*, pages 420–435. Springer, 2002. `doi:10.1007/3-540-36377-7_19`.

32 Natarajan Shankar. A mechanical proof of the church-rosser theorem. *J. ACM*, 35(3):475–522, 1988. `doi:10.1145/44483.44484`.

33 The Coq Development Team. The Coq proof assistant, January 2021. `doi:10.5281/zenodo.4501022`.

34 Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In Aart Middeldorp, editor, *Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–362. Springer, 2001. `doi:10.1007/3-540-45127-7_27`.

Seventeen Provers Under the Hammer

Martin Desharnais
Graduate School of Computer Science, Saarland Informatics Campus, Saarbrücken, Germany
Max-Planck-Institut für Informatik, Saarland Informatics Campus, Saarbrücken, Germany

Petar Vukmirović
Vrije Universiteit Amsterdam, The Netherlands

Jasmin Blanchette
Vrije Universiteit Amsterdam, The Netherlands
Max-Planck-Institut für Informatik, Saarland Informatics Campus, Saarbrücken, Germany

Makarius Wenzel
Augsburg, Germany

Abstract

One of the main success stories of automatic theorem provers has been their integration into proof assistants. Such integrations, or "hammers," increase proof automation and hence user productivity. In this paper, we use Isabelle/HOL's Sledgehammer tool to find out how useful modern provers are at proving formulas in higher-order logic. Our evaluation follows in the steps of Böhme and Nipkow's Judgment Day study from 2010, but instead of three provers we use 17, including SMT solvers and higher-order provers. Our work offers an alternative yardstick for comparing modern provers, next to the benchmarks and competitions emerging from the TPTP World and SMT-LIB.

2012 ACM Subject Classification Computing methodologies → Theorem proving algorithms

Keywords and phrases Automatic theorem proving, interactive theorem proving, proof assistants

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.8

Supplementary Material *Dataset*: `https://doi.org/10.5281/zenodo.5940084`

Funding This research has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka).
Jasmin Blanchette: has received funding from the Netherlands Organization for Scientific Research (NWO) under the Vidi program (project No. 016.Vidi.189.037, Lean Forward).

Acknowledgements We are grateful to the maintainers of StarExec for letting us use their service. Developers of automatic theorem provers, including Peter Baumgartner, Ahmed Bhayat, Pascal Fontaine, Konstantin Korovin, Giles Reger, Andrew Reynolds, Philipp Rümmer, Alexander Steen, and Martin Suda have helped us run their systems. Michael Färber, Thibault Gauthier, Konstantin Korovin, Lorenz Leutgeb, Jens Otten, Philipp Rümmer, Stephan Schulz, Hans-Jörg Schurr, Alexander Steen, Martin Suda, Mark Summerfield, Dmitriy Traytel, Josef Urban, and the anonymous reviewers suggested some textual improvements.

1 Introduction

Hammers [15] are tools that integrate automatic theorem provers in proof assistants, to automatically discharge proof obligations. CoqHammer [27], HOLyHammer [35], MizAℝ [58], and Sledgehammer (Section 2) are examples of hammers. They typically consist of three main components in addition to the automatic prover backends themselves:

- The *relevance filter* heuristically selects relevant lemmas (including definitions) based on the current goal, using either an iterative procedure or machine learning.

- The *problem translation* module encodes formulas from the proof assistant's logic to the automatic provers' usually less expressive logics. The encoding should ideally be sound and complete.
- The *proof reconstruction* module translates proofs generated by the automatic provers to proof texts expressed in the proof assistant's language.

There is ample evidence that hammers can be effective. The landmark Judgment Day study by Böhme and Nipkow [20] showed for the first time that a combination of three automatic provers (E [49], SPASS [61], Vampire [38]) could discharge about half of the goals that arise in typical Isabelle/HOL [43] developments. This came as a surprise to most Isabelle users, who had not yet integrated Sledgehammer into their workflow. Since then, it has been adopted by a large majority of users.

For over a decade, Judgment Day has been the most comprehensive evaluation of Sledgehammer, even though it considered only three provers and seven Isabelle theory files. Smaller evaluations in later papers [7, 8, 11–13, 17, 45, 50, 53] cover more provers, but they mostly predate the new generation of higher-order provers: cvc5 [1], Ehoh [60], Leo-III [51], Vampire [9], and Zipperposition [59]. We do not have a clear picture of how well these provers work in a hammer setting. We do not even know whether native support for higher-order logic outperforms encodings.

In this paper, we evaluate 17 modern provers (Section 3), both first- and higher-order. They support a wide range of formats from the TPTP [54] and SMT-LIB [2] families. We first generate problems using a tool called Mirabelle (Section 4), which invokes Sledgehammer on goals originating from 50 Isabelle proof developments. We use the MePo [42] relevance filter, which iteratively selects lemmas in a deterministic fashion based on the goal. To translate the problem, we use Sledgehammer's TPTP and SMT-LIB modules, targeting a wide range of provers, including SMT (satisfiability modulo theories) solvers. Once the problems are generated, we run the provers on them (Section 5). The evaluation yields a large collection of new benchmarks that can be used to tune automatic provers – much larger than Judgment Day, which currently consists of 1240 goals. Our collection of problems, called Seventeen, and the raw evaluation data are archived online.[1]

The evaluation is first and foremost an assessment of the provers and their features and of various encoding schemes. Our setup does not attempt to reconstruct proofs in Isabelle. To guard against incorrect proofs, we use only encodings known to be sound [12] and provers that have shown themselves to be trustworthy over the years. The question of how to reconstruct higher-order proofs in Isabelle is still partly open.

In addition, our evaluation provides raw data that can be used to fine-tune how Sledgehammer uses automatic provers. The evaluation also gives insights into which translation methods are useful in hammers, guiding the development of hammers.

Although it may be tempting to view this paper as a prover competition, there are important differences. In a competition, the developers have full control over how their provers are invoked, and there is a clear protocol for interacting with the organizers. We tried to pass meaningful command-line options and consulted prover developers, but there are no guarantees that we succeeded in finding the best options for all provers. Of course, the success rates we report are only lower bounds on what can be achieved.

[1] https://doi.org/10.5281/zenodo.5940084

Table 1 Output formats supported by Sledgehammer.

Format	Description
FOF	Untyped first-order logic
TF0	Many-sorted first-order logic
TX0	Many-sorted first-order logic with Booleans, if–then–else, and let
SMT2	Many-sorted first-order logic with Booleans, if–then–else, let, and linear arithmetic
TF1	Rank-1 polymorphic first-order logic
TX1	Rank-1 polymorphic first-order logic with Booleans, if–then–else, and let
TH0$^-$	Many-sorted higher-order logic
TH0$^+$	Many-sorted higher-order logic with Hilbert choice, if–then–else, and let
SMT3	Many-sorted higher-order logic with if–then–else, let, and linear arithmetic
TH1$^-$	Rank-1 polymorphic higher-order logic
TH1$^+$	Rank-1 polymorphic higher-order logic with Hilbert choice, if–then–else, and let

2 Sledgehammer

When we invoke Sledgehammer on a goal (i.e., a proof obligation, which might be provable or not), Sledgehammer's relevance filters and problem translation modules come into play. We briefly describe them below. The literature covers them in more detail.

2.1 The Relevance Filter

Given a goal and a cutoff number n, the relevance filter selects a subset of n lemmas among all the lemmas that are currently loaded (typically numbering in the thousands) and ranks them from 1 to n in order of decreasing expected relevance for proving the goal. Isabelle includes three relevance filters.

- MePo [42], named after *Me*ng and *P*aulso*n*, works iteratively starting from the goal. It is superficially similar to the SInE [31] algorithm implemented in several automatic provers (E, Vampire, Zipperposition) but was developed independently.
- MaSh [13], the *Ma*chine learner for *S*ledge*h*ammer, implements two fast machine learning algorithms: naive Bayes and k-nearest neighbors. It learns which lemmas are useful to reason about which symbols and ranks the lemmas accordingly.
- MeSh [13] is a combination of *Me*Po and Ma*Sh*.

While MaSh and MeSh give higher success rates than MePo [13], they are tricky to use properly in our complicated evaluation setup, with 50 different Isabelle developments. They also introduce nondeterminism. Since we are more interested in the relative performance of the provers than in the absolute success rates, we decided to use only MePo for this paper. The MaSh paper [13] has a detailed evaluation of the three filters.

MePo operates on a set of *known symbols*, initialized to consist of the symbols that occur in the goal. Roughly speaking, MePo works as follows:

1. Rank the (thousands of) available lemmas. The more symbols a lemma shares with the goal, and the fewer other symbols it contains, the higher its weight.
2. Select a number of lemmas based on their weights, removing them from the set of available lemmas. If no lemmas have suitably high weights, stop.

Table 2 Type encodings supported by Sledgehammer's TPTP module.

Encoding	Description
g	Polymorphism-preserving encoding based on type guards (predicates)
g?	Lightweight variant of g
g??	Featherweight variant of g
g@	Alternative polymorphism-preserving encoding based on type guards
t	Polymorphism-preserving encoding based on type tags (functions)
t?	Lightweight variant of t
t??	Featherweight variant of t
t@	Alternative polymorphism-preserving encoding based on type guards
$\tilde{\text{g}}$, $\tilde{\text{g}}$?, $\tilde{\text{g}}$??, $\tilde{\text{t}}$, $\tilde{\text{t}}$?, $\tilde{\text{t}}$??	Monomorphizing variants of g, g?, g??, t, t?, t??, respectively

3. Enlarge the set of known symbols to include all the symbols occurring in the newly selected lemmas.
4. If n lemmas have been selected, stop; otherwise, go to step 1.

A further refinement is that symbols are annotated with their types, to increase precision. For example, if the symbol nil of type *nat list* is known, lemmas containing nil of polymorphic type α *list* will be considered relevant, unlike lemmas containing nil of type *bool list*. (Type constructors are written in postfix notation.)

2.2 The TPTP Translation

The TPTP translation module [41] generates problems in TPTP formats FOF, TF0, TX0, TF1, TX1, TH0, and TH1. Moreover, we draw an unofficial distinction between TH0$^-$ and TH0$^+$ variants of TH0 and similarly for TH1. Table 1 presents a brief overview of all these formats, as well as the SMT-LIB formats described below. Note that all higher-order formats support interpreted Booleans. Moreover, although some of the TPTP formats provide interpreted arithmetic types and symbols, the TPTP translation module does not exploit this and maps Isabelle's arithmetic operators to uninterpreted symbols.

When translating Isabelle's logic to the automatic provers' possibly weaker logics, four types of constructs may need to be encoded: types, partial application, λ-abstractions, and Booleans. The encodings are naturally not used in formats that support the respective features; for example, TX0, TX1, TH0, and TH1 support interpreted Booleans, so there is no need to encode them.

- For the *types* and the type classes, some translation schemes encode Isabelle's entire rank-1 polymorphic type system using terms and clauses [12, 41]. An alternative that works particularly well in conjunction with the many-sorted formats TF0, TX0, and TH0 is to iteratively monomorphize the problem [12, Section 5.6]. Monomorphization is generally incomplete [18, Section 2]. Sledgehammer's sound type encodings are listed in Table 2 and described by Blanchette et al. [12].
- The *Boolean* constants False, True, the logical connectives, and the quantifiers are either mapped to their TPTP equivalents or translated to uninterpreted "proxy" symbols. Axioms are provided for reasoning about the proxies (e.g., False $\neq$ True).

- *Partial application* is encoded using a distinguished binary symbol app. For example, rev and rev xs are mapped to rev and app(rev, xs), respectively. As an optimization, if all occurrences of rev take at least one argument, it can be passed directly (e.g., rev(xs)).
- λ-*abstractions* are encoded using SKBCI combinators [57] or λ-lifting [33]. The generated problem may include axioms that define the introduced symbols or leave them *opaque*.

▶ **Example 1.** Consider the Isabelle symbol filter of type $(\alpha \Rightarrow \mathit{bool}) \Rightarrow \alpha\ \mathit{list} \Rightarrow \alpha\ \mathit{list}$ such that filter $p\ xs$ is defined as the sublist of xs that consists of the elements x for which $p\ x$ holds. The following lemma is part of Isabelle's list library:

$$\mathsf{filter}\ P\ (\mathsf{filter}\ Q\ xs) = \mathsf{filter}\ (\lambda x.\ Q\ x \wedge P\ x)\ xs$$

If the target is TF0, and both monomorphization and SKBCI are used, the encoding is

$$\mathsf{filter}_{nat}(P, \mathsf{filter}_{nat}(Q, xs)) = \mathsf{filter}_{nat}(\mathsf{S}_{nat,bool,bool}(\mathsf{B}_{nat,bool,bool\Rightarrow bool}(\mathsf{conj}, Q), P), xs)$$

where the subscripts identify monomorphic instances. In addition, axioms are included to characterize the combinators $\mathsf{S}_{nat,bool,bool}$ and $\mathsf{B}_{nat,bool,bool\Rightarrow bool}$ and the proxy conj. By contrast, if the target is TH1, the lemma is encoded as itself.

2.3 The SMT-LIB Translation

The SMT-LIB translation module [19, Chapter 2] was first developed as part of the *smt* proof method [21] and later added to Sledgehammer [11]. The SMT-LIB 2 standard roughly amounts to TPTP TX0 with arithmetic and other theories. The forthcoming SMT-LIB 3 is expected to support higher-order logic and polymorphism and thus be a superset of TPTP TH1. Both cvc5 and a fork of veriT support a preliminary version of the standard.

The SMT-LIB translation module monomorphizes polymorphic types. This is done even for SMT-LIB 3 output. In addition, for SMT-LIB 2, partial application is encoded using app, and λ-lifting is used. Isabelle's basic arithmetic operators on integers and reals are mapped to the corresponding SMT operators. We use the names SMT2 and SMT3 to refer to the fragments of SMT-LIB 2 and 3 used by Sledgehammer, respectively.

3 The Automatic Provers

For our evaluation, we selected 17 provers that support at least one of Sledgehammer's formats. The formats supported by each prover are listed in Table 3. All of the provers participated in the CASC [56] or the SMT-COMP [4] competition at some point. Except when mentioned otherwise, the provers were not specifically tuned for Sledgehammer problems.

- *agsyHOL* [39] is a higher-order prover based on an intuitionistic sequent calculus and a narrowing engine. It was developed to showcase the technology behind the Agsy [40] proof tool for Agda. We use version 1.0.
- *Beagle* [5] is a first-order prover based on hierarchic superposition, a calculus that generalizes standard superposition (which in turn generalizes resolution) with theory reasoning – in Beagle's case, linear arithmetic reasoning. The main developer, Peter Baumgartner, points out that the prover is stronger on problems that require arithmetic reasoning and that it usually performs worse than state-of-the-art superposition provers on problems without theories. We use version 0.9.50.
- *cocATP*[2] is a Coq-inspired higher-order automatic theorem prover based on the calculus of constructions without inductive constructions. We use version 0.2.0.

[2] http://www.tptp.org/CASC/J7/SystemDescriptions.html#cocATP---0.2.0

Table 3 Input formats supported by the provers.

Prover	FOF	TF0	TX0	SMT2	TF1	TX1	TH0$^-$	TH0$^+$	SMT3	TH1$^-$	TH1$^+$
agsyHOL							✓				
Beagle	✓	✓		✓							
cocATP							✓				
cvc5	✓	✓		✓			✓		✓		
E	✓	✓	✓				✓				
ENIGMA	✓										
iProver	✓	✓	✓	✓							
leanCoP	✓										
Leo-III	✓	✓	✓		✓	✓	✓	✓		✓	✓
Princess	✓	✓		✓							
Satallax							✓				
SPASS	✓										
Vampire	✓	✓	✓	✓	✓	✓	✓			✓	
veriT				✓							
Z3	✓	✓		✓							
Zenon	✓										
Zipperposition	✓	✓			✓		✓			✓	

- *cvc5* is a first-order SMT solver based on CDCL(T) with some support for higher-order logic [1]. It is CVC4's [3] successor. The developers provided us with a portfolio of configurations that are tuned for low timeouts. We use version 0.0.4.
- E [49] is a first-order prover based on the superposition calculus. An extension called Ehoh supports some higher-order constructs [60]. We use E 2.6 with Ehoh. This version can parse TH0 but does not yet implement higher-order unification. Note that Vukmirović is a developer of E.
- *ENIGMA* [34] is a variant of E that uses machine learning for determining the order in which clauses are processed by E, thereby steering the search space traversal. ENIGMA's models were trained on the TPTP library, which could give suboptimal performance on Sledgehammer benchmarks. We use version 0.5.1.
- *iProver* [37] is a first-order prover based on instantiation. Recently, a superposition module was added [29]. The developers prepared a version of the prover trained using HOL-ML [32] on Sledgehammer benchmarks – version post-3.5 (git revision 04c55471083). Compared with 3.5, this version adds an extended parser that supports TF0, TX0, and SMT2 benchmarks.
- *leanCoP* [44] is a first-order prover based on the clausal connection calculus, a goal-oriented refinement of the clausal tableau calculus. Implemented in a few lines of Prolog, it combines extreme minimalism with decent performance. We use version 2.2.
- *Leo-III* [51] is a higher-order prover based on the higher-order paramodulation calculus. Paramodulation is a variant of superposition. The provers E and CVC4 are invoked as "end-game" backends at intervals on a first-order encoding of the current clause set. The main developer, Alexander Steen, provided command-line options suited for our experiments, added support for TX0, TX1, TH0$^+$, and TH1$^+$, and fixed a few issues we discovered. We use version 1.6.6.

- *Princess* [48] is a first-order SMT solver based on a tableau calculus and with support for several arithmetic and nonarithmetic theories. We use version 2021-11-15 with command-line options provided by the developer.
- *Satallax* [24] is a higher-order prover based on a tableau calculus guided by a SAT solver. Ehoh is invoked at regular intervals as an "end-game" backend on an applicative first-order (or λ-free higher-order) encoding of the current clause set. We use version 3.5.
- *SPASS* [61] is a first-order prover based on superposition. We use version 3.9.
- *Vampire* [38] is a higher-order prover based on superposition and instantiation. Its superposition calculus uses SKBCI combinators to represent λ-abstractions [9]. We use version 4.6.1.sl,[3] a customized version provided by the developers, with command-line options also provided by them. This version uses Z3 as a backend.
- *veriT* [23] is a first-order SMT solver based on the CDCL(T) calculus. A prototype supports the SMT3 format [1], but we use the standard version 2021.06-rmx. We invoke veriT with command-line options that were derived by the developers using a custom tool [50, Section 5.1] based on Sledgehammer problems.
- *Z3* [28] is a first-order SMT solver based on the CDCL(T) calculus. We use version 4.8.12. The solver was not optimized for these benchmarks.
- *Zenon* [22] is a first-order prover based on a tableau calculus. We use version 0.7.1.
- *Zipperposition* [59] is a higher-order prover based on λ-superposition, a higher-order variant of the superposition calculus. The E prover is invoked as an "end-game" backend at regular intervals on an applicative first-order (or λ-free higher-order) encoding of the current clause set. We run it in a custom mode designed for low timeouts. We use version 2.1. Note that Vukmirović and Blanchette are developers of Zipperposition.

4 Mirabelle

Mirabelle is a testing and evaluation tool included with Isabelle/HOL since version 2010. It was initially developed Sascha Böhme, Blanchette, and other Isabelle developers as a Perl script and some Standard ML code. It was used for many evaluations of Sledgehammer and automatic provers [6–8, 11–13, 16, 17, 20, 45, 47, 50, 53, 60], starting with Judgment Day [20]. It was also used to generate TPTP, CASC, and SMT-LIB benchmarks.

For Isabelle version 2021-1, Mirabelle was reimplemented by Wenzel and Desharnais. The new tool is implemented as part of Isabelle/Scala and is properly integrated with modern Isabelle concepts such as sessions and parallel proof checking.

The new Mirabelle is a command-line tool that takes four arguments as input:

- *An Isabelle session*: A session is a collection of Isabelle theory files forming a project. For example, each entry of the *Archive of Formal Proofs* (AFP) [14] constitutes a session.
- *A theory filter*: The filter selects a subset of the theory files from the session for further processing. For example, a theory filter may keep all theory files or only a specified one.
- *A goal filter*: The filter selects a subset of the goals within the selected theory files for further processing. For example, a goal filter may keep all goals or only the first n goals from the selected theories.
- *A list of actions*: Each action is applied to each selected goal and produces a final report.

[3] `https://github.com/vprover/vampire/releases/tag/v4.6.1.sl`

Mirabelle runs the specified session using Isabelle, collecting all selected goals. At the end, it applies the actions to the collected goals, using parallelism. Each action application produces some output, which can contain more details than the final reports. A Mirabelle *goal* consists of the Isabelle goals before and after a proof step, the type of proof step (e.g., one-line **by** proof), the theory file, and the goal's position in the file. A Mirabelle *action* consists of two functions: run performs the action, and finalize produces the final report.

Mirabelle includes a number of predefined actions. The one we use is Sledgehammer. It supports the same options as the **sledgehammer** command as well as some Mirabelle-specific ones. These can be specified on the command line. The action can either run the automatic provers or only generate the TPTP or SMT-LIB files without running the provers. It is this latter mode that we use for our evaluation. Other Mirabelle actions include *arith*, *metis* [46], and Quickcheck [26]. Users can register custom actions.

▶ **Example 2.** The following command launches Mirabelle on the `Compiler.thy` file from the `VeriComp` session and runs E for 30 seconds on the first 10 goals:

```
isabelle mirabelle -d '$AFP' -O output -T Compiler -m 10 \
  -A "sledgehammer[provers=e,timeout=30]" VeriComp
```

The Sledgehammer invocation on the first goal is identified by the label `0.sledgehammer goal.by VeriComp.Compiler 61:1935`. The associated action output is

```
succeeded (26348+464) [e]:
Try this: using L1.prog_behaves_def by auto (16 ms)
none (sledgehammer): succeeded (0)
```

The second line gives an Isabelle proof that reconstructs the automatic prover's proof in Isabelle and the time needed for reconstruction.

5 Evaluation

For our evaluation, we employed Mirabelle to generate problems from 5000 goals originating from 50 randomly selected entries of the AFP (100 goals per entry). Our objective was to include goals from various areas of mathematics and computer science representing diverse formalization styles and using various features. Within an entry, the goals were selected at regular intervals, alleviating the issue that consecutive goals tend to be similar. We used the repository revisions b87fcf474e7f of Isabelle (15 January 2022) and e2ae9549a7b0 of the AFP (19 December 2021). The experiments were run on the StarExec Iowa cluster [52] with a CPU timeout of 30 s per problem.

5.1 Base Configuration

Table 4 presents the number of proved problems (out of 5000) for the 17 provers and the different supported input formats. In this and later tables, the maximum of each row is shown in bold, and the maximum of each column is shown in italics.

The problems were generated using a *base configuration* of Sledgehammer, which sets the following options to provide a reasonable baseline for the experiments:

- The *fact_filter* option, which specifies the relevance filter, is set to MePo.
- The *max_facts* option, which controls the target number of Isabelle facts (definitions, lemmas, etc.) to include in generated problems, is set to 512 – a large number chosen to stress the provers.

Table 4 Proved problems for each prover and each input format, using the base configuration.

	FOF	TF0	TX0	SMT2	TF1	TX1	TH0$^-$	TH0$^+$	SMT3	TH1$^-$	TH1$^+$
agsyHOL							**814**				
Beagle	901	**1064**		853							
cocATP							**584**				
cvc5	*2619*	***2779***		*2631*			2522		*2557*		
E	2394	2456	2534				**2557**				
ENIGMA	**2301**										
iProver	2291	**2418**	2211	2144							
leanCoP	**1487**										
Leo-III	1938	**2136**	2084		503	459	1632	*1539*		608	*447*
Princess	1205	**1353**		1260							
Satallax							**1695**				
SPASS	**1550**										
Vampire	2495	**2608**	*2587*	2204	*2471*	*2212*	2179			*1814*	
veriT				**2463**							
Z3	2226	2249		**2252**							
Zenon	**812**										
Zipperposition	2551	2607			1702		***2718***			1674	

- The *induction_rules* option, which controls the translation of induction rules, is set so that these rules are excluded from generated problems.
- The *uncurried_aliases* option, which controls the translation of curried function applications in the TPTP module, is set to "false."
- The *lam_trans* option, which controls the translation scheme for λ-abstractions in the TPTP module, is set to use λ-lifting with defining equations for first-order formats and to keep the λ's as is for higher-order formats.
- The *type_enc* option, which controls the translation scheme for types in the TPTP module, is set as follows: For polymorphic formats, keep the types as is. For monomorphic formats, monomorphize the types. For FOF, use Sledgehammer's efficient `g??` encoding [12].
- The *max_mono_iters* and *max_new_mono_instances* options, which limit the number of axiom instances generated by monomorphization, are set to 3 and 100, respectively.

The other Sledgehammer options are left with their default values [10]. In the next subsections, we will deviate from this baseline to study the effect of various options.

In Table 4, we see that the best prover in the base configuration is cvc5. Remarkably, this success is achieved with the TF0 format and not with the arithmetic-capable SMT2. The situation is similar for iProver, Princess, and Vampire, but not Z3.

Intuitively, we would also expect support for higher-order logic to be beneficial, but this is not always true. E and Zipperposition perform better with the higher-order TH0$^-$ format than with the first-order TF0, but for cvc5, Leo-III, and Vampire the opposite is true. Similar effects are visible with formats supporting if–then–else and let (TX0, TX1, TH0$^+$, TH1$^+$).

Is higher-order logic useful at all? It would appear so. The data underlying Table 4 reveals that among the 3321 goals that are collectively proved by all provers and formats, 146 goals are proved with higher-order formats but not with first-order formats.

Table 5 Running time of each prover's best input format.

Prover	Format	Proved problems	Percentile 25	50	75	90	95	99	100
agsyHOL	TH0$^-$	814	0.5	1.5	4.2	9.0	15.3	24.9	29.5
Beagle	TF0	1064	5.5	7.0	9.6	12.3	13.7	15.4	16.8
cocATP	TH0$^-$	584	2.4	4.9	11.1	20.1	24.6	28.4	30.1
cvc5	TF0	2779	0.2	0.3	0.5	2.1	7.6	24.4	30.0
E	TH0$^-$	2557	0.2	0.4	1.1	9.5	13.5	23.6	29.2
ENIGMA	FOF	2301	2.7	2.9	3.4	4.5	6.2	9.0	15.1
iProver	TF0	2418	1.3	1.3	4.3	7.8	12.8	25.9	29.9
leanCoP	FOF	1487	1.4	1.4	3.5	11.6	12.7	26.3	29.2
Leo-III	TF0	2136	5.5	6.5	7.6	8.7	9.2	10.3	12.0
Princess	TF0	1353	6.8	7.6	8.5	9.2	9.5	9.8	11.0
Satallax	TH0$^-$	1695	2.6	5.8	12.7	20.0	21.1	24.4	28.6
SPASS	FOF	1550	1.7	3.7	10.0	18.1	22.9	28.8	29.7
Vampire	TF0	2608	0.2	0.3	5.3	11.0	12.8	23.0	29.4
veriT	SMT2	2463	0.1	0.1	0.2	1.1	4.6	19.6	29.2
Z3	SMT2	2252	0.2	0.2	0.3	0.6	2.3	17.4	29.8
Zenon	FOF	812	0.6	1.6	3.2	6.3	11.4	24.7	29.0
Zipperposition	TH0$^-$	2718	0.9	2.1	4.4	6.9	10.2	23.1	30.0

Finally, we notice that for all provers that support both TF0 and TF1, the monomorphized TF0 results are better than the TF1 results, and similarly for TX0 vs. TX1, for TH0$^-$ vs. TH1$^-$, and for TH0$^+$ vs. TH1$^+$.

5.2 Running Time

How quickly do the provers find the proofs? Table 5 answers this question for each prover, focusing on the most successful input format for the prover according to Table 4. The median, or 50th percentile, is shown, as well as other percentiles. (Percentiles are calculated using linear interpolation.) For example, the number 12.3 in the Beagle row indicates that 90% of the 1064 problems proved by Beagle are proved within 12.3 s. As in Table 4, the baseline configuration is used. Recall that the timeout is 30 s.

Table 5 shows that if a prover finds a proof, it will likely find it quickly. Some provers are dazzlingly fast; the median time it takes to find a proof is 0.1 s for veriT, 0.2 s for Z3, and 0.3 s for cvc5 and Vampire. Remarkably, Princess proves its 1353 problems within 11 s; the remaining 19 s are unnecessary.

5.3 Number of Facts

An important Sledgehammer option is *num_facts*, which governs the number of facts to be selected by the relevance filter. These facts are included as axioms in the generated problems. The base configuration specifies 512 facts, but Table 6 shows what happens when we change this option (and keep the other options as in the base configuration). Figure 1 depicts the same information graphically for six provers, using a logarithmic x-axis.

The stronger provers tend to peak with more facts than the weaker ones. For example, the strongest prover, cvc5, peaks at 512 facts. Some of the stronger provers (E, Vampire, Zipperposition) implement the SInE [31] algorithm; others simply seem to scale well in the presence of extraneous axioms.

Table 6 Proved problems for each prover's best input format and different numbers of facts.

	Format	16	32	64	128	256	512	1024
agsyHOL	TH0$^-$	1106	**1199**	1128	1006	916	814	610
Beagle	TF0	1256	1544	**1672**	1611	1419	1064	526
cocATP	TH0$^-$	745	830	**837**	793	706	584	440
cvc5	TF0	1309	1704	2117	2496	*2736*	***2779***	2669
E	TH0$^-$	1346	1736	2116	2438	**2590**	2557	2368
ENIGMA	FOF	1297	1670	2004	2246	**2347**	2301	2137
iProver	TF0	1294	1667	2009	2300	**2431**	2418	2303
leanCoP	FOF	1150	1376	1510	**1581**	1562	1487	1356
Leo-III	TF0	1305	1680	2043	2333	**2385**	2136	1344
Princess	TF0	1230	1542	**1734**	1655	1636	1353	172
Satallax	TH0$^-$	1233	1459	1596	1702	**1725**	1695	1632
SPASS	FOF	1273	1578	1778	**1843**	1787	1550	973
Vampire	TF0	1303	1691	2049	2379	2569	**2608**	2576
veriT	SMT2	1312	1683	2020	2307	2446	**2463**	2382
Z3	SMT2	1314	1668	1964	2207	**2310**	2252	2150
Zenon	FOF	955	1048	**1067**	1009	899	812	712
Zipperposition	TH0$^-$	*1376*	*1778*	*2162*	*2504*	2699	**2718**	*2680*

5.4 Encoding of Lambdas

Not all provers support λ-abstractions. How practical are the encodings of λ's implemented in Sledgehammer, and is native support for λ's preferable when available? Table 7, which includes the nine provers that support TF0, presents a very fragmented picture: Some provers prefer λ-lifting ("lift") to SKBCI combinators ("combs"), while others prefer SKBCI combinators or a combination of λ-lifting and combinators. Some prefer opaque definitions, while others work better when the defining equations are present.

Particularly striking is that only two higher-order provers, E and Zipperposition, work best with TH0$^-$ and native λ's, whereas the other three prefer TF0 encodings. This is disappointing; we would have hoped that native higher-order support in a prover pays off.

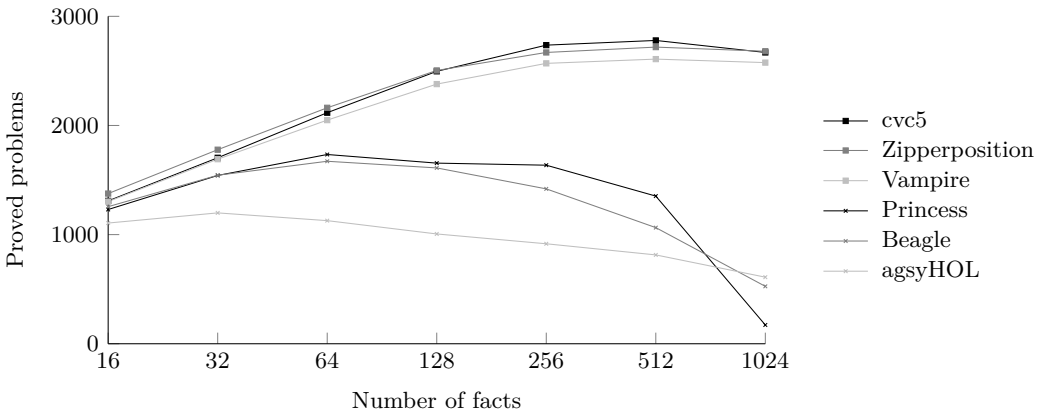


Figure 1 Proved problems for the top provers' best input formats and different numbers of facts.

Table 7 Proved problems by prover for the different encodings of λ-abstractions.

	TF0					TH0⁻
Prover	Lift	Opaque lift	Combs	Opaque combs	Lift & combs	Native
Beagle	1064	**1069**	998	1026	930	
cvc5	*2779*	*2628*	*2838*	*2793*	***2859***	2522
E	2456	2331	2471	2444	2434	**2557**
iProver	2418	2320	2423	**2430**	2388	
Leo-III	2136	2093	2133	**2152**	2061	1632
Princess	1353	**1369**	1319	**1369**	1119	
Vampire	2608	2461	2602	**2610**	2579	2179
Z3	2249	2154	**2326**	2299	2325	
Zipperposition	2607	2118	2635	2590	2642	***2718***

Table 8 Proved problems by prover for the different polymorphism-preserving encodings.

	FOF								TF1
Prover	`g`	`g?`	`g??`	`g@`	`t`	`t?`	`t??`	`t@`	native
Beagle	446	560	637	514	544	**650**	640	487	
cvc5	1660	*2289*	*2552*	1936	*2092*	1989	***2654***	*2216*	
E	1451	2080	**2195**	1646	1800	1820	2162	1390	
ENIGMA	1366	1974	**2063**	1531	1570	1599	2030	1299	
iProver	1587	2016	**2178**	1756	2087	*1992*	2167	1368	
leanCoP	1053	1276	1455	1212	**1565**	1356	1451	763	
Leo-III	871	1075	1504	1076	829	890	**1521**	1252	503
Princess	47	344	**835**	136	214	416	798	454	
SPASS	755	1032	**1194**	935	941	859	1168	784	
Vampire	*1918*	2172	2312	*2032*	1673	1975	2294	1528	***2471***
Z3	1472	2054	2238	1677	1765	1692	**2254**	1844	
Zenon	606	**634**	626	605	424	564	628	403	
Zipperposition	1689	1907	2042	1660	1941	1951	**2267**	1631	1702

For Leo-III and Satallax, which rely on backends, there is a potential explanation: The backends can work directly with TF0 problems but not with TH0. For Vampire, a possible explanation is that the higher-order version of the prover was not as extensively tuned as the first-order version.

5.5 Encoding of Types

Some provers do not support types at all, or they support only monomorphic types. How practical are the type encodings implemented in Sledgehammer, and is native support for types preferable when available? Table 8, which includes the 13 provers that support FOF, presents the results for the *polymorphism-preserving* encodings – i.e., encodings that encode the rank-1 polymorphic type system in its full generality. In contrast, Table 9 presents the results for the *monomorphizing* encodings – i.e., encodings that first heuristically instantiate type variables to eliminate polymorphism. The last column of each table offers a comparison with native TF1 or TF0.

Table 9 Proved problems by prover for the different monomorphizing encodings.

	FOF						TF0
Prover	g̃	g̃?	g̃??	t̃	t̃?	t̃??	native
Beagle	473	842	901	544	887	880	**1064**
cvc5	2033	*2479*	*2619*	*2092*	*2525*	*2687*	***2779***
E	2133	2324	2394	1800	2303	2371	**2456**
ENIGMA	2065	2239	**2301**	1570	2220	2283	
iProver	1963	2171	2291	2087	2285	2278	**2418**
leanCoP	1136	1367	1487	**1565**	1533	1460	
Leo-III	1236	1744	1938	829	1814	1983	**2136**
Princess	159	923	1205	214	1010	1188	**1353**
SPASS	1318	1396	**1550**	941	1329	1533	
Vampire	2302	2426	2495	1673	2385	2484	**2608**
Z3	2127	2200	2226	1765	2048	2223	**2249**
Zenon	812	**818**	812	424	745	788	
Zipperposition	*2360*	2289	2551	1941	2455	2522	**2607**

Our findings are similar to those of Blanchette et al. [12]: Despite being incomplete, monomorphizing encodings outperform the polymorphism-preserving encodings. Overall, the best encodings are the monomorphizing g̃?? and t̃??. But an even better option is to monomorphize the problem and use the many-sorted TF0 format directly. We also see that Leo-III's and Zipperposition's native polymorphism underperforms compared with the polymorphic-preserving FOF encodings. The reason is probably that these provers rely on monomorphic backends. For example, Zipperposition disables its E backend when invoked on a polymorphic problem.

5.6 Portfolio

So far, we have evaluated only the performance of provers in isolation. What happens if we combine them? It turns out that a virtual portfolio consisting of all the provers in all the configurations of Tables 4 to 9, each invoked for 30 s, would prove 3508 goals. This corresponds to a success rate of 70.2%, which is clearly lower than the 76.7% figure obtained in the MaSh paper [13] on the Judgment Day benchmarks. Two possible explanations suggest themselves: The MaSh paper' evaluation used the machine learning filters MaSh and MeSh, which are stronger than MePo, and as observed elsewhere [14, Section 6] Judgment Day might consist of relatively easy goals.

Such a virtual portfolio is not very realistic, but it does give an idea of the state of the art in automated reasoning. A more realistic alternative is to consider the *greedy sequence* of length n. The sequence is obtained by iteratively (1) taking first the (or some) best prover configuration for the goals of interest, and (2) removing all goals proved by this configuration. These two steps are repeated n times, yielding n configurations. The greedy sequence is not necessarily optimal, but it can be computed efficiently.

Table 10 presents the greedy sequence of length 16 based on our experiments. Given 480 s – or 30 s and 16 threads – we can solve 3440 goals (68.8%). This shows how complementary the different provers and options are.

Table 10 A greedy sequence of provers and configurations.

Prover	Format	Configuration	Proved problems
cvc5	TF0	lift & combs	2859
Zipperposition	TH0$^-$	1024 facts	+230
Vampire	TF1	base	+77
veriT	SMT2	1024 facts	+60
E	TX0	base	+51
cvc5	TF0	128 facts	+38
Zipperposition	TH0$^-$	256 facts	+24
Beagle	SMT2	base	+20
cvc5	FOF	t??	+17
Vampire	TF0	1024 facts	+13
Z3	SMT2	1024 facts	+11
E	TH0$^-$	32 facts	+11
Vampire	TH0$^-$	base	+9
E	TH0$^-$	1024 facts	+9
Z3	SMT2	base	+6
Zipperposition	TH0$^-$	base	+5
Total			3440

6 Related Work

The various evaluations of Sledgehammer, cited at the beginning of Section 4, surely constitute the most closely related work. Among these, Böhme and Nipkow's Judgment Day study [20] stands out as the most comprehensive; it considers the following aspects: success rate, running time, proof complexity, and proof minimization. But it is over a decade old. Most of the other evaluations focus on a single new Sledgehammer feature and how it improves the success rate; for example, Sultana et al. [53] evaluate encodings of higher-order features (cf. Section 5.4), and Blanchette et al. [12] evaluate type encodings (cf. Section 5.5).

Similar evaluations for other hammers or hammer-like systems include a three-prover evaluation by Kaliszyk and Urban [36], called MizAR 40, using MizAR, a three-prover evaluation by Czajka and Kaliszyk [27] using CoqHammer, an eight-prover evaluation by Filliâtre [30] using Why3, and a 19-prover evaluation by Brown et al. [25], called GRUNGE, using a custom exporter from HOL4. Among these, GRUNGE is the most similar to our effort, as it includes a large number of provers and exploits support for higher-order logic and polymorphism where available. But there are also several important differences:

- *The benchmarks presented in the GRUNGE paper include only the set of lemmas needed for a proof in HOL4, whereas our benchmarks contain heuristically selected lemmas from Isabelle's libraries.* Including only a typically small set of lemmas makes the benchmarks easier to prove than selecting hundreds of lemmas heuristically. It is less representative of the typical hammer use case, where a proof is not available.
- *The GRUNGE benchmarks correspond to top-level lemmas or theorems in HOL4, whereas our benchmarks correspond to Isabelle goals or subgoals.* Proving a lemma is generally more difficult than proving a goal.
- *GRUNGE's encoding of polymorphism is simple but inefficient, whereas our evaluation relies on state-of-the-art monomorphization.* One of GRUNGE's two translation schemes [25, Section 4.3] corresponds to Sledgehammer's `t` encoding, whose performance was found to be very suboptimal in Section 5.5.

- *GRUNGE uses different provers to our evaluation.* In particular, GRUNGE does not generate SMT-LIB problems, nor does it exploit the recently added support for higher-order logic in cvc5, E, and Vampire. It also misses out on the improvements to Zipperposition over the past three years, which led it to win the higher-order division of the 2020 and 2021 editions of CASC [55, 56].

The top five provers on the GRUNGE benchmarks, with the number of proved problems out of 12 140, are Leo-III with 7090, Vampire with 5929, CVC4 with 5709, E with 5118, and HOLyHammer with 5059. The top prover is only the ninth best according to our Table 4. The discrepancy is easy to explain: For GRUNGE, Leo-III was, next to Zipperposition, the only system that fully supported polymorphic higher-order logic, and all the other systems saw versions of the benchmarks encoded using encodings that were not primarily tuned toward performance of particular provers, whereas for our evaluation we used state-of-the-art encodings of polymorphic types and higher-order features.

As a benchmark suite, Seventeen can be used as a complement to existing libraries, such as the TPTP [54], which includes both first- and higher-order problems in the TPTP formats, and SMT-LIB [2], which consists of first-order problems in SMT-LIB 2 syntax. Various collections of hammer-generated problems exist, such as GRUNGE, Judgment Day, and Miz$\mathbb{AR}$ 40. All these libraries are not necessarily mutually exclusive; for example, many Miz$\mathbb{AR}$- and Sledgehammer-generated problems are in the TPTP, and Sledgehammer-generated problems [47, Section 5] are also part of SMT-LIB.

7 Conclusion

In this paper, we evaluated 17 modern automatic provers, including SMT solvers and higher-order provers, thereby superseding the aging Judgment Day study; and we evaluated several aspects of a hammer, such as the encodings of types and λ-abstractions. In particular, we wanted to determine whether native implementations of various features perform better than encodings, which is a reasonable thing to hope for. We have now updated Sledgehammer's default setup to get the best out of the provers.

Specifically, we found that native support for monomorphic types was beneficial, but the current support for polymorphism is disappointing. Native support for features such as linear arithmetic and higher-order logic helps some provers and is detrimental to others. Overall, we see that simply implementing a feature in a prover is often not enough; to be useful, the feature must be fined-tuned based on benchmarks. Some provers seem to misbehave on Sledgehammer benchmarks, which indeed may look quite different from the problems used by the provers' developers to tune their systems. Since hammers are among the most useful applications of automatic provers, we contend that it is worthwhile to tune provers on hammer-generated benchmarks as a complement to the TPTP and SMT-LIB.

References

1 Haniel Barbosa, Andrew Reynolds, Daniel El Ouraoui, Cesare Tinelli, and Clark W. Barrett. Extending SMT solvers to higher-order logic. In Pascal Fontaine, editor, *CADE-27*, volume 11716 of *LNCS*, pages 35–54. Springer, 2019.

2 Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. URL: `https://www.SMT-LIB.org/`.

3 Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV 2011*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.

4 Clark W. Barrett, Leonardo de Moura, and Aaron Stump. SMT-COMP: satisfiability modulo theories competition. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV 2005*, volume 3576 of *LNCS*, pages 20–23. Springer, 2005.

5 Peter Baumgartner, Joshua Bax, and Uwe Waldmann. Beagle – A hierarchic superposition theorem prover. In Amy P. Felty and Aart Middeldorp, editors, *CADE-25*, volume 9195 of *LNCS*, pages 367–377. Springer, 2015.

6 Alexander Bentkamp, Jasmin Blanchette, Simon Cruanes, and Uwe Waldmann. Superposition for lambda-free higher-order logic. *Log. Meth. Comput. Sci.*, 17(2):1:1–1:38, 2021.

7 Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, and Petar Vukmirović. Superposition for full higher-order logic. In André Platzer and Geoff Sutcliffe, editors, *CADE-28*, volume 12699 of *LNCS*, pages 396–412. Springer, 2021.

8 Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, Petar Vukmirovic, and Uwe Waldmann. Superposition with lambdas. *J. Autom. Reason.*, 65(7):893–940, 2021.

9 Ahmed Bhayat and Giles Reger. A combinator-based superposition calculus for higher-order logic. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *IJCAR 2020, Part I*, volume 12166 of *LNCS*, pages 278–296. Springer, 2020.

10 Jasmin Blanchette. Hammering away: A user's guide to Sledgehammer for Isabelle/HOL, 2021. URL: `https://isabelle.in.tum.de/website-Isabelle2021-1/dist/Isabelle2021-1/doc/sledgehammer.pdf`.

11 Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending Sledgehammer with SMT solvers. *J. Autom. Reason.*, 51(1):109–128, 2013.

12 Jasmin Christian Blanchette, Sascha Böhme, Andrei Popescu, and Nicholas Smallbone. Encoding monomorphic and polymorphic types. *Log. Meth. Comput. Sci.*, 12(4), 2016.

13 Jasmin Christian Blanchette, David Greenaway, Cezary Kaliszyk, Daniel Kühlwein, and Josef Urban. A learning-based fact selector for Isabelle/HOL. *J. Autom. Reason.*, 57(3):219–244, 2016.

14 Jasmin Christian Blanchette, Maximilian Haslbeck, Daniel Matichuk, and Tobias Nipkow. Mining the archive of formal proofs. In Manfred Kerber, editor, *CICM 2015*, volume 9150 of *LNCS*, pages 1–15. Springer, 2015.

15 Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *J. Formaliz. Reason.*, 9(1):101–148, 2016.

16 Jasmin Christian Blanchette, Nicolas Peltier, and Simon Robillard. Superposition with datatypes and codatatypes. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *IJCAR 2018*, volume 10900 of *LNCS*, pages 370–387. Springer, 2018.

17 Jasmin Christian Blanchette, Andrei Popescu, Daniel Wand, and Christoph Weidenbach. More SPASS with Isabelle: Superposition with hard sorts and configurable simplification. In Lennart Beringer and Amy Felty, editors, *ITP 2012*, volume 7406 of *LNCS*, pages 345–360. Springer, 2012.

18 François Bobot and Andrei Paskevich. Expressing polymorphic types in a many-sorted language. In Cesare Tinelli and Viorica Sofronie-Stokkermans, editors, *FroCoS 2011*, volume 6989 of *LNCS*, pages 87–102. Springer, 2011.

19 Sascha Böhme. *Proving Theorems of Higher-Order Logic with SMT Solvers*. PhD thesis, Technische Universität München, 2012.

20 Sascha Böhme and Tobias Nipkow. Sledgehammer: Judgement Day. In Jürgen Giesl and Reiner Hähnle, editors, *IJCAR 2010*, volume 6173 of *LNCS*, pages 107–121. Springer, 2010.

21 Sascha Böhme and Tjark Weber. Fast LCF-style proof reconstruction for Z3. In Matt Kaufmann and Lawrence C. Paulson, editors, *ITP 2010*, volume 6172 of *LNCS*, pages 179–194. Springer, 2010.

22 Richard Bonichon, David Delahaye, and Damien Doligez. Zenon: An extensible automated theorem prover producing checkable proofs. In Nachum Dershowitz and Andrei Voronkov, editors, *LPAR 2007*, volume 4790 of *LNCS*, pages 151–165. Springer, 2007.

23 Thomas Bouton, Diego Caminha Barbosa De Oliveira, David Déharbe, and Pascal Fontaine. veriT: An open, trustable and efficient SMT-solver. In Renate A. Schmidt, editor, *CADE-22*, volume 5663 of *LNCS*, pages 151–156. Springer, 2009.

24 Chad E. Brown. Satallax: An automatic higher-order prover. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *IJCAR 2012*, volume 7364 of *LNCS*, pages 111–117. Springer, 2012.

25 Chad E. Brown, Thibault Gauthier, Cezary Kaliszyk, Geoff Sutcliffe, and Josef Urban. GRUNGE: A grand unified ATP challenge. In Pascal Fontaine, editor, *CADE-27*, volume 11716 of *LNCS*, pages 123–141. Springer, 2019.

26 Lukas Bulwahn. The new Quickcheck for Isabelle – Random, exhaustive and symbolic testing under one roof. In Chris Hawblitzel and Dale Miller, editors, *CPP 2012*, volume 7679 of *LNCS*, pages 92–108. Springer, 2012.

27 Łukasz Czajka and Cezary Kaliszyk. Hammer for Coq: Automation for dependent type theory, 2018.

28 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

29 André Duarte and Konstantin Korovin. Implementing superposition in iProver (system description). In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *IJCAR 2020, Part II*, volume 12167 of *LNCS*, pages 388–397. Springer, 2020.

30 Jean-Christophe Filliâtre. One logic to use them all. In Maria Paola Bonacina, editor, *CADE-24*, volume 7898 of *LNCS*, pages 1–20. Springer, 2013.

31 Krystof Hoder and Andrei Voronkov. Sine qua non for large theory reasoning. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *CADE-23*, volume 6803 of *LNCS*, pages 299–314. Springer, 2011.

32 Edvard K. Holden and Konstantin Korovin. Heterogeneous heuristic optimisation and scheduling for first-order theorem proving. In Fairouz Kamareddine and Claudio Sacerdoti Coen, editors, *CICM 2021*, volume 12833 of *LNCS*, pages 107–123. Springer, 2021.

33 R. J. M. Hughes. Super-combinators: A new implementation method for applicative languages. In *LFP 1982*, pages 1–10. ACM Press, 1982.

34 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, *CICM 2017*, volume 10383 of *LNCS*, pages 292–302. Springer, 2017.

35 Cezary Kaliszyk and Josef Urban. HOL(y)Hammer: Online ATP service for HOL Light. *Math. Comput. Sci.*, 9(1):5–22, 2015.

36 Cezary Kaliszyk and Josef Urban. MizAR 40 for Mizar 40. *J. Autom. Reason.*, 55(3):245–256, 2015.

37 Konstantin Korovin. iProver – An instantiation-based theorem prover for first-order logic (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR 2008*, volume 5195 of *LNCS*, pages 292–298. Springer, 2008.

38 Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *CAV 2013*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013.

39 Fredrik Lindblad. A focused sequent calculus for higher-order logic. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *IJCAR 2014*, volume 8562 of *LNCS*, pages 61–75. Springer, 2014.

40 Fredrik Lindblad and Marcin Benke. A tool for automated theorem proving in Agda. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *TYPES 2004*, volume 3839 of *LNCS*, pages 154–169. Springer, 2004.

41 Jia Meng and Lawrence C. Paulson. Translating higher-order clauses to first-order clauses. *J. Autom. Reason.*, 40(1):35–60, 2008.

42 Jia Meng and Lawrence C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *J. Applied Logic*, 7(1):41–57, 2009.

43 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

44 Jens Otten. leanCoP 2.0 and ileanCoP 1.2: High performance lean theorem proving in classical and intuitionistic logic (system descriptions). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR 2008*, volume 5195 of *LNCS*, pages 283–291. Springer, 2008.

45 Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska, editors, *IWIL-2010*, volume 2 of *EPiC*, pages 1–11. EasyChair, 2012.

46 Lawrence C. Paulson and Kong Woei Susanto. Source-level proof reconstruction for interactive theorem proving. In Klaus Schneider and Jens Brandt, editors, *TPHOLs 2007*, volume 4732 of *LNCS*, pages 232–245. Springer, 2007.

47 Andrew Reynolds and Jasmin Christian Blanchette. A decision procedure for (co)datatypes in SMT solvers. *J. Autom. Reason.*, 58(3):341–362, 2017.

48 Philipp Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *LPAR 2008*, volume 5330 of *LNCS*, pages 274–289. Springer, 2008.

49 Stephan Schulz, Simon Cruanes, and Petar Vukmirovic. Faster, higher, stronger: E 2.3. In Pascal Fontaine, editor, *CADE-27*, volume 11716 of *LNCS*, pages 495–507. Springer, 2019.

50 Hans-Jörg Schurr, Mathias Fleury, and Martin Desharnais. Reliable reconstruction of fine-grained proofs in a proof assistant. In André Platzer and Geoff Sutcliffe, editors, *CADE-28*, volume 12699 of *LNCS*, pages 450–467. Springer, 2021.

51 Alexander Steen and Christoph Benzmüller. Extensional higher-order paramodulation in Leo-III. *J. Autom. Reason.*, 65(6):775–807, 2021.

52 Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec: A cross-community infrastructure for logic solving. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *IJCAR 2014*, volume 8562 of *LNCS*, pages 367–373. Springer, 2014.

53 Nik Sultana, Jasmin Christian Blanchette, and Lawrence C. Paulson. LEO-II and Satallax on the Sledgehammer test bench. *J. Applied Logic*, 11(1):91–102, 2013.

54 Geoff Sutcliffe. The TPTP problem library and associated infrastructure – From CNF to TH0, TPTP v6.4.0. *J. Autom. Reason.*, 59(4):483–502, 2017.

55 Geoff Sutcliffe. The 10th IJCAR automated theorem proving system competition – CASC-J10. *AI Commun.*, 34(2):163–177, 2021.

56 Geoff Sutcliffe and Martin Desharnais. The CADE-28 automated theorem proving system competition – CASC-28. *AI Communications*, 34(4):259–276, 2022.

57 David A. Turner. A new implementation technique for applicative languages. *Softw. Pract. Exper.*, 9:31–49, 1979.

58 Josef Urban, Piotr Rudnicki, and Geoff Sutcliffe. ATP and presentation service for Mizar formalizations. *J. Autom. Reason.*, 50(2):229–241, 2013.

59 Petar Vukmirović, Alexander Bentkamp, Jasmin Blanchette, Simon Cruanes, Visa Nummelin, and Sophie Tourret. Making higher-order superposition work. In André Platzer and Geoff Sutcliffe, editors, *CADE-28*, volume 12699 of *LNCS*, pages 415–432. Springer, 2021.

60 Petar Vukmirović, Jasmin Christian Blanchette, Simon Cruanes, and Stephan Schulz. Extending a brainiac prover to lambda-free higher-order logic. In Tomas Vojnar and Lijun Zhang, editors, *TACAS 2019*, volume 11427 of *LNCS*, pages 192–210. Springer, 2019.

61 Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS version 3.5. In Renate A. Schmidt, editor, *CADE-22*, volume 5663 of *LNCS*, pages 140–145. Springer, 2009.

Formalising Szemerédi's Regularity Lemma in Lean

Yaël Dillies
Department of Pure Mathematics and Mathematical Statistics, University of Cambridge, UK

Bhavik Mehta
Department of Pure Mathematics and Mathematical Statistics, University of Cambridge, UK

Abstract

Szemerédi's Regularity Lemma is a fundamental result in graph theory with extensive applications to combinatorics and number theory. In essence, it says that all graphs can be approximated by well-behaved unions of random bipartite graphs. We present a formalisation in the Lean theorem prover of a strong version of this lemma in which each part of the union must be approximately the same size. This stronger version has not been formalised previously in any theorem prover. Our proof closely follows the pen-and-paper method, allowing our formalisation to provide an explicit upper bound on the number of parts. An application of this lemma is also formalised, namely Roth's theorem on arithmetic progressions in qualitative form via the triangle removal lemma.

2012 ACM Subject Classification Theory of computation → Interactive proof systems; Mathematics of computing → Extremal graph theory

Keywords and phrases Lean, formalisation, formal proof, graph theory, combinatorics, additive combinatorics, Szemerédi's Regularity Lemma, Roth's Theorem

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.9

Supplementary Material A snapshot of the development branch containing all the formalisation relevant here is available at:
Software (Source Code): `https://github.com/b-mehta/regularity-lemma`
archived at `swh:1:dir:e1da55f4222dac91b940ca052928eaace09762da`

Funding *Bhavik Mehta*: Cantab Capital Institute for the Mathematics of Information

Acknowledgements We want to thank Anne Baanen, Thomas Bloom, Kevin Buzzard, Timothy Gowers, Heather Macbeth, the mathlib maintainers and the anonymous reviewers for useful comments on previous versions of this paper. We are grateful to the mathlib community, whose work has been invaluable for the prerequisites of this project.

1 Introduction

Extremal Combinatorics is a modern and rapidly developing area of discrete mathematics [2], with problems that are often motivated by questions in other areas, such as Geometry, Number Theory, Theoretical Computer Science and Game Theory. It studies the maximal (or minimal) size of a collection of objects (such as natural numbers, edges in a graph or subsets of a finite set), subject to particular restrictions. An especially simple question in this area is Mantel's theorem [21]: What is the maximum number of edges in a graph G on n vertices which does not contain a triangle (i.e. three mutually adjacent vertices a, b, c)? Another well-studied problem is given by Roth's theorem on arithmetic progressions, which asks for the maximum size of a subset $A \subseteq \{1, \dots, n\}$ which does not contain three distinct integers a, b, c such that $a + c = 2b$.

While Mantel's theorem was resolved in 1907 – the maximum number of edges is given by $\lfloor x^2/4 \rfloor$ – we do not have precise bounds for Roth's theorem, but rather asymptotic lower and upper bounds with a nontrivial gap between them [16, 3]. Nonetheless, some of the earliest work on this problem showed that the maximum size must be smaller than any linear function of n – in fact in 1953 Roth proved it is bounded above by a constant multiple of

$\frac{n}{\log\log n}$ [27]. One proof of Roth's theorem uses Szemerédi's regularity lemma [29], one of the most powerful tools of extremal graph theory [20]. A simplified statement of this lemma says that every graph can be partitioned into parts that behave similarly in many ways to random graphs, which are often easier to prove results about than arbitrary graphs.

In this paper we describe our formal proof in the Lean theorem prover [23] of a strong form of the regularity lemma, and provide the application of the lemma to Roth's theorem. We use results from the Lean mathematical library mathlib [22], which is characterised by a distributed and decentralised community of contributors, and ubiquitous classical reasoning. Some of the more general results formalised have already been accepted into mathlib, and the remainder are in the process of being merged. The up-to-date development branch is publicly available on GitHub[1], but we also provide a snapshot of our code online[2].

A simultaneous and independent formalisation of a version of Szemerédi's regularity lemma and Roth's theorem in Isabelle/HOL has been completed by Chelsea Edmonds, Angeliki Koutsoukou-Argyraki and Lawrence Paulson, also at the University of Cambridge [10, 9]. While the statements of Roth's theorem which have been formalised are equivalent, the version of the regularity lemma which we have formalised produces a partition with stronger properties: namely the parts of the partition are all almost exactly the same size. This extra property comes at the cost of a larger bound, but both bounds are so large that this is not mathematically significant.

Our primary source for this formalisation was lectures given by Andrew Thomason on Extremal Graph Theory at the University of Cambridge in 2019, especially for the proof of the regularity lemma which we were able to follow with only minor changes. We also used notes by Yufei Zhao [32], particularly for the triangle removal lemma and corners theorem.

Throughout the paper we provide code blocks of Lean from our formalisation, with some adjustments for the sake of readability. In particular, proofs are omitted using `sorry` when only the type is relevant, implicit arguments are omitted when the types are clear from the context, and occasionally declarations are given shorter names.

In Section 2 we go into the mathematical details of Szemerédi's regularity lemma, the triangle removal lemma and Roth's theorem, giving mathematical statements of each and indicating the key proof ideas which we have formalised. Section 3 describes the formalised version of the regularity lemma and we give its statement in Lean and explain some of the details of the formalisation. Similarly, Section 4 discusses the triangle counting and triangle removal lemmas, the latter of which is one of the primary applications of the regularity lemma, while Section 5 describes the application away from graph theory to the corners theorem and Roth's theorem. In Section 6, we go into more detail about how our development compares with the Isabelle formalisation, discuss related and future work, and summarise our findings.

2 Mathematical details

Let us now give the results which are vital to this paper. We assume familiarity with basic graph theory; a detailed account of this field may be found in [4]. We take all graphs to be finite and simple, we write $V(G)$ for the vertex set of the graph G, and we write $|G|$ as an abbreviation for $|V(G)|$, the number of vertices in the graph.

[1] `https://github.com/leanprover-community/mathlib/tree/szemeredi/src/combinatorics/szemeredi`
[2] `https://github.com/b-mehta/regularity-lemma`

An *equipartition* of a finite set V of n elements is a partition $V_1, \dots, V_k$ such that for each i, $\lfloor n/k \rfloor \le |V_i| \le \lceil n/k \rceil$. Equivalently, for each i, j we have $||V_i| - |V_j|| \le 1$. We also refer to such partitions as *equitable*.

We now take vertex sets $X, Y \subseteq V(G)$. The *edge count* $e_G(X, Y)$ of the pair (X, Y) is the number of edges (in G) between vertices in X and vertices in Y, and the *density* of the pair is given by

$$d_G(X, Y) = \frac{e_G(X, Y)}{|X||Y|}.$$

Note that $0 \le d_G(X, Y) \le 1$. We will often omit the subscript G in both of these if the context is clear.

For a real $\varepsilon > 0$, the pair of vertex sets $X, Y \subseteq V(G)$ is said to be *ε-uniform* if for every $A \subseteq X$ and $B \subseteq Y$ with $|A| \ge \varepsilon|X|$ and $|B| \ge \varepsilon|Y|$ we have

$$|d(A, B) - d(X, Y)| < \varepsilon.$$

The partition $V_1, \dots, V_k$ is then said to be *ε-uniform* (also called *ε-regular*) if no more than $\varepsilon\binom{k}{2}$ of the pairs (V_i, V_j) for $i < j$ fail to be ε-uniform. In practice, we will typically have ε very small (certainly $\ll 1$), and n very large.

Intuitively, a pair of vertex sets is uniform if taking subsets does not affect the edge density significantly, provided we do not take a subset which is too small. This is a property satisfied by taking the obvious bipartition on a random bipartite graph, for instance, and it can be seen as a quasirandom or pseudo-random property [30].

2.1 Proof idea of the regularity lemma

▶ **Theorem 1** (Szemerédi [29]). *Let $\varepsilon > 0$, and let l be a natural number. Then there exists an integer L such that every graph G with $|G| \ge l$ has an ε-uniform equipartition into m parts for some m such that $l \le m \le L$.*

Most importantly, note the choice of L depends only on ε and l and must not depend on our choice of graph G and in particular not on its size. In fact, our proof gives an explicit value of L as a function of ε and l given as an exponential tower of height $\approx 4\varepsilon^{-5}$ with l at the top of the tower, which we have also formalised, and it is known that any upper bounds on the regularity lemma must be of tower type [13].

In addition, note that the condition that $m \ge l$ is included in the statement in order to ensure that the number of parts is not too small. This condition is itself helpful to prevent the partition being too trivial: for instance the partition into exactly one part is always equitable and uniform.

The proof of the lemma uses a quantity often called the *energy* or *index* of a partition $P = \{V_1, \dots, V_k\}$ (with respect to the graph G) given by

$$q_G(P) = \frac{1}{k^2} \sum_{1 \le i < j \le k} d_G(V_i, V_j)^2.$$

Observe that the energy is nonnegative, and no larger than $\frac{1}{2}$.

The idea of the proof is not too difficult to describe now. We begin with a trivial equipartition P_0 into exactly l parts, taking the parts arbitrarily so long as the partition is equitable. At stage P_i, either the partition is ε-uniform and we stop, or we will construct a refined equipartition P_{i+1} with not many more parts and such that the energy of P_{i+1} is at

least $\varepsilon^5/8$ more than the energy of P_i. This process must terminate, as the energy is bounded above by a constant, at which point the final partition has precisely the desired properties. In particular, the number of steps taken is no more than $4/\varepsilon^5$, and hence provided that we have a bound on the number of parts of P_{i+1} in terms of P_i, we can deduce a bound on the number of parts in the final partition, which is independent of G.

All that remains to complete the proof sketch is to describe the refinement process. We do not give a detailed description of this process as it is fairly technical, and more detailed accounts can be found in [4], but instead give a flavour of the ideas used in the process. Given the equipartition P_i which we know not to be ε-uniform with respect to G we will give a new partition Q, which is a refinement of P_i, and has the desired energy increment (in fact, we will have an increment of $\varepsilon^5/2$, better than we had hoped for). To do this, for the non-uniform pairs we pick subsets that witness the non-uniformity, and then pick Q such that these subsets are all unions of parts of Q. However, this partition Q may not be equitable, so we must do an additional "shuffling" step to produce P_{i+1} by producing an equipartition which closely approximates Q in the appropriate sense. Of course, our P_{i+1} may no longer have the required energy increment, but by very careful choice of the shuffling operation we may show that the energy of P_{i+1} is close to the energy of Q, recovering the desired energy increment of $\varepsilon^5/8$.

2.2 Triangle removal

A well-known application of the regularity lemma is the triangle removal lemma. Recall that a *triangle* in a graph is given by three distinct vertices which are adjacent, and a graph is *triangle-free* if it has no triangles.

▶ **Theorem 2** (Triangle removal theorem[32]). *Let $0 < \varepsilon \leq 1$. Then there exists a $\delta > 0$ such that for any graph G with n vertices with fewer than δn^3 triangles, we can remove no more than εn^2 edges in such a way that the resulting graph will be triangle-free.*

Intuitively, this says that if the proportion of triangles is small, then to remove all of them we only need to remove a small proportion of the edges (as the number of "possible" triangles in an arbitrary graph is on the order of n^3, and the number of possible edges is on the order of n^2). We now sketch the proof of this theorem using the regularity lemma to illustrate the sorts of ideas which need to be formalised. As before, we do not give all the details of the proof here, but they can be found in [32].

To prove this lemma, we suppose we have such a graph G, assuming for the sake of contradiction that there is no collection of fewer than εn^2 edges which can be removed to make the graph triangle-free. As is common in this sort of combinatorial proof, we will choose δ later. We first use the regularity lemma to construct a partition which is $\varepsilon/8$-uniform and has at least $l = 4/\varepsilon$ parts. We then construct the "reduced graph" G' of G by removing all edges which are internal to parts of the partition, removing edges between non-uniform pairs, and removing edges between pairs with edge density less than $\varepsilon/4$. By the properties of the partition constructed by the regularity lemma, it is not hard to show that this step does not remove more than εn^2 edges: the first set of edge removals is internal to a part, and we know that each part has size at most $\lceil n/l \rceil \approx \varepsilon n/4$; the second set operates only on non-uniform pairs of which there must be few; and the third set of removals by definition removes edges between pairs of small density so cannot remove many edges. From our contradiction hypothesis, the reduced graph must have a triangle, and by construction the triangle has each of its vertices in different parts of the partition.

We then appeal to the triangle counting lemma.

▶ **Lemma 3** (Triangle counting lemma). *Let $0 \le \varepsilon \le \frac{1}{2}$. Take a tripartite graph G with vertex sets X, Y, Z which are pairwise ε-uniform, and suppose the pairwise edge densities between them are not below 2ε. Then there must be at least $(1-2\varepsilon)\varepsilon^3|X||Y||Z|$ triangles in G.*

This lemma can be proved by a fairly straightforward counting argument, and then applied to our situation (with the obvious change in ε) to deduce that there must be a cubic number of triangles, as the three parts that our given triangle were in satisfy the hypotheses of the triangle counting lemma, thus concluding the proof.

2.3 Roth's theorem

For convenience, we restate Roth's theorem in an qualitative form.

▶ **Theorem 4** (Roth [27]). *For every $\delta > 0$, there exists an n_0 such that for any $n \ge n_0$ and any subset $A \subseteq \{1, \dots, n\}$ satisfying $|A| \ge \delta n$, there are distinct elements $a, b, c \in A$ such that $a + c = 2b$.*

Note that an equivalent conclusion is that there are a, d with $d \neq 0$ and $a, a+d, a+2d \in A$ – both of these conditions ask for three equally spaced elements of A, i.e. an arithmetic progression of length 3.

While it is possible to prove Roth's theorem directly from the triangle removal lemma [32], an alternative approach is to first prove the corners theorem [28]. The corners theorem itself is a compelling result and can be viewed as a generalisation of Roth's theorem, so to illustrate the power of the results we have formalised, we take this alternative path. For a subset $B \subseteq \{1, \dots, n\}^2$, a *corner* is a triple of points of the form $(x, y),\ (x+h, y),\ (x, y+h) \in B$ with $h > 0$.

▶ **Theorem 5** (Corners theorem [1]). *For every $\delta > 0$, there exists an n_0 such that for any $n \ge n_0$ and any subset $B \subseteq \{1, \dots, n\}$ satisfying $|B| \ge \delta n^2$, there is a corner in B.*

As previously, we sketch the flavour of the proof here rather than giving all details. Let us define an *anticorner* to be a triple of points of the form $(x, y), (x+h, y), (x, y+h) \in B$ with $h < 0$. To prove the corners theorem, it is convenient to first show the existence of *either* a corner or an anticorner in a sufficiently dense set, and then deduce the corners theorem by a straightforward mirroring argument. If $B \subseteq \{1, \dots, n\}$ has no corners, by the pigeonhole principle there exists a point $z \in \{1, \dots, 2n\}^2$ such that $B \cap (z - B)$ has size $\ge |B|^2/(2n)^2$, and this intersection contains no corners or anticorners.

Thus it suffices to show that our subset $B \subseteq \{1, \dots, n\}$ has either a corner or an anticorner – we will call this the weak corners theorem. To this end, we construct a tripartite graph X, Y, Z such that triangles are in bijective correspondence with corners of B, anticorners of B or "trivial" corners, namely those with $h = 0$. By the size condition on B, there must be at least δn^2 of these trivial corners and hence at least δn^2 triangles. Furthermore, these triangles are all edge-disjoint, so to remove all triangles we must remove at least δn^2 edges of the graph, and now the triangle removal lemma shows that we have εn^3 triangles, so we have at least $\varepsilon n^3 - \delta n^2$ non-trivial triangles. Hence if $\varepsilon n > \delta$ there is a non-trivial corner or anti-corner, as required (and we choose n_0 such that $n_0 > \varepsilon/\delta$).

Finally, from the corners theorem we can prove Roth's theorem: Given our set $A \subseteq \{1, \dots, n\}$ construct the set $B = \{(x, y) : x - y \in A\} \subseteq \{1, \dots, 2n\}^2$. It is easy to see that corners in the set B produce 3-term arithmetic progressions in A, and we can directly calculate that $|B| \ge n|A|$ to finish the proof.

In the following sections, we describe the Lean formalisation of these concepts and proofs.

3 Szemerédi's regularity lemma

We begin by describing the basic data types used for the formalisation of the regularity lemma, focusing on partitions; then discuss a technicality regarding choice of witnesses. Next we discuss in detail how the induction construction works, and the calculations involved in showing it has the required properties, and conclude this section by showing how these parts fit together to prove the main theorem.

As mentioned previously, we will omit some proofs by using the Lean keyword `sorry`, some arguments are omitted when the types are clear from the context, and occasionally declarations are given shorter names.

3.1 Partitions

The notion in mathlib of `finset` provides a type of finite sets which can used for finitary operations, such as a sum or a finite union, and considered as (coerced to) sets using the syntax `(s : set α)`, a *type ascription*. There already exist partition-like concepts in mathlib, namely `setoid` and `indexed_partition`, but neither of those satisfactorily support finitely many parts. As the regularity lemma is inherently finite, we settled on defining a new kind of partition.

A finite partition of s is a finite set of pairwise disjoint finite sets (called parts) whose (finite) union is s. For technical reasons and following usual mathematical convention, we forbid the trivial part. A partition is equitable if all its parts have almost the same size, in our case written by saying the function `card` is equitable on the set of parts.

Our parts in a partition will not be indexed by $\{1, \dots, k\}$, but rather simply considered as a collection. This is a design decision which we found most convenient to work with, for instance so we do not worry about indexing when constructing new partitions from old.

```
structure finpartition (s : finset α) :=
(parts : finset (finset α))
(disjoint : parts.pairwise_disjoint)
(sup_parts : parts.sup id = s)
(not_bot_mem : ⊥ ∉ parts)

def set.equitable_on (s : set α) (f : α → ℕ) : Prop :=
∀ a₁ a₂, a₁ ∈ s → a₂ ∈ s → f a₁ ≤ f a₂ + 1

def finpartition.is_equipartition
  {s : finset α} (P : finpartition s) : Prop :=
(P.parts : set (finset α)).equitable_on card
```

It is useful at this point to briefly discuss Lean's "dot notation". Given a term `x` of type `my_type` say, along with a definition `my_type.func` taking `x` as an explicit argument, we may abbreviate `my_type.func x` as `x.func` for convenience's sake. In this case, we will write `P.is_equipartition` because `P` is a `finpartition`.

We now define the energy as a real number. To simplify later algebraic manipulations, the expression we use will in fact be double the mathematical definition given previously, we sum over $U \neq V$ thanks to `finset.off_diag`: given a finite set, this will construct the finite set of ordered pairs which are unequal. Using the indexed partition notation, this corresponds more closely to $\sum_{1 \leq i \neq j \leq k}$.

```
def finset.off_diag (s : finset α) : finset (α × α) := sorry
```

As a consequence, note that each refinement will increase the energy by $\frac{\varepsilon^5}{4}$, not $\frac{\varepsilon^5}{8}$, and the energy is bounded above by 1, rather than $\frac{1}{2}$.

```
def finpartition.energy (P : finpartition s) (G : simple_graph α) : ℝ :=
(∑ UV in P.parts.off_diag, (G.edge_density UV.1 UV.2)^2) / P.parts.card^2
```

3.2 Witnesses

When the pair (U, V) is non-uniform, the proof reads "Pick $U' \subseteq U, V' \subseteq V$ witnesses of non-uniformity". If one does that naïvely, extracting (U', V') from the existential in (U, V), the witnesses we get from (U, V) and (V, U) will not in general match. Instead of getting a pair of witnesses we get two halves of one, which will not be good enough to prove the desired properties about the choices. So we must pick the witnesses in such a way as to make the witnesses picked from both sides agree.

Our first try was using `sym2`: `sym2 X` is the type of unordered pairs of `X`. This seems promising to avoid our problem with ordered pairs, as we can make the ordered witnesses for the ordered pair (U, V) out of unordered witnesses for the unordered pair (U, V). But now we must match the witnesses to the original pairs which, while possible to implement, is awkward to work with in practice.

In the end, we went for a much simpler solution: Put an arbitrary order on the parts and only take the witnesses of (U, V) if $U < V$; otherwise take the witnesses of (V, U) and swap them.

The lemma `not_witness_prop` simply converts a proof that the pair (U, V) is not uniform to an existential statement asserting that witnesses to non-uniformity exist, so we can use the axiom of choice to pick them.

```
def witness_aux (ε : ℝ) (U V : finset α) : finset α × finset α :=
if h : ¬G.is_uniform ε U V
  then ((not_witness_prop h).some, (not_witness_prop h).some_spec.2.some)
  else (U, V)

def witness (ε : ℝ) (U V : finset α) : finset α :=
if well_ordering_rel U V
  then (G.witness_aux ε U V).1
  else (G.witness_aux ε V U).2
```

From mathlib we have `well_ordering_rel` which constructs a well ordering on any given type using the axiom of choice, and `well_ordering_rel U V` is then a predicate for whether $U < V$ in this ordering.

3.3 Constructing the new partition

The induction step is rather delicate, we need to construct the new partition Q from the existing partition P very carefully to have all of the appropriate requirements: ensuring it is equitable is the most difficult.

We proceed in five steps, where the first four will operate inside a fixed part U of P, and the fifth combines all of these to make the final partition Q.

1. First, we collect the witnesses of non-uniformity with respect to U. For each part V of P such that (U, V) is not ε-uniform, we take the corresponding witness.

```
def finpartition.witnesses (P : finpartition (univ : finset α))
  (G : simple_graph α) (ε : ℝ) (U : finset α) : finset (finset α) :=
(P.parts.filter (λ V, U ≠ V ∧ ¬G.is_uniform ε U V)).image (G.witness ε
    U)
```

2. Second, from a set of sets F, we can construct the partition of a set s such that its points lie in the same part if and only they are in the same sets of F. In particular, this means that every set in F will be a union of parts from the partition. We construct the partition by going through every $E \subseteq F$ (written as `F.powerset`, and constructing the set `s.filter (λ a, ∀ t ∈ F, t ∈ E ↔ a ∈ t)` of the elements of `s` which are in every element of E and no element of $F \setminus E$, to produce a part. We must also remove $\emptyset$, which is done by the `of_erase` constructor, and we elide the proofs that this is indeed a partition.

```
def atomise (s : finset α) (F : finset (finset α)) : finpartition s :=
of_erase
    (F.powerset.image (λ E, s.filter (λ a, ∀ t ∈ F, t ∈ E ↔ a ∈ t)))
    sorry
```

3. For a partition P of a set s of size $am + b(m+1)$, find a partition Q of s into a parts of size m, b parts of size $m+1$ such that each part of P is the union of the parts of Q it contains along with at most m "spare" vertices.

```
def equitabilise (s : finset α) (P : finpartition s)
  (m a b : ℕ) (h : a * m + b * (m + 1) = s.card) :
  finpartition s := sorry
```

   The construction is fairly easy in informal mathematics, but not immediate to formalise: an additional complication was that in our reference the precise condition on Q was not explicitly given, and the size condition "h" was not spelled out particularly clearly – although both of these were implicit in the argument and did not require any change.
   The construction we used was also not described in our reference as it is intuitive – particularly at this level of mathematics – but we needed to give the construction ourselves and formalise it. The idea is to repeatedly remove those sets of size m or $m+1$ from s which are entirely contained in one part of P, as these can immediately form a part of Q. This terminates (as the size of the remainder strictly decreases), and from what is left we may form parts of Q arbitrarily: the size condition on s ensures that we have a good number of vertices left, enough to form entire parts.
   To formalise this idea, we use strong induction on the (size of the) finite set `s`, removing a set of size m if $a > 0$, and a set of size $m+1$ otherwise, and hence construct the desired partition and give its required properties.
4. Putting all this together, we get a refinement of each part U of the partition. The refinement is obtained by taking the witnesses of non-uniformity, atomising using these, and equitabilising the resulting partition. Depending on whether U was a "big" or a "small" part, the precise parameters for the equitabilise step change, which is the content of the outer `if ... then ... else ...`.

```
def finpartition.chunk_increment (P : finpartition (univ : finset α))
  (hP : P.is_equipartition) (G : simple_graph α) (ε : ℝ)
  (U : finset α) (hU : U ∈ P.parts) :
  finpartition U :=
if U.card = card α / P.parts.card
```

```
  then (atomise U (P.witnesses G ε U)).equitabilise m
         (4^P.parts.card - a) a sorry
  else (atomise U (P.witnesses G ε U)).equitabilise m
         (4^P.parts.card - a - 1) (a + 1) sorry
```

Note, the `hP` and `hU` hypotheses are used in the elided `sorry` to satisfy the cardinality condition from `equitabilise`.

5. Finally, juxtaposing the refinement of each part yields the desired partition refinement. We do so using `finpartition.bind`, which takes a partition P and partitions of its parts as input, and returns the partition obtained by juxtaposing those subpartitions.

```
def finpartition.increment (P : finpartition (univ : finset α)) (hP :
    P.is_equipartition) (G : simple_graph α) (ε : R) : finpartition
    (univ : finset α) :=
P.bind (P.chunk_increment hP G ε)
```

3.4 Calculations

The goal is to prove that one refinement step increases the energy by at least $\frac{\varepsilon^5}{4}$. To do this, we break the energy into two sums: one over uniform parts and one over non-uniform ones. We bound each sum term-wise.

```
variables (P : finpartition univ) (hP : P.is_equipartition)
  (hPG : ¬P.is_uniform G ε)

lemma energy_increment_uniform
  {U V : finset α} (hU : U ∈ P.parts) (hV : V ∈ P.parts) :
  (G.edge_density U V)^2 - ε^5/25 ≤
    (∑ (a, b) in (P.chunk_increment hP G ε hU).parts ×
    (P.chunk_increment hP G ε hV).parts,
      (G.edge_density a b)^2) / 16^P.parts.card := sorry

lemma energy_increment_nonuniform
  {U V : finset α} (hU : U ∈ P.parts) (hV : V ∈ P.parts)
  (hUV : U ≠ V) (hGUV : ¬ G.is_uniform ε U V) :
  (G.edge_density U V)^2 - ε^5/25 + ε^4/3 ≤
    (∑ (a, b) in (P.chunk_increment hP G ε hU).parts ×
    (P.chunk_increment hP G ε hV).parts,
      (G.edge_density a b)^2) / 16^P.parts.card := sorry

lemma energy_increment :
  P.energy G + ε^5 / 4 ≤ (P.increment G ε).energy G := sorry
```

A great deal of the proof is spent on lower bounding the energy of the refined partition. This part was the most tedious and the least mathematically interesting, both on paper and during formalization. It accounts for roughly 700 lines of code. Such length is mostly explained by the equitability requirement: the non-equitable regularity lemma requires much less fiddly calculations. Nevertheless we can partly attribute this to our unfamiliarity with formal manipulations of complicated sums at the start of this project, and we found mathlib automation not particularly suited to our case. For instance, tactics for linear arithmetic such as `linarith` are not currently useful to deal with sums, while rewrite lemmas as commonly used by the simplifier are difficult to use to show chains of inequalities by transitivity.

3.5 Induction

Our end theorem is *effective*, in the sense that we give an explicit bound for n_0, rather than merely stating its existence. To be able to take a refined partition at each stage, we have a technical condition that $100 < 4^n\varepsilon^5$ and we also want the end partition to have at least l parts, so we start the induction with a dummy partition whose size is expressed by `initial_bound`. Recall that a refinement step on a partition in n parts yields a partition in $n4^n$ parts. This is expressed by `exp_bound` below. As each refinement step increases the energy by $\frac{\varepsilon^5}{4}$ and the energy stays between 0 and 1, we won't need more than $\frac{4}{\varepsilon^5}$ refinements, which makes `szemeredi_bound` our final bound.

```
def exp_bound (n : ℕ) : ℕ := n * 4^n

def initial_bound (ε : ℝ) (l : ℕ) : ℕ :=
max l (⌊log (100/ε^5) / log 4⌋₊ + 1)

def szemeredi_bound (ε : ℝ) (l : ℕ) : ℕ :=
let L := exp_bound^[⌊4 / ε^5⌋₊] (iteration_bound ε l) in L * 16^L
```

Here, `⌊x⌋₊` is notation for the floor function. and `f^[n]` is notation for function iteration, namely the composition $\underbrace{f \circ \cdots \circ f}_{n \text{ times}}$.

We are now in a position to state and prove the regularity lemma.

```
theorem szemeredi_regularity [fintype α] (G : simple_graph α)
  (ε : ℝ) (hε : 0 < ε) (l : ℕ) (hl : l ≤ card α) :
  ∃ (P : finpartition univ),
    P.is_equipartition ∧ l ≤ P.parts.card ∧ P.parts.card ≤
    szemeredi_bound ε l ∧ P.is_uniform G ε :=
```

This is then mathematically proved by iteration on energy. To lead the process, we formally use an induction over the naturals, where our specific goal hypothesis is displayed here:

```
∀ i : ℕ, ∃ (P : finpartition univ), P.is_equipartition ∧
  t ≤ P.parts.card ∧ P.parts.card ≤ exp_bound^[i] t ∧
    (P.is_uniform G ε ∨ ε^5 / 4 * i ≤ P.energy G)
```

The $i = 0$ case can be proved by taking an arbitrarily chosen equipartition with t parts which clearly satisfies all the requirements, while the inductive case will be satisfied by using `finpartition.increment`, together with `energy_increment` or using the original partition. Then, applying this with `i` chosen to be `⌊4 / ε^5⌋₊ + 1`, the right hand side of the ∨ must fail as the energy is bounded above by 1, so we must have a uniform partition.

4 Triangle counting and triangle removal

In this section we describe the formalisation of the triangle counting and triangle removal lemmas.

We begin by defining n-cliques in a graph, an obvious generalisation of triangles, by giving a predicate describing when a given finite set of vertices forms an n-clique: there are n of them and they are pairwise adjacent. It is then easy to define a predicate for a graph having no triangles: any finite set of vertices cannot form a 3-clique.

```
def simple_graph.is_n_clique (G : simple_graph α) (n : ℕ) (s : finset α) :
  Prop :=
s.card = n ∧ (s : set α).pairwise G.adj

def simple_graph.no_triangles (G : simple_graph α) : Prop :=
∀ t, ¬ G.is_n_clique 3 t
```

We use the coercion of `finset` to `set` as mentioned previously, as the `pairwise` predicate is defined for arbitrary sets. While this generic definition is not directly useful to this application, it will no doubt be useful to other graph theorists working with mathlib: for instance to state Ramsey's theorem or Turán's theorem [31]. Recall also that we may write `G.is_n_clique 3 t` since `G` is a simple graph using Lean's dot notation.

The (finite) set of triangles in a graph can then be defined, and we can prove the key property of this set, that a triple `{x, y, z}` is in the set of triangles if and only if the three obvious adjacency relations hold.

```
def simple_graph.triangle_finset (G : simple_graph α) :
  finset (finset α) := sorry

lemma mem_triangle_finset (x y z : α) :
  {x, y, z} ∈ G.triangle_finset ↔ G.adj x y ∧ G.adj x z ∧ G.adj y z :=
sorry
```

Following the logical progression, we now state and prove the triangle counting lemma. While the mathematical statement is usually given in terms of a tripartite graph, in applications (including ours) one often wants to take the three vertex sets as disjoint subsets of a single graph, so an alternative statement is helpful as well.

```
lemma triangle_counting
  {X Y Z : finset α} {ε : ℝ} (hε₀ : 0 < ε) (hε₁ : ε ≤ 1)
  (dXY : 2 * ε ≤ G.edge_density X Y) (uXY : G.is_uniform ε X Y)
  (dXZ : 2 * ε ≤ G.edge_density X Z) (uXZ : G.is_uniform ε X Z)
  (dYZ : 2 * ε ≤ G.edge_density Y Z) (uYZ : G.is_uniform ε Y Z) :
  (1 - 2 * ε) * ε^3 * X.card * Y.card * Z.card ≤
    ((X.product (Y.product Z)).filter $
    λ (x, y, z),
        G.adj x y ∧ G.adj x z ∧ G.adj y z).card :=
sorry

lemma triangle_counting2 {X Y Z : finset α} {ε : ℝ}
  (dXY : 2 * ε ≤ G.edge_density X Y) (uXY : G.is_uniform ε X Y)
  (hXY : disjoint X Y)
  (dXZ : 2 * ε ≤ G.edge_density X Z) (uXZ : G.is_uniform ε X Z)
  (hXZ : disjoint X Z)
  (dYZ : 2 * ε ≤ G.edge_density Y Z) (uYZ : G.is_uniform ε Y Z)
  (hYZ : disjoint Y Z) :
  (1 - 2 * ε) * ε^3 * X.card * Y.card * Z.card ≤ G.triangle_finset.card :=
sorry
```

In the statement of `triangle_counting`, we have explicitly worked with the edge relation of `G`, as opposed to re-using the previous definition of triangles: this is simply to emphasise the point that the "triangles" considered in this lemma are in fact *ordered* triples, rather than merely finite sets of size 3. In contrast, in `triangle_counting2`, we do use `G.triangle_finset.card` as we are now counting triangles of the graph `G`.

Next we must define the *reduced graph* as discussed in Section 2.

```
def reduced_graph (ε : ℝ) (P : finpartition (univ : finset α)) :
    simple_graph α :=
{ adj := λ x y, G.adj x y ∧
    ∃ U V ∈ P.parts, x ∈ U ∧ y ∈ V ∧ U ≠ V ∧
        G.is_uniform (ε/8) U V ∧
        ε/4 ≤ G.edge_density U V,
  symm := sorry,
  loopless := sorry }

lemma reduced_graph_le {ε : ℝ} {P : finpartition (univ : finset α)} :
  reduced_graph G ε P ≤ G :=
λ x y ⟨h, _⟩, h
```

To define a simple graph we give the adjacency relation, and prove it is symmetric and loopless: these proofs are omitted as they are completely routine. To express the adjacency relation, we say that `x` and `y` are adjacent in the reduced graph exactly when they are adjacent in the original graph, and that there exist parts `U, V` of the partition `P` containing `x, y` respectively, which are distinct, uniform and have an `edge_density` which is at least $\varepsilon/4$. This captures precisely the definition as given on paper, as the chosen parts must be unique by definition of a partition, and the distinctness property ensures that we are removing any edge both of whose endpoints are in a single part. We may then easily prove that the reduced graph is a subgraph of the original graph, given as a $\leq$ relation.

The other key property of the reduced graph we need is that the number of removed edges is less than εn^2.

```
lemma reduced_edges_card [nonempty α]
  {ε : ℝ} {P : finpartition (univ : finset α)}
  (hε : 0 < ε) (hP : P.is_equipartition) (hPε : P.is_uniform G (ε/8))
  (hP' : 4 / ε ≤ P.parts.card) :
  (G.edge_finset.card - (reduced_graph G ε P).edge_finset.card : ℝ) <
    ε * (card α)^2 :=
```

As is to be expected, `G.edge_finset` refers to the finite set of edges of the graph `G`. The explicit type ascription on the left of the inequality is needed for Lean to infer that we are discussing an inequality of reals, otherwise it will assume that we are stating an inequality of naturals as the left-hand-side is a natural number.

At this point we are able to prove the triangle removal lemma, but we instead pause to give a reformulation which is more convenient to prove (both mathematically and formally): the two versions are easily seen to be equivalent, indeed they are simply contrapositives of one another.

Given $\varepsilon > 0$, we say a graph G is *ε-far from being triangle-free* if one must delete at least $\varepsilon|G|^2$ edges of G to remove all triangles.

▶ **Theorem 6** (triangle removal, restated). *Let $0 < \varepsilon \leq 1$. Then there exists a $\delta > 0$ such that any graph G which is ε-far from being triangle-free contains at least $\delta|G|^3$ triangles.*

To express the property of being far from triangle-free, we say that any subgraph G' of G which is triangle-free must have no more than $\varepsilon|G|^2$ edges fewer than G.

```
def triangle_free_far (G : simple_graph α) (ε : ℝ) : Prop :=
∀ (G' ≤ G), G'.no_triangles →
    ε * (card α)^2 ≤ (G.edge_finset.card - G'.edge_finset.card : ℝ)
```

While the statement alone only says that some δ must exist for the given ε, the proof in fact gives an explicit form for what δ must be. We thus find it convenient to express δ as a function of ε, but have made no attempt to optimise this bound, but simply observe it has the same asymptotic behaviour (in the big-O sense) as proofs which follow the same approach in the standard literature. However a 2010 result of Fox [12] proves (a generalisation) of the triangle removal lemma which does not use the regularity lemma, and hence is able to achieve a significant improvement on the bound on δ. Thus, the precise value of the bound given should not be taken too seriously, but we provide it for completeness.

```
def triangle_removal_bound (ε : ℝ) : ℝ :=
min (1 / (2 * ⌈4/ε⌉₊^3))
    ((1 - ε/4) * (ε/(16 * szemeredi_bound (ε/8) ⌈4/ε⌉₊))^3)
```

Finally we have all the tools available to state and prove both forms of the lemma of this section.

```
lemma triangle_removal_2 {ε : ℝ} (hε : 0 < ε) (hε₁ : ε ≤ 1)
  (hG : G.triangle_free_far ε) :
  triangle_removal_bound ε * (card α)^3 ≤ G.triangle_finset.card :=
sorry

lemma triangle_removal {ε : ℝ} (hε : 0 < ε) (hε₁ : ε ≤ 1)
  (hG : (G.triangle_finset.card : ℝ) <
           triangle_removal_bound ε * (card α)^3) :
  ∃ (G' ≤ G),
    (G.edge_finset.card - G'.edge_finset.card : ℝ) < ε * (card α)^2
      ∧ G'.no_triangles :=
sorry
```

With these theorems formalised, all that remains is to deduce the corners theorem and Roth's theorem.

5 Corners theorem and Roth's theorem

In this section we describe the formal proof of the corners theorem and Roth's theorem. As indicated in Section 2, the proof of the corners theorem works by first showing a "weak" corners theorem in which an anticorner may be constructed by means of constructing an appropriate graph and applying the triangle removal lemma.

For a finite set A of pairs of naturals, we define when a triple forms a corner or anticorner.

```
def is_corner (A : finset (ℕ × ℕ)) : ℕ → ℕ → ℕ → Prop :=
λ x y h, (x, y) ∈ A ∧ (x + h,y) ∈ A ∧ (x, y + h) ∈ A

def is_anticorner (A : finset (ℕ × ℕ)) : ℕ → ℕ → ℕ → Prop :=
λ x y h, (x, y) ∈ A ∧ (h ≤ x ∧ (x-h,y) ∈ A) ∧ (h ≤ y ∧ (x, y-h) ∈ A)
```

It is important to note here that the formalised definition is different from the mathematical one given previously. In particular, we allow `h` to be zero in both cases, and so in the formal statements of the theorems we must take care to disallow this case, else the theorems become trivially true. Nonetheless, it is useful to allow this case in the definition, as "trivial" corners with $h = 0$ are important to talk about in the proof in order to show the graph we will construct is far from being triangle free.

The reader may also be surprised to see the condition $h \leq x$ appearing in the definition of anticorners. This is present as Lean's subtraction operation on naturals is truncated subtraction, also known as the monus operator, and gives $x - y = 0$ in the case $x \leq y$ so the mathematical objection of "A is a set of pairs of naturals, so cannot contain negatives" does not apply. Thus we must insist that $h \leq x$, otherwise the anticorner may behave unpredictably.

The auxiliary tripartite graph we construct from the set A to prove the weak corners theorem has the first vertex set as horizontal lines of $\{1, \ldots, n\}^2$, the second vertex set as vertical lines of the same set, and the third vertex set as diagonal lines. This latter vertex set will be indexed by the integers $\{1, \ldots, 2n\}$, and the diagonal line corresponding to vertex k is represented by the set of points (x, y) with $x + y = k$. Edges of the graph are then given by pairs of lines in different directions which intersect at a point of A.

We can construct this in Lean by using its built-in inductive type mechanism, which defines a new type by specifying its constructors. The definition `corners_edge` below in fact has six constructors, we will show just two of them, the others are analogous.

```
inductive corners_vertices (N : ℕ)
| horiz : fin N → corners_vertices
| vert : fin N → corners_vertices
| diag : fin (2 * N) → corners_vertices

inductive corners_edge (A : finset (ℕ × ℕ)) :
  corners_vertices N → corners_vertices N → Prop
| hv {h v : fin N} : (h, v) ∈ A →
       corners_edge (horiz h) (vert v)
| hd {h : fin N} {k : fin (2 * N)} : h ≤ k → (h, k - h) ∈ A →
       corners_edge (horiz h) (diag k)

def corners_graph (N : ℕ) (A : finset (ℕ × ℕ)) :
  simple_graph (corners_vertices N) :=
{ adj := corners_edge A, symm := sorry, loopless := sorry }
```

The type `fin N` is the type of naturals strictly less than N, and corresponds to our notion of $\{1, \ldots, n\}$. We have omitted some type ascriptions here for the sake of readability, but emphasise that all inequalities and subtractions here take place over the natural numbers, and there is of course a canonical embedding from `fin N` to the naturals. As previously, we insist on inequalities in naturals as additional hypotheses in order to avoid issues with natural subtraction.

To give an example, the part of the `corners_edge` definition labelled `hd` (horizontal-diagonal) allows construction of an edge between the vertex represented by the horizontal line `h` and the vertex represented by the diagonal line `k` exactly when `h` $\leq$ `k` and `(h, k - h)` $\in$ `A`, i.e. exactly when the two lines intersect at a point in `A`. The other five cases cover the remaining line intersections. Thus the Lean definition mimics the mathematical definition, the sole exception being zero-indexing in Lean rather than one-indexing in the mathematics.

As discussed in the proof sketch, we now show that the `corners_graph` is far from being triangle-free, in particular $\varepsilon/16$-far. It is useful first to show that if we can find t-many edge-disjoint triangles, and $e|G|^2 \leq t$, then our graph is ε-far from being triangle free, and then deduce that if $\varepsilon n_0^2 \leq |A|$, the corners graph of A is $\varepsilon/16$-far from being triangle free. The former proof can be done simply by observing that if we have removed edges and removed all the triangles, we must have removed at least one edge from every triangle (by edge-disjointness), and as we have many triangles we immediately show the number of removed edges must satisfy the required inequality.

```
lemma triangle_free_far_of_disjoint_triangles {ε : ℝ}
  (tris : finset (finset α)) (htris : tris ⊆ G.triangle_finset)
  (pd : (tris : set (finset α)).pairwise (λ x y, (x ∩ y).card ≤ 1))
  (tris_big : ε * card α ^ 2 ≤ tris.card) :
  G.triangle_free_far ε := sorry

lemma disjoint_triangles {ε : ℝ} (hA : A ⊆ (range N).product (range N))
    (hA' : ε * N^2 ≤ A.card) :
  (corners_graph N A).triangle_free_far (ε/16) := sorry
```

With this result at hand, it is now easy to show the weak corners theorem, as the triangle removal lemma allows us to conclude that there are non-trivial triangles. We show that if every corner or anticorner is trivial, then the number of triangles is bounded above by n^2, thus conclude that if `ε * n^2 ≤ A.card` with sufficiently large n we are done.

```
lemma trivial_triangles_corners_graph
  {A : finset (ℕ × ℕ)} {n : ℕ}
  (cs : ∀ (x y h : ℕ), is_corner A x y h → h = 0)
  (as : ∀ (x y h : ℕ), is_anticorner A x y h → h = 0) :
  (corners_graph n A).triangle_finset.card ≤ n^2 := sorry

lemma weak_corners_theorem {ε : ℝ} (hε : 0 < ε) :
  ∃ n₀ : ℕ, ∀ n, n₀ ≤ n →
    ∀ A ⊆ (range n).product (range n), ε * n^2 ≤ A.card →
      ∃ x y h, 0 < h ∧ (is_corner A x y h ∨ is_anticorner A x y h) :=
```

Now, the proof of the corners theorem and Roth's theorem can be done, to conclude our formal proofs.

```
lemma corners_theorem {ε : ℝ} (hε : 0 < ε) :
  ∃ n₀ : ℕ, ∀ n, n₀ ≤ n →
    ∀ A ⊆ (range n).product (range n), ε * n^2 ≤ A.card →
      ∃ x y h, 0 < h ∧ is_corner A x y h :=
sorry

lemma roth (δ : ℝ) (hδ : 0 < δ) :
  ∃ n₀ : ℕ, ∀ n, n₀ ≤ n →
    ∀ A ⊆ range n, δ * n ≤ A.card →
      ∃ a d, 0 < d ∧ a ∈ A ∧ a + d ∈ A ∧ a + 2 * d ∈ A :=
sorry
```

6 Discussion

We conclude by discussing related work; the advantages of mathlib for work like ours, suggesting future work and concluding.

6.1 Comparison with Isabelle

Edmonds, Koutsoukou-Argyraki, and Paulson independently and simultaneously formalised Szemerédi's regularity lemma and Roth's theorem in Isabelle/HOL [10, 9]. Due to the striking coincidence, this particular piece of related work is deserving of extra discussion.

While there are many similarities in our work, our approaches differ on several points:

- We formalised the equitable regularity lemma. This means that all parts in the final partition are ensured to be the same size, up to a difference of 1.
- We picked a different definition of uniformity of partitions.
- We showed the corners theorem on the way to Roth's theorem.

A particular deficiency in the current mathlib tactic system is for natural number arithmetic. For instance, one specific lemma that is needed for our formalization is

```
example (i j : ℕ) (hj : 0 < j) :
  j * (j - 1) * (i / j + 1) ^ 2 < (i + j) ^ 2 :=
sorry
```

which took 6 lines of proof, primarily working manually with elementary inequalities on real numbers, while strong automation could be able to simplify proofs such as these.

6.1.1 Equitability

There are a number of variants of the regularity lemma: the Isabelle/HOL development proves a version in which the resulting partition need not be equitable. From our "equitable" version, it is easy to deduce the non-equitable version, but not vice-versa: to the best of the authors' knowledge, the proof must be entirely re-done rather than strengthened to make this upgrade. We also note that mathematically, the extra restriction makes the induction step more difficult, both in its construction and proof: we must perform the `equitabilise` step, and work through a series of inequalities to recover the energy boost. Together, this represents more than half of the proof of the regularity lemma on paper, and empirically took the majority of person-hours to formalise. While part of this time may be attributed simply to the fact that the proof is relatively complex, we also believe it is in part due to both authors' unfamiliarity with real number calculations in Lean at the start of the project.

6.1.2 Uniformity

There are two standard definitions of uniformity of a partition:

(a) The first one says that there is at most a proportion of ε pairs of non-uniform parts.

(b) The second one says that

$$\sum_{\substack{u,v\in P\\ \neg(u,v)\ \varepsilon\text{-uniform}}} |u||v| \leq \varepsilon |P|^2.$$

We chose the former while Edmonds et al. went for the latter. When the partitions in question are all equitable (as ours are), these two definitions are equivalent, up to a factor of 2 in the parameter ε one way and 4 the other way. However when the partitions are not equitable, neither trivially implies the other: definition (a) will allow a small number of very large non-uniform pairs while definition (b) allows many very small non-uniform pairs.

6.2 Related work

The cap set problem, formalised in Lean in 2019 [7], has many mathematical similarities to Roth's theorem: they both ask about the maximum size of a subset of a particular set which avoids three-term arithmetic progressions. However, the method of proof in the two problems are very different: Roth's theorem uses graph theory and the regularity method, whereas the cap set problem uses linear algebra and the polynomial method.

Lean's graph theory library is relatively new, but has been used to show Hall's Marriage Theorem including infinitary versions [17], and the Friendship theorem[3]. Many other theorem provers have developed graph theory libraries, including PVS [5], Mizar [18], Isabelle [25], Coq [8]. As far as we are aware, there has not been any other formalisation progress in the area of extremal graph theory or extremal combinatorics.

6.3 Concluding thoughts

The choice of the regularity lemma was in part to provide another powerful tool in the Lean mathematician's toolbox, as well as to investigate how subtle induction arguments may be adapted to formalised proofs. An obvious future step is to formalise additional proofs which use the regularity lemma, such as the blow-up lemma [19] and the Erdős-Stone theorem [11], or applications outside of extremal graph theory such as bounds on the Ramsey number of graphs of bounded maximum degree [6].

More broadly, the ideas of this paper may be adapted to formalise generalisations of the results given here, such as hypergraph regularity lemmas of Gowers [14] or Nagle, Rödl, Schacht and Skokan [24, 26] in order to deduce Szemerédi's theorem on arithmetic progressions, which in turn begins the path towards formalising the celebrated Green-Tao theorem [15].

Our overall formalisation is about 3500 lines of code (including blank lines and comments), of which around 900 lines are for the triangle counting and removal theorems and 500 are for the corners theorem and Roth's theorem. We validated existing design choices used in mathlib primarily for finite sets and simple graphs and added minor lemmas which were not previously present. Parts of our work has already been accepted into mathlib, and we are in the process of merging the remainder.

We have additionally confirmed that combinatorics in Lean can often be done by following pen-and-paper proofs, or mathematical proofs in existing literature rather than needing to come up with new abstractions. Our work indicates the value of a single library of mathematics able to work with finite sets, real number inequalities and structures such as partitions and simple graphs to enable proofs using all of these in tandem.

References

1 Miklós Ajtai and Endre Szemerédi. Sets of lattice points that form no squares. *Stud. Sci. Math. Hungar*, 9(1975):9–11, 1974.

2 Noga Alon. Problems and results in extremal combinatorics–III. *Journal of Combinatorics*, 7(2):233–256, 2016.

3 Thomas F Bloom and Olof Sisask. Breaking the logarithmic barrier in Roth's theorem on arithmetic progressions. *arXiv preprint arXiv:2007.03528*, 2020.

4 Béla Bollobás. *Modern graph theory*, volume 184. Springer Science & Business Media, 1998.

5 Ricky W Butler and Jon A Sjogren. *A PVS graph theory library*, volume 206923. National Aeronautics and Space Administration, Langley Research Center, 1998.

6 C Chvatál, Vojtech Rödl, Endre Szemerédi, and W Tom Trotter Jr. The Ramsey number of a graph with bounded maximum degree. *Journal of Combinatorial Theory, Series B*, 34(3):239–243, 1983.

[3] https://github.com/leanprover-community/mathlib/blob/master/archive/100-theorems-list/83_friendship_graphs.lean

7 Sander R. Dahmen, Johannes Hölzl, and Robert Y. Lewis. Formalizing the Solution to the Cap Set Problem. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:19, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ITP.2019.15.

8 Christian Doczkal and Damien Pous. Graph theory in coq: Minors, treewidth, and isomorphisms. *Journal of Automated Reasoning*, 64(5):795–825, 2020.

9 Chelsea Edmonds, Angeliki Koutsoukou-Argyraki, and Lawrence C. Paulson. Roth's Theorem on Arithmetic Progressions. *Archive of Formal Proofs*, December 2021. , Formal proof development. URL: https://isa-afp.org/entries/Roth_Arithmetic_Progressions.html.

10 Chelsea Edmonds, Angeliki Koutsoukou-Argyraki, and Lawrence C. Paulson. Szemerédi's Regularity Lemma. *Archive of Formal Proofs*, November 2021. , Formal proof development. URL: https://isa-afp.org/entries/Szemeredi_Regularity.html.

11 Paul Erdös and Arthur H Stone. On the structure of linear graphs. *Bulletin of the American Mathematical Society*, 52(12):1087–1091, 1946.

12 Jacob Fox. A new proof of the graph removal lemma. *Annals of Mathematics*, pages 561–579, 2011.

13 William Timothy Gowers. Lower bounds of tower type for Szemerédi's uniformity lemma. *Geometric & Functional Analysis GAFA*, 7(2):322–337, 1997.

14 William Timothy Gowers. Hypergraph regularity and the multidimensional Szemerédi theorem. *Annals of Mathematics*, pages 897–946, 2007.

15 Ben Green and Terence Tao. The primes contain arbitrarily long arithmetic progressions. *Annals of mathematics*, pages 481–547, 2008.

16 Ben Green and Julia Wolf. A note on Elkin's improvement of Behrend's construction. In *Additive number theory*, pages 141–144. Springer, 2010.

17 Alena Gusakov, Bhavik Mehta, and Kyle A. Miller. Formalizing Hall's Marriage Theorem in Lean, 2021. arXiv:2101.00127.

18 Sebastian Koch. Unification of graphs and relations in mizar. *Formalized Mathematics*, 28(2):173–186, 2020.

19 János Komlós. The blow-up lemma. *Combinatorics, Probability and Computing*, 8(1-2):161–176, 1999.

20 János Komlós, Ali Shokoufandeh, Miklós Simonovits, and Endre Szemerédi. The regularity lemma and its applications in graph theory. In *Summer school on theoretical aspects of computer science*, pages 84–112. Springer, 2000.

21 W. Mantel. Problem 28 (Solution by H. Gouwentak, W. Mantel, J. Teixeira de Mattes, F. Schuh and W. A. Wythoff). *Wiskundige Opgaven*, 10:60–61, 1907.

22 The mathlib Community. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 367–381, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372885.3373824.

23 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.

24 Brendan Nagle, Vojtěch Rödl, and Mathias Schacht. The counting lemma for regular k-uniform hypergraphs. *Random Structures & Algorithms*, 28(2):113–179, 2006.

25 Lars Noschinski. A graph library for isabelle. *Mathematics in Computer Science*, 9(1):23–39, 2015.

26 Vojtěch Rödl and Jozef Skokan. Regularity lemma for k-uniform hypergraphs. *Random Structures & Algorithms*, 25(1):1–42, 2004.

27 Klaus F Roth. On certain sets of integers. *J. London Math. Soc*, 28(104-109):3, 1953.

28 József Solymosi. Note on a generalization of Roth's theorem. In *Discrete and computational geometry*, pages 825–827. Springer, 2003.

29 Endre Szemerédi. Regular partitions of graphs. Technical report, Stanford Univ Calif Dept of Computer Science, 1975.

30 Andrew Thomason. Pseudo-random graphs. In *North-Holland Mathematics Studies*, volume 144, pages 307–331. Elsevier, 1987.

31 Paul Turán. On an extremal problem in graph theory. *Mat. Fiz. Lapok*, 48:436–452, 1941.

32 Yufei Zhao. Graph Theory and Additive Combinatorics, 2019. URL: `https://yufeizhao.com/gtac/gtac.pdf`.

Formalized functional analysis with semilinear maps

Frédéric Dupuis
Département d'informatique et de recherche opérationnelle, University of Montr'eal, Canada

Robert Y. Lewis
Computer Science Department, Brown University, Providence, RI, USA

Heather Macbeth
Department of Mathematics, Fordham University, New York, NY, USA

Abstract

Semilinear maps are a generalization of linear maps between vector spaces where we allow the scalar action to be twisted by a ring homomorphism such as complex conjugation. In particular, this generalization unifies the concepts of linear and conjugate-linear maps. We implement this generalization in Lean's mathlib library, along with a number of important results in functional analysis which previously were impossible to formalize properly. Specifically, we prove the Fréchet–Riesz representation theorem and the spectral theorem for compact self-adjoint operators generically over real and complex Hilbert spaces. We also show that semilinear maps have applications beyond functional analysis by formalizing the one-dimensional case of a theorem of Dieudonné and Manin that classifies the isocrystals over an algebraically closed field with positive characteristic.

2012 ACM Subject Classification Mathematics of computing → Functional analysis

Keywords and phrases Functional analysis, Lean, linear algebra, semilinear, Hilbert space

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.10

Supplementary Material *InteractiveResource (Project Website)*:
`https://robertylewis.com/semilinear-paper/`

Acknowledgements We thank Johan Commelin for many conversations about isocrystals and Johannes Hölzl for comments on work in Isabelle. We thank the mathlib community and maintainer team for insightful comments and suggestions during code review.

1 Introduction

Proof assistant users have long recognized the value of abstraction. Working at high levels of generality and specializing only when needed can save significant effort in both the long and short term. In program verification, this principle manifests in the use of stepwise refinement of programs from abstract specifications to executable code [20, 30]. Mathematical generalizations that are rarely used in informal presentations are much more common in formal libraries, including the use of filters to generalize limits in topology and analysis [18] and uniform spaces as a generalization of metric spaces [2, 8, 10].

We propose another such mathematical generalization: *linear maps*, a fundamental concept in many fields of mathematics, can be seen as a special case of *semilinear maps*. A linear algebra library built on top of this more general structure can unify concepts that would otherwise be defined separately. In particular, linear and *conjugate-linear* (or *antilinear*) maps are both examples of semilinear maps. By relating these, one can avoid a large amount of code duplication and state many theorems more naturally. This generalization is rarely seen explicitly in informal mathematics. Texts tend to focus on the linear case, claiming results about the conjugate-linear or semilinear cases "by analogy" when needed.

Motivated by the desire to formalize theorems from functional analysis at the proper level of abstraction, we have implemented this generalization in mathlib [25], a library of formal mathematics in the Lean proof assistant [13]. When we started this project, much of mathlib was already built on top of standard linear maps. With care and clever notation we were able to make the transition largely invisible. With the generalization complete we were able to state and prove a number of theorems far more elegantly than could have been done before.

Among the results unlocked by this refactor are the Fréchet–Riesz representation theorem, which states that a Hilbert space is either isomorphic or conjugate-isomorphic to its dual space; the generic definition of the adjoint operator on an inner product space over $\mathbb{R}$ or $\mathbb{C}$; and the spectral theorem for compact self-adjoint operators on Hilbert spaces, which gives a canonical form for an important class of linear maps by reference to their eigenvectors. This material in turn lays the groundwork for the formalization of vast areas of mathematics: complex Hilbert spaces are the bread and butter of quantum mechanics and are therefore a prerequisite for quantum information theory and a large part of mathematical physics.

Furthermore, as evidence that semilinear maps are useful for more than unifying real and complex vector spaces, we have also formalized the one-dimensional case of a theorem of Dieudonné and Manin [24] that classifies the isocrystals over an algebraically closed field of characteristic $p > 0$. This is a foundational result in p-adic Hodge theory.

Related literature documents the struggles in other libraries to unify real and complex linear algebra. For instance, Aransay and Divasón [5], working in Isabelle, write:

> We miss ... the definition of a "common place" or generic structure representing inner product spaces over real and complex numbers ... that could permit a definition and formalisation of the Gram-Schmidt process for both structures simultaneously.

Their work introduces a "local" solution to the issue, but we argue that basing a library on semilinear maps is the "global" solution. We discuss related work in more detail in Section 9.

We estimate that over the course of this project we have added 15k lines of code to mathlib, with 1k more lines waiting to be merged. We provide links to our contributions, indicating where they can be found in the library, on the project website.[1]

2 Mathematical preliminaries

2.1 Semilinear maps

Given modules M_1, M_2 over semirings R_1, R_2 and a ring homomorphism $\sigma : R_1 \to R_2$, a *σ-semilinear map* from M_1 to M_2 is a function $f : M_1 \to M_2$ satisfying the two axioms

1. for all $x, y \in M_1$, $f(x + y) = f(x) + f(y)$
2. for all $x \in M_1$ and $c \in R_1$, $f(cx) = \sigma(c)f(x)$.

Let us note the two canonical examples:

- For $R_1 = R_2 = R$ and σ the identity ring homomorphism $\mathrm{id}_R : R \to R$, the second condition simplifies to $f(cx) = cf(x)$, and therefore an id_R-semilinear map is precisely an R-linear map in the classic sense.
- For $R_1 = R_2 = \mathbb{C}$ and σ the complex-conjugation operation $\mathrm{conj} : \mathbb{C} \to \mathbb{C}$, the second condition simplifies to $f(cx) = \overline{c}f(x)$. Therefore a conj-semilinear map is a conjugate-linear map between complex vector spaces.

The theory of semilinear maps develops along the same lines as the theory of linear maps, with minimal adjustment. The composition of a σ-semilinear map and a τ-semilinear map, for $\sigma : R_1 \to R_2$ and $\tau : R_2 \to R_3$, is a $(\tau \circ \sigma)$-semilinear map. (For example, the composition of two conjugate-linear maps is a linear map.) If σ is bijective, the inverse of a bijective σ-semilinear map is a σ^{-1}-semilinear map.

[1] `https://robertylewis.com/semilinear-paper`

Theorems about special classes of linear maps also admit semilinear analogues. Consider, for example, the theorem that a $\mathbb{K}$-linear map $f : E_1 \to E_2$, for $\mathbb{K}$ a normed field and E_1, E_2 normed spaces over $\mathbb{K}$, is continuous if and only if it is *bounded* ($\|f(x)\| \leq M\|x\|$ for some fixed M, for all x). This theorem generalizes to σ-semilinear maps, for $\sigma : \mathbb{K}_1 \to \mathbb{K}_2$, if the ring homomorphism σ is an isometry.

2.2 Conjugate-linear maps in functional analysis

An *inner product space* is a vector space E over a scalar field $\mathbb{K} \in \{\mathbb{R}, \mathbb{C}\}$ equipped with an *inner product* $\langle \cdot, \cdot \rangle$, namely a $\mathbb{K}$-valued function of two arguments which is conjugate-linear in the first argument and linear in the second argument and which has symmetry and positivity properties:

1. for all $u, v, w \in E$, $\langle u + v, w \rangle = \langle u, w \rangle + \langle v, w \rangle$ and $\langle w, u + v \rangle = \langle w, u \rangle + \langle w, v \rangle$;
2. for all $c \in \mathbb{K}$ and $v, w \in E$, $\langle cv, w \rangle = \overline{c}\langle v, w \rangle$ and $\langle v, cw \rangle = c\langle v, w \rangle$;
3. for all $v, w \in E$, $\langle v, w \rangle = \overline{\langle w, v \rangle}$;
4. for all $v \in E$, the quantity $\langle v, v \rangle$ (which by (3) is real) is nonnegative, and strictly positive unless $v = 0$.

For the case of real scalars, $\mathbb{K} = \mathbb{R}$, we consider the conjugation operation as being the identity; this allows a development of the complex case to subsume the simpler real case.

An inner product space has an associated norm $\|v\| = \sqrt{\langle v, v \rangle}$ and hence a metric structure. A *Hilbert space* is an inner product space for which this metric is complete. This condition is automatic in finite dimension.

The *dual* of an inner product space E is the $\mathbb{K}$-vector space of continuous linear maps $\varphi : E \to \mathbb{K}$. There is a natural conjugate-linear map from E to its dual E^*: the vector $v \in E$ is mapped to the vector $\langle v, \cdot \rangle$ in E^*. To see the conjugate-linearity of this map, observe that $\langle cv, \cdot \rangle = \overline{c}\langle v, \cdot \rangle$. It is not difficult to see that, for an appropriate norm on E^*, this map is an isometry. A more subtle theorem, the **Fréchet–Riesz representation theorem**, asserts that for a Hilbert space E this conjugate-linear map is bijective.

Given Hilbert spaces E, F over $\mathbb{K}$ and a continuous linear map $T : E \to F$, it can be proved that there is a unique continuous linear map $T^* : F \to E$, the *adjoint* of T, such that for all $v \in E$ and $w \in F$, $\langle Tv, w \rangle = \langle v, T^*w \rangle$. It turns out that the operation of sending $T : E \to F$ to its adjoint $T^* : F \to E$ is a conjugate-linear map from $E \to F$ to $F \to E$. To see the conjugate-linearity in this case, observe that

$$\langle v, (cT)^* w \rangle = \langle (cT)v, w \rangle = \overline{c}\langle Tv, w \rangle = \overline{c}\langle v, T^*w \rangle = \langle v, (\overline{c}T^*)w \rangle.$$

Like the conjugate-linear map appearing in the Fréchet–Riesz representation theorem, the adjoint map $T \mapsto T^*$ turns out to be bijective and (for an appropriate norm) isometric.

Several important classes of continuous linear maps are defined using the adjoint. A continuous linear map $T : E \to E$ is *self-adjoint*, if $T^* = T$, and it is *normal*, if $T^*T = TT^*$. Self-adjoint implies normal.

The *Hilbert sum* $\bigoplus_{i \in \iota} E_\iota$ of a family of inner product spaces $(E_i)_{i \in \iota}$ is an inner product space whose elements are choices $(v_i)_{i \in \iota}$ of an element from each E_i, such that the collection of chosen elements is square-summable in the sense that $\sum_{i \in \iota} \|v_i\|^2 < \infty$. Elements in the Hilbert sum $\bigoplus_{i \in \iota} E_i$ can be added and scalar-multiplied in the obvious way. The inner product on the Hilbert sum is given by $\langle (v_i)_{i \in \iota}, (w_i)_{i \in \iota} \rangle = \sum_{i \in \iota} \langle v_i, w_i \rangle$. It can be proved that if each E_i is a Hilbert space (i.e., complete) then so is $\bigoplus_{i \in \iota} E_i$. A linear map $T : \bigoplus_{i \in \iota} E_i \to \bigoplus_{i \in \iota} E_i$ is *diagonal* if there exist scalars $(\mu_i)_{i \in \iota}$ such that for all $(v_i)_{i \in \iota} \in \bigoplus_{i \in \iota} E_i$, $T\left((v_i)_{i \in \iota}\right) = (\mu_i v_i)_{i \in \iota}$.

A linear map $T : E \to F$ between normed spaces is *compact* if the image under T of the unit ball in E is precompact (that is, has compact closure) in F. This condition implies the

continuity of T but is more restrictive. The **spectral theorem** states that a normal (over $\mathbb{C}$) or self-adjoint (over $\mathbb{R}$ or $\mathbb{C}$), compact linear map $T : E \to E$ is equivalent to a diagonal map, in the sense that there exists a bijective linear isometry Φ from E to a Hilbert sum $\bigoplus_{i\in\iota} F_i$, such that the linear map $\Phi \circ T \circ \Phi^{-1}$ is diagonal. In fact, the F_i may be chosen to be the eigenspaces of T, with the μ_i chosen to be the associated eigenvalues.

In finite dimension, every linear map is compact. In this setting the spectral theorem reduces to the more elementary **diagonalization theorem** for a normal endomorphism T of a finite-dimensional inner product space E: there exists a bijective linear isometry Φ from E to a finite direct sum of finite-dimensional inner product spaces $(F_i)_{i\in\iota}$, such that the linear map $\Phi \circ T \circ \Phi^{-1}$ is diagonal.

2.3 Frobenius-semilinear maps

Given a commutative ring R and a prime p, there is a classical construction [17] of an associated commutative ring $\mathbb{W}(R)$, the ring of *p-typical Witt vectors* of R. The elements of this ring are sequences of elements of R, but the definitions of addition and multiplication are rather elaborate. The motivating example is that for R the finite field $\mathbb{Z}/p\mathbb{Z}$, the ring $\mathbb{W}(\mathbb{Z}/p\mathbb{Z})$ is the ring of p-adic integers.

The ring $\mathbb{W}(R)$ admits a canonical ring-endomorphism, the *Frobenius endomorphism*. Concretely, when R has characteristic p, it operates by sending a sequence $(x_0, x_1, x_2, \ldots)$ to $(x_0^p, x_1^p, x_2^p, \ldots)$. In the example of the p-adic integers $\mathbb{W}(\mathbb{Z}/p\mathbb{Z})$, this endomorphism is the identity, so the construction becomes interesting only for more complicated rings R, such as field extensions of $\mathbb{Z}/p\mathbb{Z}$.

For R an integral domain of characteristic p, the ring $\mathbb{W}(R)$ is also an integral domain, and therefore has a well-defined field of fractions. In this case, the Frobenius endomorphism of $\mathbb{W}(R)$ extends to an endomorphism of its field of fractions. If moreover the ring R is perfect, then the Frobenius endomorphism is an automorphism (that is, bijective), as is the induced automorphism of its field of fractions.

Let us fix an algebraically closed field R of characteristic p (which is necessarily a perfect integral domain), and denote by K the field of fractions of $\mathbb{W}(R)$ and by $\varphi : K \to K$ the Frobenius automorphism of K. There is a very well-developed theory of φ-semilinear maps between vector spaces over K. Notably, an important **theorem of Dieudonné and Manin** [24] provides an analogue of the spectral theorem. For a finite-dimensional vector space V over K, it classifies the *isocrystals* (bijective φ-semilinear maps $f : V \to V$), by constructing for such an f a decomposition of V as a direct sum of vector spaces V_i which are preserved by f and on each of which the map f has a certain canonical form.

In the one-dimensional case this classification can be stated in a fairly elementary way. Let $f : K \to K$ be a φ-semilinear automorphism of K, considered as a vector space over itself. Then there exists an invertible element a of K and an integer $m \in \mathbb{Z}$, such that for all $v \in K$, $f(v) = p^m a^{-1} \varphi(av)$.

3 Lean preliminaries

The mathlib library builds its algebraic hierarchy using *type classes* [25, 27]. Baanen [6] gives an in depth account of mathlib's use of type classes, which we summarize very briefly.

Each argument to a Lean declaration is declared as *explicit* (`()`), *implicit* (`{}`), or *instance-implicit* (`[]`). Explicit arguments must be provided when the declaration is applied; implicit arguments are inferred by unification; instance-implicit arguments are inferred by type class instance resolution.

The fundamental type class of mathlib's linear algebra library is `module`.

```
class module (R : Type u) (M : Type v) [semiring R] [add_comm_monoid M]
  extends distrib_mul_action R M :=
(add_smul : ∀ (r s : R) (x : M), (r + s) · x = r · x + s · x)
(zero_smul : ∀ (x : M), (0 : R) · x = 0)
```

This type class says that the additive monoid `M` has an `R`-module structure: it supports scalar multiplication by elements of the semiring `R`, and this scalar multiplication behaves properly with respect to addition on `M`. When `R` is a field instead of a semiring, an `R`-module is in fact a vector space. Many definitions and theorems apply in the more general setting, and when the vector space setting is needed, the transition is invisible.

A type class is a structure (i.e. a record type) that takes zero or more *parameters* and has zero or more *fields*. In the above, the arguments `R` and `M` are parameters, as are the arguments that `R` is a semiring and `M` is an additive commutative monoid. In order to elaborate the type `module R M`, Lean's type class inference algorithm must be able to infer the latter arguments automatically. The fields of `module` are `add_smul` and `zero_smul`, and a projection to `distrib_mul_action R M`. To construct a term of type `module R M`, the user must provide these values; given a term of type `module R M`, the user can access these values. The `extends` keyword can be read as "inherits from." An assumption `distrib_mul_action R M` is available while defining the fields `add_smul` and `zero_smul`, and indeed, the scalar action used in these fields is derived from this instance.

By default the parameters to a type class are *input parameters*. Lean will begin its instance search when all input parameters are known. By denoting certain parameters as *output parameters*, the user can instruct Lean to begin searching for instances of that class before those parameters are known; they will be determined by the solution to the search. Baanen [6, Section 5.1] describes output parameters in more detail.

Like mathlib, we freely use classical logic and do not focus on defining things computably. Within code blocks in this paper, we omit the bodies of definitions and theorems when only the type is relevant, omit some implicit arguments when the types are clear from context, and occasionally rename declarations for the sake of presentation.

4 Semilinear maps in Lean

Section 2 covered the mathematical motivation for semilinear maps. Here we focus on our implementation of this generalization in Lean. This work is done in the context of mathlib [25], a project with over 860k lines of code, 240 contributors, and countless users. Given the difficulty and importance of maintaining such a large library [28], we were motivated to make this refactor with as little disruption as possible.

4.1 Defining semilinear maps

Before beginning our refactor to use semilinear maps, mathlib's linear algebra development was based on the more familiar concept of linear maps.

```
structure linear_map (R : Type u) (M₁ : Type v) (M₂ : Type w)
  [semiring R] [add_comm_monoid M₁] [add_comm_monoid M₂] [module R M₁]
  [module R M₂] extends add_hom M₁ M₂, mul_action_hom R M₁ M₂
```

Given two `R`-modules M_1 and M_2, a linear map is an additive homomorphism $M_1 \to M_2$ that respects the multiplicative action of `R`. A `mul_action_hom` is a homomorphism between types acted on by the same type of scalars [29].

For readers unfamiliar with Lean syntax, it may be clarifying to see what information goes in to defining such a linear map. Despite the intimidating syntax, the input information is exactly as expected: if you have types `R`, M_1, and M_2 with the appropriate operations and structure, you can construct a linear map by providing a function $M_1 \to M_2$ and proofs that this function factors through addition and scalar multiplication.

```
example (R : Type u) (M₁ : Type v) (M₂ : Type w)
  [semiring R] [add_comm_monoid M₁] [add_comm_monoid M₂] [module R M₁]
  [module R M₂] : linear_map R M₁ M₂ :=
{ to_fun := _, -- M₁ → M₂
  map_add' := _, -- ∀ (x y : M₁), to_fun (x + y) = to_fun x + to_fun y
  map_smul' := _ } -- ∀ (m : R) (x : M₁), to_fun (m • x) = m • to_fun x
```

As noted in Section 2, the domain and codomain of a linear map are modules over the same semiring. The same is true in the definition of linear equivalences:

```
structure linear_equiv (R : Type u) (M₁ : Type v) (M₂ : Type w)
  [semiring R] [add_comm_monoid M₁] [add_comm_monoid M₂] [module R M₁]
  [module R M₂] extends linear_map R M₁ M₂, add_equiv M₁ M₂
```

The type signature of a semilinear map[2] is more complicated, involving two scalar semirings and a ring homomorphism between them. It no longer makes sense to extend `mul_action_hom`, since the multiplicative actions are over different scalar types, so we instead add the field `map_smul` directly. The arguments `R` and `S` can be inferred from σ and are thus marked as implicit. The type `R →+* S` is the type of ring homomorphisms from `R` to `S`.

```
structure semilinear_map {R : Type*} {S : Type*} [semiring R] [semiring S]
  (σ : R →+* S) (M₁ : Type*) (M₂ : Type*)
  [add_comm_monoid M₁] [add_comm_monoid M₂] [module R M₁] [module S M₂]
  extends add_hom M₁ M₂ :=
(map_smul' : ∀ (r : R) (x : M₁), to_fun (r • x) = (σ r) • to_fun x)
```

While the type signature has grown more complicated, the constructor for a semilinear map is quite similar to that of a linear map:

```
example {R : Type*} {S : Type*} [semiring R] [semiring S]
  (σ : R →+* S) (M₁ : Type*) (M₂ : Type*)
  [add_comm_monoid M₁] [add_comm_monoid M₂] [module R M₁] [module S M₂] :
  semilinear_map σ M₁ M₂ :=
{ to_fun := _, -- M₁ → M₂
  map_add' := _, -- ∀ (x y : M₁), to_fun (x + y) = to_fun x + to_fun y
  map_smul' := _ } -- ∀ (r : R) (x : M₁), to_fun (r • x) = σ r • to_fun x
```

The generalization to semilinear equivalences is similar, but more involved in order to gracefully handle inversion of such maps. The additional parameter σ' and the `ring_hom_inv_pair` type class are explained in Section 4.3.

```
structure semilinear_equiv {R : Type*} {S : Type*} [semiring R] [semiring S]
  (σ : R →+* S) {σ' : S →+* R} [ring_hom_inv_pair σ σ']
  [ring_hom_inv_pair σ' σ] (M₁ : Type*) (M₂ : Type*)
  [add_comm_monoid M₁] [add_comm_monoid M₂] [module R M₁] [module S M₂]
  extends linear_map σ M₁ M₂, add_equiv M₁ M₂
```

[2] In our mathlib contribution we did not rename the type `linear_map` to `semilinear_map`. This simplified the refactor and makes the definition easier to find for beginners. For the sake of clarity in this paper, we refer to the generalized type by the more accurate name.

	Linear	Conjugate-linear	Semilinear	Meaning
Map	`M₁ →ₗ[R] M₂`	`M₁ →ₗ⋆[R] M₂`	`M₁ →ₛₗ[σ] M₂`	Between modules; factors over addition and scalar multiplication
Continuous map	`M₁ →L[R] M₂`	`M₁ →L⋆[R] M₂`	`M₁ →SL[σ] M₂`	Between topological modules; a continuous map
Equivalence	`M₁ ≃ₗ[R] M₂`	`M₁ ≃ₗ⋆[R] M₂`	`M₁ ≃ₛₗ[σ] M₂`	An invertible map
Isometry	`M₁ ≃ₗᵢ[R] M₂`	`M₁ ≃ₗᵢ⋆[R] M₂`	`M₁ ≃ₛₗᵢ[σ] M₂`	Between normed modules; a norm-preserving equivalence

Figure 1 Notation for various classes of (semi)linear operators that appear in this paper.

4.2 Notation for semilinear maps

One can see from these definitions that semilinear maps are not a drop-in replacement for linear maps. The type signature is different, even when looking only at explicit arguments. To convert an `R`-linear map to a semilinear map, one must know to invoke `ring_hom.id R`, the identity ring homomorphism on `R`.

Given how frequently linear maps appear in mathlib, this refactor threatened to be painful. Our job was made immensely easier by the use of notation. Before our refactor mathlib used the notation `M₁ →ₗ[R] M₂` to stand for for `linear_map R M₁ M₂`. By redefining this notation to stand for `semilinear_map (ring_hom.id R) M₁ M₂` we were largely able to avoid breaking definitions and proofs throughout the library. The same approach, with notation `M₁ ≃ₗ[R] M₂`, worked to generalize linear equivalences to semilinear equivalences. We introduced similar notation `M₁ →ₛₗ[σ] M₂` to stand for `semilinear_map σ M₁ M₂`, and `M₁ →ₗ⋆[R] M₂` to stand for a semilinear map with respect to a fixed involution such as complex conjugation.

The composition of linear maps proved to be a complication. As we note in Section 4.3, an additional type class must be inferred to justify that two semilinear maps can be composed. This inference was fragile in the presence of other features, like implicit coercions, that complicate elaboration. We introduced notation $\circ_l$ for the composition of linear maps, using `ring_hom.id` to justify the composition, and manually inserted this notation where needed.

For our new definition to be useful, theorems stated for linear maps `M₁ →ₗ[R] M₂` needed to be upgraded to theorems about semilinear maps `M₁ →ₛₗ[σ] M₂` when possible. Doing so is mostly mechanical and our use of notation let us approach this without hurry. Because theorems generalized to semilinear maps still apply directly to the linear case we were able to do this generalization incrementally from the bottom up. In particular, several more specialized classes of linear maps and equivalences are also present in mathlib (Figure 1). Our bottom-up approach allowed us to break down the refactor into more manageable pieces by generalizing these one at a time.

4.3 Composition of semilinear maps

Composition of maps is complicated by this generalization. The composition of two linear maps is straightforward: it is easy to check that the composition of the underlying functions preserves addition and scalar multiplication. With semilinear maps one must also compose the homomorphisms between scalar rings. Given `f : M₁ →ₛₗ[σ₁₂] M₂` and `g : M₂ →ₛₗ[σ₂₃] M₃`, we would naturally end up with `g.comp f : M₁ →ₛₗ[σ₂₃.comp σ₁₂] M₃`.

This ends up being awkward to handle in many common situations. Suppose we wish to state that `f : M₁ →ₛₗ[σ₁₂] M₂` and `g : M₂ →ₛₗ[σ₂₁] M₁` are inverse maps: `f.comp g = (id : M₁ →ₗ[R] M₁)`. This statement is not type-correct, since the ring homomorphism on the left is `σ₁₂.comp σ₂₁` and the one on the right is the identity. Such an issue appears in practice, for example, when defining the adjoint as a conjugate-linear map (Section 6).

To solve this issue, we introduce a type class `ring_hom_comp_triple` that states that two ring homomorphisms compose to a third.

```
class ring_hom_comp_triple [semiring R₁] [semiring R₂] [semiring R₃]
  (σ₁₂ : R₁ →+* R₂) (σ₂₃ : R₂ →+* R₃) (σ₁₃ : out_param (R₁ →+* R₃)) : Prop :=
(comp_eq : σ₂₃.comp σ₁₂ = σ₁₃)
```

We register a number of global instances of this class. We then use the `ring_hom_comp_triple` type class in the definition of composition.

```
def semilinear_map.comp {R₁ R₂ R₃ : Type*} {M₁ M₂ M₃ : Type*}
  [semiring R₁] [semiring R₂] [semiring R₃]
  [add_comm_monoid M₁] [add_comm_monoid M₂] [add_comm_monoid M₃]
  {mod_M₁ : module R₁ M₁} {mod_M₂ : module R₂ M₂} {mod_M₃ : module R₃ M₃}
  {σ₁₂ : R₁ →+* R₂} {σ₂₃ : R₂ →+* R₃} {σ₁₃ : R₁ →+* R₃}
  [ring_hom_comp_triple σ₁₂ σ₂₃ σ₁₃]
  (g : M₂ →ₛₗ[σ₂₃] M₃) (f : M₁ →ₛₗ[σ₁₂] M₂) :
  (M₁ →ₛₗ[σ₁₃] M₃)
```

While this may appear to be a rather verbose type signature for the composition of maps, it allows us to avoid the above problem without introducing further complications. In common situations, the appropriate global instances generate the necessary `ring_hom_comp_triple` argument without input from the user. For example, the following global instance allows for the composition of two (genuine) linear maps, or more generally for the composition of a semilinear map with a linear map.

```
instance [semiring R₁] [semiring R₂] {σ₁₂ : R₁ →+* R₂} :
  ring_hom_comp_triple (ring_hom.id R₁) σ₁₂ σ₁₂
```

Another instance helps in the setting of conjugate-linear maps.[3]

```
instance [comm_semiring R] [star_ring R] :
  ring_hom_comp_triple (conj R) (conj R) (ring_hom.id R)
```

We expand on the types here in Section 5.1; in concrete terms, this instance says that the conjugation operation on a type supporting conjugation is an involution. This allows us to compose two conjugate-linear maps to obtain, definitionally, a linear map. The intention is that users should never work directly with a composition `g.comp f : M₁ →ₛₗ [σ₂₃.comp σ₁₂] M₃`, but instead with `g.comp f : M₁ →ₛₗ[σ₁₃] M₃` for some σ_{13} satisfying `ring_hom_comp_triple` σ_{12} σ_{23} σ_{13}, which is strictly more general.

Similar issues appear with semilinear equivalences, specifically when defining the symmetric equivalence: if `e : E ≃ₛₗ[σ] F`, the "natural" definition of the symmetric equivalence would give `e.symm : F ≃ₛₗ[σ.symm] E`. Some ring homomorphisms, notably conjugation on $\mathbb{C}$, have the property that `σ.symm = σ`. But these equalities are rarely definitional and

[3] In fact, we do not state this instance explicitly; it is derived by type class inference from the `ring_hom_inv_pair` instance for `conj` (see below) and yet another global instance generating a `ring_hom_comp_triple` with the identity from a `ring_hom_inv_pair`.

spurious `symm`s can block type checking. Introducing a new type class `ring_hom_inv_pair` that states that two ring homomorphisms are inverses of each other, analogous to the type class `ring_hom_comp_triple` described above, again solves this issue.

```
class ring_hom_inv_pair [semiring R₁] [semiring R₂] (σ : R₁ →+* R₂)
  (τ : out_param (R₂ →+* R₁)) : Prop :=
(comp_eq : τ.comp σ = ring_hom.id R₁)
(comp_eq₂ : σ.comp τ = ring_hom.id R₂)
```

Now, with a suitable instance stating that the conjugation operation on a type supporting conjugation is its own inverse, we can work with a conjugate-linear equivalence `e : E ≃ₗ⋆ [R] F`, i.e. `e : E ≃ₛₗ[conj] F`, over scalars of that type, and have that its inverse `e.symm` be genuinely of type `F ≃ₗ⋆[R] E`.

```
instance [comm_semiring R] [star_ring R] : ring_hom_inv_pair (conj R) (conj R)
```

5 Fréchet–Riesz representation theorem

In the following three sections we describe results that we were able to formalize at the proper level of generality thanks to our refactor. By the "proper level" of generality, we mean that our results hold generically over the real and complex numbers without case splits or separate declarations.

5.1 The `is_R_or_C` type class

Many results in functional analysis, including those presented here, hold for a field $\mathbb{K} \in \{\mathbb{R}, \mathbb{C}\}$. Such results are usually presented in the literature by giving proofs for the complex case, with the real case following in the obvious way: replace complex conjugation by the identity, i by zero, and so on.

Before beginning our refactor, we introduced a type class `is_R_or_C` to mathlib used to formalize this kind of result. A type that instantiates `is_R_or_C` is a complete nondiscrete field with (real) norm containing an element i and functions `conj`, `re` and `im` that satisfy a number of ad-hoc axioms chosen to mimic the behavior of a field that is either $\mathbb{R}$ or $\mathbb{C}$. The `conj` operator is an involutive ring homomorphism, enabling the notation discussed in Section 4.3. Two global instances stating `is_R_or_C ℝ` and `is_R_or_C ℂ` allow theorems over the generic type class to be specialized immediately to either concrete type. The conjugation operator `conj` is definitionally equal to the identity function in the real case and the complex conjugation function in the complex case. We note an experiment with a similar type class in Isabelle [5].

Working over an `is_R_or_C` field enables many nice features. In particular, the conjugation operator `conj` is definitionally equal to the identity function in the real case and the complex conjugate in the complex case. Hilbert spaces in mathlib are defined over `is_R_or_C` fields. Given two Hilbert spaces `E` and `F` over a field `K`, conjugate-linear maps `E ≃ₗ⋆[K] F` are precisely maps which are semilinear with respect to `conj`, and thus in the real case are linear maps by definition. Within mathlib, this type class has already been used extensively beyond the results mentioned in this paper, notably by Sébastien Gouëzel for stating in correct generality the Hahn–Banach theorem, the smooth case of the inverse function theorem, and more.

```
local notation `𝒦` := algebra_map ℝ _

class is_R_or_C (K : Type*) extends nondiscrete_normed_field K,
  star_ring K, normed_algebra ℝ K, complete_space K :=
(re : K →+ ℝ)
(im : K →+ ℝ)
(I : K)
(I_re_ax : re I = 0)
(I_mul_I_ax : I = 0 ∨ I * I = -1)
(re_add_im_ax : ∀ (z : K), 𝒦 (re z) + 𝒦 (im z) * I = z)
(of_real_re_ax : ∀ r : ℝ, re (𝒦 r) = r)
(of_real_im_ax : ∀ r : ℝ, im (𝒦 r) = 0)
(mul_re_ax : ∀ z w : K, re (z * w) = re z * re w - im z * im w)
(mul_im_ax : ∀ z w : K, im (z * w) = re z * im w + im z * re w)
(conj_re_ax : ∀ z : K, re (conj z) = re z)
(conj_im_ax : ∀ z : K, im (conj z) = -(im z))
(conj_I_ax : conj I = -I)
(norm_sq_eq_def_ax : ∀ (z : K), ∥z∥^2 = (re z) * (re z) + (im z) * (im z))
(mul_im_I_ax : ∀ (z : K), (im z) * im I = im z)
(inv_def_ax : ∀ (z : K), z⁻¹ = conj z * 𝒦 ((∥z∥^2)⁻¹))
(div_I_ax : ∀ (z : K), z / I = -(z * I))
```

Figure 2 The `is_R_or_C` type class is satisfied only by fields isomorphic to $\mathbb{R}$ or $\mathbb{C}$. The `star_ring` assumption endows `K` with an involutive operator `conj` that respects addition and multiplication.

5.2 Fréchet–Riesz representation theorem

Our first application of semilinear maps is in proving the Fréchet–Riesz representation theorem. While the real case has been formalized in Coq [7] and Mizar [26], and the complex case in Isabelle [11], we are not aware of a development that unifies the two.[4]

Given a Hilbert space E, its *dual space* E^* consists of the set of continuous linear functionals on E (i.e. $E^* = \{f : E \to \mathbb{K} \mid f \text{ is linear and continuous}\}$). The dual space certainly includes elements of the form f_v that map $w \in E$ to $\langle v, w\rangle$, and the Fréchet–Riesz representation theorem states that all elements of the dual space are of this form. That is, there exists an (in fact, isometric) equivalence between E and E^* that maps v to to f_v.

The difficulty in formalizing this is that while this equivalence is linear in the real case, in the complex case, it is *conjugate*-linear. The challenge is to construct this object in such a way that (1) there is a common definition for both the real and complex case, and (2) the added complication of conjugate-linearity is completely transparent in the real case. Before our refactor mathlib simply had two separate constructions. We are able to replace those two constructions with the following, which satisfies both requirements stated above:

```
def to_dual [is_R_or_C 𝕂] [inner_product_space 𝕂 E] [complete_space E] :
  E ≃ₗᵢ⋆[𝕂] normed_space.dual 𝕂 E
```

```
lemma to_dual_apply [is_R_or_C 𝕂] [inner_product_space 𝕂 E]
  [complete_space E] {x y : E} : to_dual 𝕂 E x y = ⟪x, y⟫
```

[4] This theorem should not be confused with the Riesz–Markov–Kakutani representation theorem, which has been formalized in Mizar, PVS (unfortunately referred to as the "Riesz representation theorem"), and possibly other proof assistants.

Read aloud this definition says that "a real or complex Hilbert space E is isometrically conjugate-isomorphic to its dual space." But when specialized to the real case, the statement is definitionally equal to "E is isometrically isomorphic to its dual space."

Our proof of this theorem does not differ from the real version of the proof in mathlib prior to our refactor. In fact, the patch unifying the real and complex versions[5] added only 45 lines of code and removed 79; the only change beyond rearranging and documentation was to generalize the statement of the theorem. The Lean implementation of the orthogonal projection on real inner product spaces, a tool used in the proof, had been written by Zhouhang Zhou as a port of work in Coq by Boldo et al. [7].

6 Adjoints of operators on Hilbert spaces

Given a continuous linear map A between two Hilbert spaces E and F, the adjoint of A is the unique continuous linear map $A^* : F \to E$ such that for all $x \in E$ and $y \in F$, $\langle y, Ax\rangle_F = \langle A^*y, x\rangle_E$. The adjoint satisfies a number of properties: it is involutive (i.e. $(A^*)^* = A$), it is an isometry, and, most importantly for our purposes here, it is conjugate-linear. Hence, it was natural to bundle it in mathlib as a conjugate-linear isometric equivalence as follows:

```
def continuous_linear_map.adjoint [is_R_or_C 𝕜] [inner_product_space 𝕜 E]
  [inner_product_space 𝕜 F] [complete_space E] [complete_space F] :
  (E →L[𝕜] F) ≃ₗᵢ⋆[𝕜] (F →L[𝕜] E)
```

```
lemma continuous_linear_map.adjoint_inner_left [is_R_or_C 𝕜]
  [inner_product_space 𝕜 E] [inner_product_space 𝕜 F] [complete_space E]
  [complete_space F] (A : E →L[𝕜] F) (x : E) (y : F) :
  ⟪continuous_linear_map.adjoint A y, x⟫ = ⟪y, A x⟫
```

This definition fully exploits the algebraic formalism built for semilinear maps, including the composition mechanism of Section 4.3. For example, the statement that the composition of the adjoint operation with itself is equal to the identity map from `E →L[𝕜] F` to itself (a "true" $\mathbb{K}$-linear map) would not typecheck without the `ring_hom_comp_triple` mechanism.

In finite dimension, every linear map is a continuous linear map, so the adjoint construction actually applies to every linear map. We provide this construction as `linear_map.adjoint` for the benefit of future users interested only in the finite-dimensional setting.

An operator T on a Hilbert space is said to be *self-adjoint* if $T = T^*$ and *normal* if $T^*T = TT^*$. We allow these definitions to apply both to the finite-dimensional setting with `linear_map.adjoint` and to the general setting with `continuous_linear_map.adjoint` by in fact writing these definitions in the more general context of a `star_ring`, a ring equipped with a fixed involutive ring homomorphism.

```
def self_adjoint [ring R] [star_ring R] : add_subgroup R :=
{ carrier := {x | star x = x},   ... }
```

```
def is_star_normal [ring R] [star_ring R] (x : R) :=
star x * x = x * star x
```

When `R` is the ring `E →ₗ[𝕜] E` of linear endomorphisms of a finite-dimensional inner product space `E` (with ring operation composition), the involution `star` is `linear_map.adjoint`. When `R` is the ring `E →L[𝕜] E` of continuous linear endomorphisms of a Hilbert space `E`, the involution `star` is `continuous_linear_map.adjoint`.

[5] https://github.com/leanprover-community/mathlib/pull/9924

7 Versions of the spectral theorem

7.1 The ℓ^2 construction

The spectral theorem, in finite dimension also known as the diagonalization theorem, expresses an operator on a Hilbert space in the canonical form of a "diagonal" operator. To describe this canonical form, one needs some version of the *Hilbert sum* or ℓ^2*-space* constructions. Before we started, mathlib already had a finitary version of this construction, namely the following construction for an inner product space structure on the product of finitely many inner product spaces.

```
def pi_Lp {ι : Type u} (p : ℝ) (G : ι → Type v) : Type (max u v) :=
Π (i : ι), G i
```

```
instance pi_Lp.inner_product_space {𝕜 : Type w} [is_R_or_C 𝕜]
  {ι : Type u} [fintype ι] (G : ι → Type v)
  [Π (i : ι), inner_product_space 𝕜 (G i)] :
  inner_product_space 𝕜 (pi_Lp 2 G)
```

Note that the `p` parameter is not used in the definition of `pi_Lp`. A normed space structure that depends on `p` is defined on this family of types.

We require the general version of this construction, with a possibly-infinite index set ι. We first define a predicate `mem_ℓp f p` on dependent functions in `Π (i : ι), G i` which, for $p = 2$, amounts to the norm-squared of the function being a convergent sum. The associated subset of `Π (i : ι), G i` is named `lp G p`, proved to be an additive subgroup, and for $p = 2$ equipped with an inner product space structure. This inner product space is called the Hilbert sum of the family `G`. In the general version we allow `p` to be an extended nonnegative real.

```
def lp {ι : Type u} (G : ι → Type v) [Π (i : ι), normed_group (G i)]
  (p : ℝ≥0∞) : add_subgroup (Π (i : ι), G i) :=
{ carrier := {f | mem_ℓp f p},   ... }
```

```
instance lp.inner_product_space {ι : Type u} {𝕜 : Type w} [is_R_or_C 𝕜]
  {G : ι → Type v} [Π (i : ι), inner_product_space 𝕜 (G i)] :
  inner_product_space 𝕜 (lp G 2)
```

This is a reasonably labor-intensive construction (some 500 lines of code), the difficulties being a series of small analytic arguments about the convergence of the sums involved. It is closely analogous to Rémy Degenne's mathlib construction of the inner product space structure on $L^2(X, G)$, with related work in Isabelle [15]. However, neither construction is a strict generalization of the other: the L^2 construction allows for integrals with respect to an arbitrary measure rather than just sums, whereas the ℓ^2 construction applies to dependent functions of type `Π (i : ι), G i` in which the "codomain" varies depending on the argument. We in fact need this dependent property for the spectral theorem.

A further analytic argument establishes the completeness of ℓ^p. The key step here is an argument that a pointwise limit of a uniformly-bounded sequence of elements of ℓ^p is itself in ℓ^p. is A Hilbert space is by definition a complete inner product space and therefore this establishes that the Hilbert sum `lp G 2` is a Hilbert space.

```
instance lp.complete_space {ι : Type u} {G : ι → Type v}
  [Π (i : ι), normed_group (G i)] [∀ (i : ι), complete_space (G i)]
  {p : ℝ≥0∞} [fact (1 ≤ p)] : complete_space (lp G p)
```

Finally, given a Hilbert space E of interest, an important argument establishes a mechanism for "collating" a family of isometries from the summands `G i` into E to an isometric isomorphism from `lp G 2` into E. It is sufficient (and necessary) that the images of the family of isometries form a mutually-orthogonal family of subspaces of E, and that their joint span be dense in E.

```
def orthogonal_family.linear_isometry_equiv [is_R_or_C 𝕜]
  [inner_product_space 𝕜 E] [complete_space E]
  [∏ (i : ι), inner_product_space 𝕜 (G i)] {V : ∏ (i : ι), G i →ₗᵢ[𝕜] E}
  (hV : orthogonal_family 𝕜 V) [∀ (i : ι), complete_space (G i)]
  (hV' : (⨆ (i : ι), (V i).to_linear_map.range).topological_closure = ⊤) :
  E ≃ₗᵢ[𝕜] (lp G 2)
```

We also provide the finitary, i.e. `pi_Lp`, version of this construction.

```
def isometry_L2_of_orthogonal_family
  [is_R_or_C 𝕜] [inner_product_space 𝕜 E] [fintype ι] [decidable_eq ι]
  {V : ι → submodule 𝕜 E} (hV : direct_sum.submodule_is_internal V)
  (hV' : orthogonal_family 𝕜 (λ (i : ι), (V i).subtypeₗᵢ)) :
  E ≃ₗᵢ[𝕜] pi_Lp 2 (λ (i : ι), V i)
```

7.2 Common outline of the spectral theorems

A diagonal operator on `lp G 2` or `pi_Lp 2 G` is an operator that, for some fixed sequence of scalars μ : $\iota \to \mathbb{K}$, sends each dependent function `f : ∏ (i : ι), G i` to the pointwise-rescaled function `λ i, μ i • f i`. The spectral theorem for compact self-adjoint (respectively, normal) operators states that such an operator over `is_R_or_C` (respectively, $\mathbb{C}$) is equivalent to a diagonal operator on `lp G 2`, for some family of inner product spaces G. The finite-dimensional special case, the diagonalization theorem, states that a normal endomorphism of a finite-dimensional inner product space over $\mathbb{C}$ is equivalent to a diagonal operator on some `pi_Lp 2 G`.

The key point of all such theorems, which we defer discussing to Section 7.3, is a proof that every operator from the stated class has an eigenvalue (unless the operator is the trivial operator on the trivial vector space). The proof of this important point is what differs from theorem to theorem. In this subsection we discuss the common part of the proofs of the theorems, namely the reduction to the existence of an eigenvalue.

This part is essentially algebraic and is carried out for a endomorphism of an inner product space E that satisfies the following property, common to those three cases:

```
def inner_product_space.is_normal (T : E →ₗ[𝕜] E) : Prop :=
  ∃ (T' : E →ₗ[𝕜] E), T' ∗ T = T ∗ T' ∧ ∀ x y, ⟪T' x, y⟫ = ⟪x, T y⟫
```

We first show that the eigenspaces of such an operator are mutually orthogonal.

```
lemma orthogonal_family_eigenspaces [is_R_or_C 𝕜] [inner_product_space 𝕜 E]
  {T : E →ₗ[𝕜] E} (hT : inner_product_space.is_normal T) :
  orthogonal_family 𝕜 (λ (μ : 𝕜), (eigenspace T μ).subtypeₗᵢ)
```

This puts us in a position to apply the final construction from Section 7.1 to the collection of eigenspaces of T. Specifically, if the completeness property `(⨆ (μ : 𝕜), (eigenspace T μ)).topological_closure = ⊤` or its finite-dimensional analogue can be established, then those results establish an isometric isomorphism between E and the Hilbert sum of its own eigenspaces. It is easy to check that the operator T, when transferred by this isometric isomorphism to the Hilbert sum, is diagonal.

A further sequence of lemmas leads to this completeness property, and it is here that the eigenvalue existence result is required. It is shown that an `inner_product_space.is_normal` operator preserves orthogonal complements of eigenspaces.

```
lemma invariant_orthogonal_eigenspace [is_R_or_C 𝕂] [inner_product_space 𝕂 E]
  {T : E →ₗ[𝕂] E} (hT : inner_product_space.is_normal T) (μ : 𝕂) (v : E)
  (hv : v ∈ (eigenspace T μ)^⊥) :
  T v ∈ (eigenspace T μ)^⊥
```

Such an operator preserves the mutual orthogonal complement of all its eigenspaces.

```
lemma orthogonal_supr_eigenspaces_invariant [is_R_or_C 𝕂]
  [inner_product_space 𝕂 E] {T : E →ₗ[𝕂] E}
  (hT : inner_product_space.is_normal T) {|v : E|}
  (hv : v ∈ (⨆ (μ : 𝕂), eigenspace T μ)^⊥) :
  T v ∈ (⨆ (μ : 𝕂), eigenspace T μ)^⊥
```

The restriction of such an operator to this mutual orthogonal complement, which is therefore well-defined, itself has no eigenvalues.

```
lemma orthogonal_supr_eigenspaces [is_R_or_C 𝕂] [inner_product_space 𝕂 E]
  {T : E →ₗ[𝕂] E} (hT : inner_product_space.is_normal T) (μ : 𝕂) :
  eigenspace (T.restrict (orthogonal_supr_eigenspaces_invariant hT)) μ = ⊥
```

From here, if the existence of an eigenvalue for all nontrivial operators in the class considered is known, by contraposition the subspace `(⨆ (μ : 𝕂), eigenspace T μ)^⊥` (being the domain of the operator `T.restrict (orthogonal_supr_eigenspaces_invariant hT)`, which has no eigenvalues) must be trivial. Standard Hilbert space theory implies that the subspace `⨆ (μ : 𝕂), eigenspace T μ` must be dense, the desired completeness result.

7.3 Existence of an eigenvalue

The first version of the spectral theorem we prove is for normal endomorphisms of a finite-dimensional inner product space over $\mathbb{C}$.

```
def diagonalization [inner_product_space ℂ E] [finite_dimensional ℂ E]
  {T : E →ₗ[ℂ] E} (hT : is_star_normal T) :
  E ≃ₗᵢ[ℂ] pi_Lp 2 (λ μ : eigenvalues T, eigenspace T μ)

lemma diagonalization_apply_self_apply [inner_product_space ℂ E]
  [finite_dimensional ℂ E] {T : E →ₗ[ℂ] E} (hT : is_star_normal T) (v : E)
  (μ : eigenvalues T) :
  diagonalization hT (T v) μ = (μ : ℂ) • (diagonalization hT) v μ
```

We also provide the more classical version of this theorem, stating that there exists an orthonormal basis of eigenvectors of `T`.

For this class of operators, the proof of the existence of an eigenvalue is straightforward. In finite dimension, an endomorphism has a well-defined characteristic polynomial. Over an algebraically closed field this polynomial must have a root, and this root is an eigenvalue.

The second version of the spectral theorem we prove is for self-adjoint compact operators on a Hilbert space. Here a map between normed spaces is said to be compact, if the image of every bounded subset has compact closure.

```
def compact_map [nondiscrete_normed_field 𝕂] [normed_group E]
  [normed_space 𝕂 E] [normed_group F] (T : E → F) : Prop :=
∀ s : set E, metric.bounded s → is_compact (closure (T '' s))
```

A compact linear map is automatically continuous, so it is no loss of generality to take `T` to be of type `E →L[𝕂] E`. In this setting we state the spectral theorem as follows.

```
def diagonalization' [is_R_or_C 𝕂] [inner_product_space 𝕂 E]
  [complete_space E] {T : E →L[𝕂] E} (hT : T ∈ self_adjoint (E →L[𝕂] E))
  (hT_cpct : compact_map T) :
  E ≃ₗᵢ[𝕂] (lp (λ μ, eigenspace (T : E →ₗ[𝕂] E) μ) 2)
```

```
lemma diagonalization_apply_self_apply' [is_R_or_C 𝕂]
  [inner_product_space 𝕂 E] [complete_space E] {T : E →L[𝕂] E}
  (hT : T ∈ self_adjoint (E →L[𝕂] E)) (hT_cpct : compact_map T) (v : E)
  (μ : 𝕂) :
  diagonalization' hT hT_cpct (T v) μ = μ • diagonalization' hT hT_cpct v μ
```

For this class of operators, the proof of the existence of an eigenvalue comes from a long and delicate calculation involving the Rayleigh quotient, some 700 lines of code. It is proved that local maxima/minima of the Rayleigh quotient are eigenvectors, that the operator norm of `T` is the supremum of the absolute value of the Rayleigh quotient, and (using the compactness of `T`) that the Rayleigh quotient of `T` achieves its maximum.

Having established in this project the basic properties of compact operators, the infinite-dimensional theorem of the spectral theorem for compact normal operators is also within reach. There, the proof of the existence of an eigenvalue comes from an argument about the resolvent, a holomorphic function with values in the Banach space `E →ₗ[ℂ] E`. The current development of complex analysis in mathlib by Yury Kudryashov [19] is sufficiently general for this setting. However, this would not supersede the spectral theorem we prove for compact self-adjoint operators: the latter works generically over $\mathbb{R}$ and $\mathbb{C}$, which is more elegant than to deduce it in the real setting from the normal-operator version over $\mathbb{C}$ by making an argument about the operator's complexification.

8 Frobenius-semilinear maps and isocrystals

Our formal development of semilinear maps was motivated by applications in functional analysis to unify statements and proofs over $\mathbb{R}$ and $\mathbb{C}$. But these maps are interesting and fruitful objects of study in their own right. As an example of an interesting result about semilinear maps that are *not* linear or conjugate-linear, we formalize the one-dimensional case of a theorem of Dieudonné and Manin [24] (see Demazure [14, chapter 4] for a classical exposition and Lurie [21] for a modern outline without proof), which classifies the isocrystals over an algebraically closed field of characteristic $p > 0$ (Section 2.3).

We denote the ring of `p`-typical Witt vectors over `k` by `𝕎 k` and the field of fractions of this ring by `K(p, k)`. This was defined in mathlib by Commelin and Lewis [12], along with the Frobenius endomorphism `frobenius : 𝕎 k →+* 𝕎 k`.

For the remainder of this section, we work in a context where `p` is a prime natural number and `k` is an integral domain of characteristic `p` with a `p`th root function.

```
variables (p : ℕ) [fact p.prime]
  {k : Type*} [comm_ring k] [is_domain k] [char_p k p] [perfect_ring k p]
```

Since the base ring `k` has characteristic `p`, `frobenius` satisfies the following property:

```
lemma coeff_frobenius_char_p (x : 𝕎 k) (n : ℕ) :
  (frobenius x).coeff n = (x.coeff n) ^ p
```

The additional hypothesis that `k` has a `p`th root function implies that `frobenius` is in fact an automorphism, and with `k` an integral domain, this induces an automorphism on the field of fractions `K(p, k)`. Locally we let φ`(p, k)` denote this map.

We will be interested in maps between `K(p, k)`-vector spaces that are semilinear in φ ("Frobenius-semilinear"). To facilitate the use of these maps, we add an instance of `ring_hom_inv_pair` (Section 4.3) for φ and its inverse. We also introduce notation `V` $\to^{fl}$ `[p, k] V₂` and `V` $\simeq^{fl}$`[p, k] V₂` for the types of Frobenius-semilinear maps and equivalences.

An *isocrystal* is a vector space over the field `K(p, k)` additionally equipped with a Frobenius-semilinear automorphism.

```
class isocrystal (V : Type*) [add_comm_group V] extends module K(p, k) V :=
(frob : V ≃ᶠˡ[p, k] V)
```

We denote the map `frob` by `Φ(p, k)`. We say two isocrystals over `K(p, k)` are equivalent (denoted `V` $\simeq^{fi}$`[p, k] V₂`) if there is a linear equivalence `f : V` $\simeq_l$`[K(p, k)] V₂` which is "Frobenius-equivariant," that is, for all `x`, `Φ(p, k) (f x) = f (Φ(p, k) x)`.

The Dieudonné–Manin theorem classifies the isocrystal structures in every finite dimension, up to this notion of equivalence, over an algebraically closed field `k`. We restrict our attention to the one-dimensional case, where the classification can be stated quite explicitly. The field `K(p, k)` is naturally a vector space over itself with dimension 1. There is a standard family of Frobenius-semilinear automorphisms `K(p, k)` $\simeq^{fl}$`[p, k] K(p, k)` indexed by the integers, namely `p^m` $\cdot$ φ`(p, k)` for each `m : ℤ`, where the Frobenius automorphism φ`(p, k)` is itself considered as a Frobenius-semilinear automorphism. This induces a $\mathbb{Z}$-indexed family of distinct isocrystals which we refer to as `standard_one_dim_isocrystal p k m`, and we prove that any one-dimensional isocrystal is equivalent to one of the these standard isocrystals.

```
lemma classification [field k] [is_alg_closed k] [char_p k p]
  [add_comm_group V] [isocrystal p k V] (h_dim : finrank K(p, k) V = 1) :
  ∃ (m : ℤ), nonempty (standard_one_dim_isocrystal p k m ≃ᶠⁱ[p, k] V)
```

The key to proving this statement is finding, for any `a, b : 𝕎 k` with nonzero leading coefficients, a vector `x : 𝕎 k` such that `frobenius x * a = x * b`. We define such an `x` coefficient by coefficient by an intricate recursion that invokes the algebraic closedness of `k` at each step to solve a new polynomial equation. The argument requires us to mediate between different "levels" of polynomials – universal multivariate polynomials over $\mathbb{Z}$, and multivariate and univariate polynomials over `k` – which proved challenging. Arithmetic operations on Witt vectors are notoriously complicated, and the machinery for universal calculations introduced by Commelin and Lewis [12] does not apply here. This key lemma takes 550 lines to establish.

The remainder of the proof of the isocrystal classification theorem was remarkably straightforward. We needed to extend mathlib's Witt vector library to show that when `k` is an integral domain, `𝕎 k` is too. Modulo this and the previous key lemma, the proof (including the definitions of Frobenius-semilinear maps and isocrystals) takes only 100 lines.

9 Related work

Given the fundamental importance of linear algebra, it is no surprise that theories have been developed in many proof assistants. To our knowledge, none of these libraries define semilinear maps, none prove the spectral theorem for compact operators, and none prove any of the results we describe generically over $\mathbb{R}$ and $\mathbb{C}$.

Mahmoud, Aravantinos, and Tahar [23] and Afshar et al. [3] both describe developments in HOL Light of complex vector spaces. Both use encodings inherently specific to the complex case; they do not generalize the work over the reals by Harrison [16].

Aransay and Divasón [4] introduce vector spaces over arbitrary fields to Isabelle/HOL, using a careful combination of type classes and Isabelle's *locale* feature. A paper by the same authors [5] describes an experiment to generalize the Isabelle definition of a real inner product space to a larger class of fields, using a type class that seems analogous to our class `is_R_or_C` (Section 5.1). Implementing this idea systematically would probably involve providing a locale-based generalization of *euclidean-space* at the beginning of the Isabelle/HOL mathematical analysis library, and the authors do not take this project on, despite noting how useful the generalization would be.

An Isabelle Archive of Formal Proofs entry by Caballero and Unruh [11] duplicates much of the real vector space development in the complex setting, in the process introducing conjugate-linear maps and the complex adjoint operator. Little infrastructure seems to be shared between the real and complex cases. Their development includes a proof of Fréchet–Riesz over $\mathbb{C}$, but does not indicate how it might specialize to $\mathbb{R}$. Also motivated by applications in quantum computation, Bordg et al. [9] define the conjugate-transpose, the analogue of the adjoint in the matrix setting, but again do not generalize to arbitrary fields.

Perhaps related to the more expressive type theory, Coq developments of linear algebra have taken more advantage of type polymorphism. The Mathematical Components library [22] features a theory of modules over arbitrary scalar rings, as does Coquelicot [8]. Building on both these libraries, MathComp-Analysis [1] develops structures used in functional analysis. A `linear_for` predicate in Mathematical Components expresses a concept which is mathematically slightly more general than our `semilinear_map` definition, but which has less convenient properties under composition and inversion. In a branch of the Mathematical Components repository,[6] Cohen defines Hermitian forms, which diverge in behavior over $\mathbb{R}$ and $\mathbb{C}$ similar to conjugate-linear maps. The approach here has some similarities to ours, but preserves fewer definitional equalities; in particular, our conjugate-linear maps on $\mathbb{R}$ are definitionally linear maps, while the analogous statement does not hold for the Mathematical Components approach to Hermitian forms.

Boldo et al. [7] prove the real case of Fréchet–Riesz using Coquelicot, on the way to the Lax–Milgram theorem, but do not address the complex case. Narita et al. [26] do the same in Mizar. Cohen proves the diagonalization theorem for normal matrices in the same of the Mathematical Components repository.[6] This is mathematically equivalent to the diagonalization theorem for normal endomorphisms of a finite-dimensional space described at the start of Section 7.3. Cohen's matrix version could more easily be converted for use in verified numerical analysis, whereas the abstract linear-map version we provide is more convenient in mathematical applications and also admits a more streamlined proof.

References

1 Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, and Kazuhiko Sakaguchi. Competing inheritance paths in dependent type theory: A case study in functional analysis. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2020. doi:10.1007/978-3-030-51054-1_1.

[6] https://github.com/math-comp/math-comp/pull/207

2 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.

3 Sanaz Khan Afshar, Vincent Aravantinos, Osman Hasan, and Sofiène Tahar. Formalization of complex vectors in higher-order logic. In Stephen M. Watt, James H. Davenport, Alan P. Sexton, Petr Sojka, and Josef Urban, editors, *Intelligent Computer Mathematics - International Conference, CICM 2014, Coimbra, Portugal, July 7-11, 2014. Proceedings*, volume 8543 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 2014. doi:10.1007/978-3-319-08434-3_10.

4 Jesús Aransay and Jose Divasón. Generalizing a mathematical analysis library in Isabelle/HOL. In Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, volume 9058 of *Lecture Notes in Computer Science*, pages 415–421. Springer, 2015. doi:10.1007/978-3-319-17524-9_30.

5 Jesús Aransay and Jose Divasón. A formalisation in HOL of the fundamental theorem of linear algebra and its application to the solution of the least squares problem. *J. Autom. Reason.*, 58(4):509–535, April 2017. doi:10.1007/s10817-016-9379-z.

6 Anne Baanen. Use and abuse of instance parameters in the Lean mathematical library. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving (ITP 2022)*, Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2022. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

7 Sylvie Boldo, François Clément, Florian Faissole, Vincent Martin, and Micaela Mayero. A Coq formal proof of the Lax–Milgram theorem. 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 79–89. ACM, 2017. doi:10.1145/3018610.3018625.

8 Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for Coq. *Mathematics in Computer Science*, 9(1):41–62, March 2015. doi:10.1007/s11786-014-0181-1.

9 Anthony Bordg, Hanna Lachnitt, and Yijun He. Certified quantum computation in Isabelle/HOL. *J. Autom. Reason.*, 65(5):691–709, 2021. doi:10.1007/s10817-020-09584-7.

10 Kevin Buzzard, Johan Commelin, and Patrick Massot. Formalising perfectoid spaces. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 299–312, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372885.3373830.

11 Jose Manuel Rodriguez Caballero and Dominique Unruh. Complex bounded operators. *Archive of Formal Proofs*, September 2021. Formal proof development. URL: https://isa-afp.org/entries/Complex_Bounded_Operators.html.

12 Johan Commelin and Robert Y. Lewis. Formalizing the ring of Witt vectors. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021, pages 264–277, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3437992.3439919.

13 Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing.

14 M. Demazure. *Lectures on p-Divisible Groups*. Lecture Notes in Mathematics. Springer Berlin Heidelberg, 2006.

15 Sébastien Gouëzel. Lp spaces. *Archive of Formal Proofs*, October 2016. Formal proof development. URL: https://isa-afp.org/entries/Lp.html.

16 John Harrison. The HOL Light theory of Euclidean space. *J. Autom. Reason.*, 50(2):173–190, 2013. doi:10.1007/s10817-012-9250-9.

17 Michiel Hazewinkel. Witt vectors. Part 1. *Handbook of Algebra*, pages 319–472, 2009. doi:10.1016/s1570-7954(08)00207-6.

18 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 - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 279–294. Springer, 2013. doi:10.1007/978-3-642-39634-2_21.

19 Yury Kudryashov. Formalizing the divergence theorem and the Cauchy integral formula in Lean. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving (ITP 2022)*, Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2022. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

20 Peter Lammich and Andreas Lochbihler. Automatic refinement to efficient data structures: A comparison of two approaches. *J. Autom. Reason.*, 63(1):53–94, 2019. doi:10.1007/s10817-018-9461-9.

21 Jacob Lurie. Lecture notes on the Fargues–Fontaine curve. Lecture 26: Isocrystals, December 2018. URL: https://www.math.ias.edu/~lurie/205notes/Lecture26-Isocrystals.pdf.

22 Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, November 2020. doi:10.5281/zenodo.4282710.

23 Mohamed Yousri Mahmoud, Vincent Aravantinos, and Sofiène Tahar. Formalization of infinite dimension linear spaces with application to quantum theory. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*, volume 7871 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2013. doi:10.1007/978-3-642-38088-4_28.

24 Ju. I. Manin. Theory of commutative formal groups over fields of finite characteristic. *Uspehi Mat. Nauk*, 18(6 (114)):3–90, 1963.

25 The mathlib Community. The Lean mathematical library. In *CPP*, pages 367–381, New York, NY, USA, 2020. ACM. doi:10.1145/3372885.3373824.

26 Keiko Narita, Noboru Endou, and Yasunari Shidama. The orthogonal projection and the Riesz representation theorem. *Formaliz. Math.*, 23(3):243–252, 2015. doi:10.1515/forma-2015-0020.

27 Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21(4):795–825, 2011. doi:10.1017/S0960129511000119.

28 Floris van Doorn, Gabriel Ebner, and Robert Y. Lewis. Maintaining a library of formal mathematics. In Christoph Benzmüller and Bruce Miller, editors, *Intelligent Computer Mathematics*, pages 251–267, Cham, 2020. Springer International Publishing.

29 Eric Wieser. Scalar actions in Lean's mathlib. *CoRR*, abs/2108.10700, 2021. arXiv:2108.10700.

30 Niklaus Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, April 1971. doi:10.1145/362575.362577.

Formalising Fisher's Inequality: Formal Linear Algebraic Proof Techniques in Combinatorics

Chelsea Edmonds[1]
Department of Computer Science and Technology, University of Cambridge, UK

Lawrence C. Paulson
Department of Computer Science and Technology, University of Cambridge, UK

Abstract

The formalisation of mathematics is continuing rapidly, however combinatorics continues to present challenges to formalisation efforts, such as its reliance on techniques from a wide range of other fields in mathematics. This paper presents formal linear algebraic techniques for proofs on incidence structures in Isabelle/HOL, and their application to the first formalisation of Fisher's inequality. In addition to formalising incidence matrices and simple techniques for reasoning on linear algebraic representations, the formalisation focuses on the linear algebra bound and rank arguments. These techniques can easily be adapted for future formalisations in combinatorics, as we demonstrate through further application to proofs of variations on Fisher's inequality.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Theory of computation → Higher order logic; Mathematics of computing → Combinatorics

Keywords and phrases Isabelle/HOL, Mathematical Formalisation, Fisher's Inequality, Linear Algebra, Formal Proof Techniques, Combinatorics

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.11

Related Version *Full Version*: `https://arxiv.org/abs/2207.02728`

Supplementary Material *Software (Formalisation Repository)*: `https://www.isa-afp.org/entries/Fishers_Inequality.html`; archived at `swh:1:snp:6189000751ad63f2df385fc3e5a5fbd2996bc358`

Funding *Chelsea Edmonds*: Funded jointly by the Cambridge Trust (Cambridge Australia Scholarship) and a Cambridge Department of Computer Science Qualcomm Premium Research Scholarship. *Lawrence C. Paulson*: Supported by the ERC Advanced Grant ALEXANDRIA (Project GA 742178).

Acknowledgements Thanks to Wenda Li, for the helpful pointers on working with linear algebra in Isabelle, and Yiannos Stathopoulos, for the suggestions on utilising SErAPIS to navigate results from past formalisations.

1 Introduction

The last decade has seen increasing interest in establishing libraries of formalised mathematics, to verify correctness, gain deeper insights into proofs, and benefit from tools such as automation. While there has recently been an increase in combinatorial formalisations, the field remains under-represented in comparison to more traditional areas of mathematics, and presents several challenges to formalisation efforts. This includes the intuitive nature of traditional proofs, discrepancies in definitions, and a reliance on sometimes surprising results and techniques from other fields of mathematics. Furthermore, current formalisations in combinatorics predominantly focus on graph theory and have often been the result of proving major theorems or verifying algorithms rather than building foundational libraries. Gonthier's well-known four colour theorem formalisation in Coq [13] is one such example.

[1] Corresponding Author

Many combinatorial structures, such as graphs and designs, are types of incidence set systems. Unlike more traditional fields of mathematics, a significant amount of research into these structures has been driven by their applications in computer science. The formal verification of some of these applications further motivates the need for foundational formalisations in the field. It is common to see repeated techniques in proofs on these structures, drawing on a wide range of other mathematical fields such as linear algebra, probability, and group theory. From a formalisation perspective, this motivates research in two areas: firstly generalising the formalisation of a proof pattern to easily structure proofs and reduce duplication, and secondly exploring how we may be able to suggest the application of such a proof pattern.

This paper addresses the first point with a focus on linear algebra, aiming to make formalisation of results using common techniques more accessible. Linear algebra is used in the proofs of many foundational results on incidence systems. While more traditional combinatorial proofs have since been found in some cases, in others linear algebra remains the sole known way to prove a theorem, or presents a much cleaner and general proof [12][2].

Currently no formalisations of combinatorial proofs using a linear algebraic methodology exist. This paper presents the formalisation of linear algebraic representations of set systems, building on our prior work in design theory [9] and Isabelle/HOL's linear algebra libraries such as matrices in [25]. We develop alternate formal reasoning techniques for set systems using these representations, and the formalisation of two particularly notable proof methods: the rank argument and linear bound method. These techniques are applied to the first formalisation of Fisher's inequality, a consequential result on the bounds of set systems, which is also credited for the initial development of the linear algebra method in combinatorics. Informally, the generalised version of Fisher's inequality states that given an incidence set system of n points and m distinct subsets with a constant intersection number, m must be less than or equal to n. Godsil [12] labels Fisher's inequality as a *principle*, stating the result is simply *"too important, and too useful, to be termed a theorem"*.

This paper begins with (2) a summary of necessary mathematical background and related formalisation work, followed by (3) the formalisation of incidence matrices for set systems. In (4) and (5) we discuss formal methodology for the rank and linear bound techniques respectively, and in (6) present the formalisation of a number of variations of Fisher's inequality. We conclude in (7) with a discussion of the formalisation process when drawing on results from multiple fields, and the challenges and advantages of the linear algebra approach compared to traditional combinatorial proof in a formal environment.

2 Background

2.1 Mathematical Background

Incidence set systems are the foundation of many important structures in combinatorics, including graphs, designs, and hypergraphs. We give the design theoretic definition:

▶ **Definition 1** (Incidence Set System). *An incidence set system* $(\mathcal{V}, \mathcal{B})$ *is a collection of subsets* $\mathcal{B}$ *called* blocks *of a finite set* $\mathcal{V}$ *of* points. *A system is* simple *if* $\mathcal{B}$ *is a set, i.e. there are no repeated blocks. A* design *is a finite incidence system with no empty blocks.*

Incidence structures become interesting mathematically by restricting certain properties to impose structural conditions. We give the four most common properties here [7]:

(i) *Block size* (k_B): The cardinality of a block in the set system. A design is incomplete if $k_B < v$ for all blocks $B \in \mathcal{B}$, where $v = |\mathcal{V}|$.
(ii) *Replication number* (r_x): the number of blocks point x occurs in.
(iii) *Points index* (λ_T): The number of blocks a T subset of points occurs in.
(iv) *Intersection number*: The number of points two blocks in a design intersect on.

Using these properties we can define certain subtypes of incidence systems. For example, we can model a *constant intersect* family of sets by imposing a condition that the intersection number for any two blocks in a design is constant. Similarly, we can model specific types of designs such as a *pairwise balanced design* (PBD), which has a constant points index λ for any pair of points, or a *balanced incomplete block design* (BIBD), which is a PBD with added conditions of uniform block size k and incompleteness.

Linear algebraic techniques can initially appear disparate, however there are a number of basic principles that guide their use [12, 2]. Firstly, we define the linear algebraic representation of set systems using matrices, a common representation in both mathematics and computer science applications. For incidence systems, we define an *incidence matrix* based on the incidence relation, and its columns as *incidence vectors*. A point x is *incident* with a block B of the design if $x \in B$. Formally an incidence matrix is defined as follows:

▶ **Definition 2** (Incidence Matrix). *For a set of points* $\{x_1 ... x_v\}$ *and collection of subsets* $\{B_1 ... B_b\}$, N *is a* $v \times b$ *incidence matrix of the system defined by:*

$$N_{i,j} = \begin{cases} 1 & \text{if } x_i \in B_j \\ 0 & \text{if } x_i \notin B_j \end{cases}$$

Matrix and vector representations enable the use of numerous results from linear algebra, which Babai and Frankl summarise [2]. The basic ideas used in both the linear algebra bound and rank arguments are similar, mapping combinatorial objects to vector or matrix representations and using well known theorems on rank, dimension and linear independence to solve problems in extremal combinatorics [17]. The incidence relation is one such basic mapping, which is used in the proof of Fisher's inequality. Further detail on the rank argument, linear algebra bound method, and Fisher's inequality is provided as needed in later sections.

2.2 Related Formalisation Work

To date, the majority of combinatorics formalisations have been done in Isabelle/HOL, Coq, Lean, and Mizar. Based on a survey of combinatorial results across these systems, Isabelle/HOL (henceforth Isabelle) appears to have the most significant range of general combinatorial results, with the majority in both computer science and mathematics entries within the Isabelle Archive of Formal Proofs (AFP). Isabelle also has significant libraries in linear algebra, making it ideal for this work.

Aside from numerous graph theory libraries, of which the most extensive is presented by Noschinksi [21], there are limited formalisations of incidence set systems in Isabelle, including a small matroid library [18], and our previous work on design theory [9], which this work extends. Furthermore, no existing formalisation of Fisher's inequality or formalisations of linear algebraic proofs for combinatorics are known to be available in any system.

Isabelle's linear algebra formalisations are scattered across both the main library and AFP, with several concepts having multiple representations. A key and relevant example of this is matrices, for which there are three main formalisations: Harrison's representation [15] in the HOL-Analysis library (ported from HOL-Light), Obua's entry on finite matrices based on lists [22], and the matrix and vector library developed as part of the Jordan Normal Form

(JNF) library. As Thiemann and Yamada address [25], this last library provides significant flexibility over past definitions. It removes the type constraints in Harrison's definition where the dimensions of a matrix are modelled by the size of types, while also enabling reasoning at a more abstract level than [22]. Additionally, the JNF library formalises a number of necessary results on rank, and provides links to earlier vector representations so a large part of Isabelle's existing linear algebra libraries, such as Aransay's work [1] can still be used.

2.3 Isabelle and Locales

Isabelle/HOL is an interactive proof assistant built on higher order logic. In addition to the necessary formalisation background presented above, it has several features which proved critical in this formalisation. Firstly, Isabelle's Isar proof language [26] makes it easier to structure proofs for straightforward application of general techniques, and provides an easily readable syntax. Sledgehammer, Isabelle's automation system, also proved highly useful throughout the formalisation process.

This formalisation builds on prior work [9] which emphasises use of locales, Isabelle's module system [3], to establish a flexible and easily extendable mathematical structure hierarchy. Locales are an important element of the Isar proof language, providing persistent contexts which can be used across numerous theories drawing on similar structures. While not a new feature, first introduced in their current form in 2004, their full power for formalising mathematics has only recently been realised.

In the simplest form, a locale declaration introduces parameters (with a specified type) and assumptions. Once defined, a locale can be extended with definitions, notation, and theorems within its context. Locale expressions were designed to support multiple inheritance diamonds. Existing locales can be combined to create a new locale, and extended by adding new parameters and assumptions. The locale hierarchy can easily be transformed using the **sublocale** command, which is used to show indirect inheritance between two separately specified locales. It is also possible to instantiate locale parameters and instances through locale expressions and interpretations, in both proof and theory contexts.

3 Incidence Matrices

The foundation of any linear algebraic proof in combinatorics is the matrix or vector representation. This section covers the formalisation of incidence matrices and basic proof techniques for formal reasoning. We note that the techniques used in this section would be straightforward to adapt to formalise other linear algebraic representations, such as adjacency matrices on graphs.

3.1 Ordering Incidence Systems

The first challenge a matrix representation presents is that it is inherently ordered. As can be seen from definition (2), we (arbitrarily) label the blocks and points of a system with numbers so that we are able to refer to a certain point as a row index, and block as a column index. This is similarly required in Isabelle, as the matrix formalisation indexes rows and columns using natural numbers. As such, the existing incidence system formalisation must be adapted to be able to arbitrarily impose an ordering on the block collection and point set.

While a bijective index function would be suitable for points, bijections on multisets are less well defined and no existing formalisation exists. Instead, we created an alternate representation of incidence systems using lists. This method enables us to utilise the large

library of results on lists and translates easily to vector representations. Additionally, it has the advantage of being the representation used in several common programming libraries for designs, such as the GAP library [23]. We define the *ordered-incidence-system* locale below:

locale ordered-incidence-system =
  fixes $\mathcal{V}$s :: 'a list **and** $\mathcal{B}$s :: 'a set list
  assumes wf-list: b ∈# (mset $\mathcal{B}$s) ⟹ b ⊆ set $\mathcal{V}$s **and** distinct: distinct $\mathcal{V}$s

The flexibility of Isabelle's locale mechanism and our previous set theory formalisation, makes it straightforward to prove this locale is a type of finite incidence system through a sublocale declaration:

sublocale ordered-incidence-system ⊆ finite-incidence-system set $\mathcal{V}$s mset $\mathcal{B}$s

This avoids the need for a single locale to contain unnecessary information, such as both the ordered lists and original sets. We additionally use the permutations of set and multiset functions to define an alternate introduction rule for an ordered incidence system. The support of multiple inheritance for locales makes it easy to combine this new locale with various existing sub types of incidence structures and in turn create new ordered contexts for specific properties. Within the *ordered-incidence-system* locale, we provide a range of base lemmas to be able to reason on mappings between the set and list based representations, as well as valid indexes in the orderings.

3.2 Constant Intersect Designs

To enable the formalisation of Fisher's inequality in its multiple forms, a number of extensions are required to the existing design theory library [9]. In particular, constant intersect set systems are crucial to reasoning for the non-uniform Fisher's inequality. While the intersection number was previously defined, limited properties were proven. Using the same approach to building the original design theory hierarchy, we define a new locale to reason on incidence systems with a constant intersection number:

locale const-intersect-design = proper-design +
  fixes m :: nat
  assumes const-intersect: b1 ∈# $\mathcal{B}$ ⟹ b2 ∈# ($\mathcal{B}$ − {#b1#}) ⟹ b1 |∩| b2 = m

3.3 Incidence Matrix Construction

Incidence matrices are relatively straightforward to construct given a point and block listing. To enable reasoning on the equality between a set system and an arbitrary incidence matrix, we define the construction of an incidence matrix outside a designated locale:

definition inc-mat-of :: 'a list ⇒ 'a set list ⇒ ('b :: {ring-1}) mat **where**
inc-mat-of Vs Bs ≡ mat (length Vs) (length Bs) (λ (i,j) . if Vs ! i ∈ Bs ! j then 1 else 0)

This produces a 0-1 matrix which we can define numerous base properties for. Note that we use the *ring-1* type for the matrix elements. This type's properties are sufficient for basic reasoning lemmas and calculations on incidence matrices, and enables easy translations to a number of common field types such as $\mathbb{R}$ and $(\mathbb{Z}/2\mathbb{Z})$ when utilising vector space concepts in later parts of the formalisation. Additionally, we define a similar function to produce an incidence vector for a singular block, and prove lemmas on the relationship between these definitions. Within the *ordered-incidence-system* locale, we use N to notate its incidence matrix which is defined as an integer matrix using this function.

3.4 Incidence Matrix Properties

We define a 0-1 matrix context through an assumption on matrix elements for any matrix with elements of type *zero-neq-one*.

locale zero-one-matrix =
fixes matrix :: 'b :: {zero-neq-one} mat (M)
assumes elems01: elements-mat M $\subseteq$ {0, 1}

The *inc-mat-of* definition clearly produces a matrix satisfying this locale's assumptions. Within this environment, we are able to prove a number of basic properties on 0-1 matrices which are useful for reasoning on incidence matrices. Additionally, we define a way to map a given incidence vector v back to a block, assuming the point set is simply $\{0.. < dim(v)\}$.

It is straightforward to prove this mapping results in an incidence system: given any 0-1 matrix of dimension $(v \times b)$, we can construct a system with v points, and b subsets of those points. This proof is possible due to reasoning on 0-1 matrices and general matrix properties being declared outside of the *ordered-incidence-system* locale context. Additionally, it is possible to easily extend this locale to further restrict the type of the matrix elements as needed, such as a 0-1 integer matrix context.

Incidence matrices present an alternate method for reasoning on key properties of a design. For each key set theoretic property, we provide a matrix definition and establish equivalence outside of a locale context. The replication number definition is given below as an example, where the number of blocks a point occurs in is the number of ones in a row.

definition mat-rep-num :: ('a :: {zero-neq-one}) mat $\Rightarrow$ nat $\Rightarrow$ nat **where**
mat-rep-num M i $\equiv$ count-vec (row M i) 1

We can define similar equivalences for balance, intersection and uniformity properties. Note a number of extensions to the vector, matrix, and multiset libraries occurred during this process, included in the final formalisation. These properties enable us to reason on the balance and uniformity conditions of specific types of systems using the incidence matrix in later proofs. For example, the incidence matrix for a block design with uniform block size k will have k ones in each column.

3.5 Incidence Matrices for Simple Proofs

We can now utilise the properties above to begin proving simple results on incidence systems through these alternate representations. Drawing on introductory results from Brualdi and Ryser [5], one such proof is on the *complement* of a design. The complement of a design $(\mathcal{V}, \mathcal{B})$ has the same point set, but takes the block complement $\mathcal{V} - B$ for each $B \in \mathcal{B}$. We prove a design complement's incidence matrix simply flips all ones and zeros.

lemma ordered-complement-mat-map:
ordered-comp.N = map-mat (λx. if x = 1 then 0 else 1) N

Another important concept is that of isomorphisms on designs. It is straightforward to see that if two designs have the same incidence matrix they must be isomorphic, and similarly that two isomorphic designs must have an ordering that produces the same incidence matrix. We prove this formally by establishing a bijective function using the index function on the ordered lists of points. Lastly, Stinson [24], presents a number of results on the existence of certain types of designs based on matrix elements, such as the existence of a regular PBD.

▶ **Theorem 3.** *Let N be a $v \times b$ 0-1 matrix. Then N is the incidence matrix of a regular pairwise balanced design having v points and b blocks if and only if there exist positive integers r and λ such that $NN^T = \lambda J_v + (r - \lambda)I_v$.*

One half of this theorem is formalised in the 0-1 matrix (integer restricted) locale, the other in the regular pairwise balanced locale, which we give below.

lemma rpbd-incidence-matrix-cond: $\mathrm{N} * (\mathrm{N}^T) = \Lambda \cdot_m (\mathrm{J}_m\ \mathrm{v}) + (\mathrm{r} - \Lambda) \cdot_m (1_m\ \mathrm{v})$

3.6 Dual Systems

Dual systems are a crucial concept in design theory and for combinatorial structures more generally. Intuitively a dual swaps the point and block sets of a system, so each block represents a point in the design, and each point represents a block. The Handbook of Combinatorial Designs [7] defines the *dual set system* for $(\mathcal{V}, \mathcal{B})$ as the system $(\mathcal{B}, \mathcal{V})$, where the point B is in a block representing x if and only if $x \in B$ in the original. This definition, while common, is also ambiguous. It doesn't clearly allow for repeated blocks to be distinct points in the dual, which is necessary for properties of a dual to make sense. The ability to impose an arbitrary numbering on the points and blocks enables us to refine the definition to explicitly allow this, which is presented below:

▶ **Definition 4** (Dual System). *Let $\{x_1, \ldots x_v\}$ be the points of a set system with $\{B_1, \ldots, B_b\}$ as the blocks. The dual of the system has points $\mathcal{V}^* = \{1, \ldots, b\}$, and blocks $\mathcal{B}^* = \{B_1^*, \ldots, B_v^*\}$, where a point $y \in V^*$ is in B_n^* for some $1 \leq n \leq v$ if and only if $x_n \in B_y$.*

Using definition 4, we formalise dual designs in the *ordered-incidence-system* locale.

definition dual-blocks-ordered :: nat set list ($\mathcal{B}$s$*$) **where**
dual-blocks-ordered $\equiv$ map (λ x . {y . y < length $\mathcal{B}$s $\wedge$ x $\in$ $\mathcal{B}$s ! y}) $\mathcal{V}$s
interpretation ordered-dual-sys: ordered-incidence-system [0..<length $\mathcal{B}$s] $\mathcal{B}$s$*$

Intuitively, it is clear that switching the role of blocks and points in our original design to define the dual, results in the dual's incidence matrix being the transpose of the original.

lemma dual-incidence-mat-eq-trans: ordered-dual-sys.N = N^T

The dual system and original system have many symmetries when it comes to properties. For example, if a design has a constant intersect number, its dual is balanced. Similarly, the replication number swaps with block size. This is the first example of a case where it becomes significantly simpler to do this reasoning using the incidence matrix version of the property, demonstrating the power of a linear algebraic representation to formalise even simple lemmas. While a traditional counting proof took 26 lines, the proof could be completed in 10 lines using the incidence matrix. One important resulting statement is that the dual of a constant intersect design is a PBD, located within the *ordered-const-intersect-design* locale.

lemma dual-is-pbd:
assumes ($\bigwedge$ x . x $\in$ $\mathcal{V}$ $\Longrightarrow$ $\mathcal{B}$ rep x > 0) **and** b $\geq$ 2
shows pairwise-balance {0..<(length $\mathcal{B}$s)} (dual-blocks $\mathcal{V}$ $\mathcal{B}$s) m

4 The Rank Argument

4.1 Mathematical Technique

The rank argument uses the rank of a matrix to reason about a collection of vectors which form its columns or rows. Given a matrix A, its rank $rk(A)$ is defined as the maximum number of linearly independent column (or row) vectors in the matrix.

Godsil [12] and Bukh [6] both describe the rank argument as one of the foundational techniques in the application of linear algebra to combinatorics. While it can be applied in a number of ways, we present the methodology we use below, given two matrices N and M:

1. Establish a square matrix of the form NM.
2. Prove the columns or rows of the matrix are linearly independent, in this case by proving the determinant is non-zero.
3. Infer that the rank of the matrix must be equal to the number of rows in the matrix.
4. Using well-known theorems $rk(NM) \leq min(rk(N), rk(M))$, and that for any arbitrary matrix A, $rk(A) \leq$ dim-col(A) and $rk(A) \leq$ dim-col(A), infer an inequality between the number of rows and columns in our original matrix N.

4.2 Extending Row/Column Operations in Isabelle

While there are many methods for proving linear independence, in the context of a square matrix showing the determinant is non-zero is a typical textbook approach. When formalising the uniform Fisher's inequality, significant work was required to reason on the determinant formally, where on paper it was a one line description of elementary row and column operations to convert the matrix to an upper triangular form. For this reason, we made several general extensions to the existing row and column operations defined as part of the Gauss-Jordan algorithm formalisation in the AFP [25].

On paper, mathematics involving basic application of these elementary operations will often use terminology such as "add row 1 to all other rows in the matrix". To this effect, we extend the operations to enable addition or subtraction of multiple rows and columns at a time, such as the *add-multiple-rows* function.

```
fun add-multiple-rows :: 'a :: semiring-1 ⇒ nat ⇒ nat list ⇒ 'a mat ⇒ 'a mat where
add-multiple-rows a k [] A = A |
add-multiple-rows a k (l # ls) A = (addrow a k l (add-multiple-rows a k ls A))
```

By using lists, we can easily apply induction in addition to pre-existing lemmas on the base operations to formally reason on the functions. In particular, we prove that the determinant remains unchanged from applying these operations.

4.3 Rank Argument Formalisation

To formalise the rank argument, we first prove a simple lemma stating that the rank of the product of two matrices AB must be less than or equal to the minimum of their individual ranks. The proof is straightforward building on results from the rank theory in [25].

```
lemma rank-mat-mult-lt-min-rank-factor:
  fixes A :: 'a::{conjugatable-ordered-field} mat
  assumes A ∈ carrier-mat n m   and B ∈ carrier-mat m nc
  shows vec-space.rank n (A * B) ≤ min (vec-space.rank n A) (vec-space.rank m B)
```

Using this lemma and other results on rank, we now prove two versions of the rank argument lemma using the product of a matrix and its transpose. We give below the determinant version of the lemma.

lemma rank-argument-det:
fixes M :: ('c :: {conjugatable-ordered-field}) mat
assumes M ∈ carrier-mat x y **and** det (M∗ M^T) ≠ 0
shows x ≤ y

These lemmas can be easily applied to proofs as introduction rules. Effectively, when applied to a proof this version *hides* the rank part of the argument, thus removing the need for someone unfamiliar with the linear algebra libraries in Isabelle to interact with them. In particular, there is no need to interpret a vector space locale, or work within one. This significantly simplifies the proof process. The other version of the lemma gives a more general assumption that is not related to the determinant, enabling greater flexibility if another method is used to prove the rank of the matrix is equal to its column dimension.

5 Linear Algebra Method

5.1 The Methodology

The linear algebra method uses similar ideas to the rank argument, however is both more general and as such more widely applicable. In particular it utilises results on dimension and linear independence of vector spaces.

In general, this method is used to provide an upper bound on the size of a set, what is known as an *extremal combinatorics* problem. There are many fundamental results which use this method in combinatorics, including proofs on graphs, set systems, polynomials, and matrix representations, for which Jukna presents a good overview [17]. The general framework for applying it is to associate each object in a set with elements in a vector space of relatively low dimension and show these elements are linearly independent. The fundamental linear algebra bound theorem can then be used to establish an inequality [17].

▶ **Theorem 5** (Linear Algebra Bound). *Given a set $v_1, \ldots, v_k$ of k linearly independent vectors in a vector space of dimension m, then $k \leq m$.*

Gowers discusses the impact of these dimension tricks in a blog post [14]. He outlines that for many mathematicians the trick with applying the linear algebra method is often choosing the vector space to use, and establishing the characteristic function which maps our objects to elements in that vector space. However, from a formalisation perspective, the proof process itself can be challenging. We aim to simplify this process, to mirror the mathematical problem, where the challenge remains in knowing when to apply the method while the formal proof process itself is straightforward. The argument itself is formalised in such a way that it is easy to apply to vector representations, such as incidence vectors. We envision that it would be simple to extend or use similar patterns for more complex problems. For example, a common extension out of scope of this formalisation is associating a set to a polynomial, and showing linear independence in the corresponding function space. While this requires a slightly more complicated setup, the resulting proof still uses the same proof pattern as what we formalise here.

5.2 Formalisation

We formalise several versions of the linear algebra method, to allow reasoning directly both on matrix and vector set representations. Firstly, the linear algebra bound presented in theorem (5), has previously been formalised as the lemma *li-le-dim*, in the original vector space locale. Our methods aim to set up a framework to use this fact outside of the context of a vector space. The most general version of the lemma is on a set of vectors:

lemma lin-bound-arg-general-set:
fixes A ::('a :: {field})vec set
assumes A ⊆ carrier-vec nr **and** vec-space.lin-indpt-vs nr A
shows card A ≤ nr

It is clear when applying this method, that the majority of formal proof work is required in the linear independence part of the proof, mirroring the textbook proofs. The most foundational method for proving linear independence is to take an arbitrary linear combination over the vector set and prove that all coefficients must equal 0. Using the vector space definition of linear independence for this methodology involved several layers of unfolding definitions related to linear combinations in calculations. The last version of our linear bound method removes the linear independence criterion of the earlier representation, and replaces it with the matrix version of the linear combination condition, which we prove is equivalent in the vector space context.

lemma lin-bound-argument2:
fixes A :: ('a :: {field}) mat
assumes distinct (cols A) **and** A ∈ carrier-mat nr nc
assumes ⋀ f. vec nr (λi. ∑ j ∈ {0..<nc} . f (col A j) ∗ (col A j) $ i) = 0_v nr ⟹
∀ v∈ (set (cols A)). f v = 0
shows nc ≤ nr

This removed a significant amount of repeated work from formalisations when applied. As with the rank argument, it additionally simplified formal reasoning by removing the need to understand numerous definitions from the older vector space library and their translation. However, it is worth noting the restriction on the type of the matrix to a field, which is required to use results from the vector space locale. Utilising this lemma still requires the user to implicitly choose the vector space they work in by determining the type of the matrix they are using, and may require some initial setup.

6 Fisher's Inequality

This section outlines how the above techniques can be applied to prove a number of variations of Fisher's inequality, a consequential result on the relationship between the number of points and blocks in a set system. When discovered, it had immediate implications for applications such as experimental design [2], and as a *necessary* condition for design existence, continues to be a valuable tool in both practical and mathematical applications of incidence set systems more generally. Research on generalising the statement is widely credited for the development of linear algebraic methods in combinatorics. The result also is directly used in further proofs, such as a problem Erdős presented on three point collinearity [19].

6.1 Incidence Matrix Types

In order to use both the linear algebra method and rank argument we must be able to reason on a matrix whose elements are of any *field* type. In the context of an *ordered-incidence-system*, an incidence matrix is defined to have integer elements. However, it should be possible to easily lift the incidence matrix to any type which distinguishes zero and one, while preserving basic properties which do not manipulate the incidence matrix elements. For example, the matrix block size property simply counts the number of one's in a column. To formalise Fisher's inequality and its variations, we established a general methodology for 0-1 matrices which could be used to do this transfer between types.

Note firstly, that it is easy to lift integers to a real type using the *of-int* mapping, an injective homomorphism. Building on homomorphism lemmas in the JNF matrix library [25], it was straightforward to show certain properties would be preserved. However, the ideal situation is to do this for any such field without repeated work. For example, initial attempts at reasoning over $(\mathbb{Z}/2\mathbb{Z})$ required significant additional set up work, resulting in a Isabelle theory for modulo operations on vectors and integers. This is available as part of the formalisation for completeness, but ultimately was only minimally used in the formalisation of Fisher's inequality.

Instead, a *of-zero-neq-one* definition is established, providing a simple mapping from any *zero-neq-one* type to another. Under this definition, zero is mapped to zero, and anything else is mapped to one. In a situation where the elements are restricted to just zero and one, such as incidence matrices, the mapping is clearly an injection on that element set. For simplicity, the *lift-01-mat* definition applies this mapping to a matrix.

definition lift-01-mat :: 'b :: {zero-neq-one} mat ⇒ 'c :: {zero-neq-one} mat **where**
lift-01-mat M ≡ map-mat of-zero-neq-one M

A number of lemmas are formalised to show that key incidence matrix properties are preserved under the application of the above definition in the *zero-one-matrix* context, similar to existing lemmas on injective homomorphism mappings on matrices such as *of-int*. Note that the result of this definition is also clearly still a *zero-one-matrix* itself, and as such immediately inherits all the properties of this locale. This definition is used in both applications of the linear algebra bound method later in this section.

6.2 Uniform Fisher's Inequality

Fisher's inequality was initially proved in 1940 by Fisher [10] as a result on BIBDs, a very structured incidence system. We give this basic statement in theorem 6.

▶ **Theorem 6** (Uniform Fisher's Inequality). *Given a (v, k, λ)-BIBD, with v points, b blocks, uniform block size k and a pairwise points index λ, we have $v \leq b$.*

There are many variations of the proof of Fisher's inequality, but one of the more basic techniques is using the rank argument discussed in Section (4). In Isabelle, the theorem is located within the *ordered-bibd* locale, which has the assumptions in its context.

Notably, the majority of work revolves around calculating the determinant from the diagonal. While the general row and column operations from Section (4.2) are straightforward to apply, it remains fiddly to calculate the values of the resulting elements of the matrix after several operations. We reuse results on the properties of the square matrix NN^T for an incidence matrix N of a BIBD, discussed in Section (3.5).

The final theorem statement in Isabelle is given below, as well as a sketch of the proof demonstrating how easily the general rank argument lemma can be applied.

theorem Fishers-Inequality-BIBD: v ≤ b
proof (intro rank-argument-det[of map-mat real-of-int N v b], simp-all)
show N ∈ carrier-mat v (length $\mathcal{B}$s) **using** blocks-list-length N-carrier-mat **by** simp
let ?B = map-mat (real-of-int) (N ∗ N^T)
have b-split: ?B = map-mat (real-of-int) N ∗ (map-mat (real-of-int) N)^T ⟨*proof*⟩
have db: det ?B = (r + Λ ∗ (v − 1))∗ (r − Λ)^(v − 1) ⟨*proof*⟩
have lhn0: (r + Λ ∗ (v − 1)) > 0 ⟨*proof*⟩
have (r − Λ) > 0 ⟨*proof*⟩
then have det-not-0: det ?B ≠ 0 ⟨*proof*⟩
thus det (of-int-hom.mat-hom N ∗ (of-int-hom.mat-hom N)^T) ≠ (0:: real) ⟨*proof*⟩
qed

6.3 Variations: The Odd Town Problem

The classic introductory problem for the linear algebra method in numerous lecture notes on the subject is that of even and odd towns [2]. In particular, the odd town problem presents a bound similar to Fisher's inequality. It aims to identify the maximum number of clubs a town can have, given the conditions that the intersection of any two clubs is even, and every club has an odd number of people. We present this formally below.

▶ **Theorem 7.** *Given a set $\mathcal{V} = \{0.. < n\}$, and a collection $\mathcal{B}$ of m subsets of $\mathcal{V}$ of odd size, such that for any i, j where $i \neq j$, $B_i \cap B_j$ is even, $\mathcal{B}$ can be at most of size n, i.e. $m \leq n$.*

We can easily represent this problem formally by building on our locales for ordered incidence systems, and definitions on block size and intersection number.

locale odd-town = ordered-design +
assumes odd-groups: bl ∈# $\mathcal{B}$ ⟹ odd (card bl)
and even-inters: bl1 ∈# $\mathcal{B}$ ⟹ bl2 ∈# ($\mathcal{B}$ − {#bl1#}) ⟹ even (bl1 |∩| bl2)

Additionally, the condition for the intersection between any two blocks of odd size to be even in turn implies that no two blocks can be the same. In other words, this is a simple design, which we can prove through the sublocale command.

In this case, given our incidence vectors are 0-1 vectors, we prove they are linearly independent over the vector space $(\mathbb{Z}/2\mathbb{Z})^n$. This clearly has dimension n, our required upper bound. Note that the scalar product of an incidence vector with itself is equal to the block size which is odd: $v_i \cdot v_i = 1 \pmod 2$. Similarly for any two different vectors, the scalar product is the intersection number, which is even: $i \neq j \longrightarrow v_i \cdot v_j = 0 \pmod 2$. These results are formalised in separate lemma statements using the matrix definitions of block size and intersection number. The proof of linear independence proceeds by taking an arbitrary linear combination equal to 0, and multiplying it by an incidence vector v from our set. Using the results on the scalar product, we can prove that the coefficient of v must be 0, as required.

The calculations above are trivial in $(\mathbb{Z}/2\mathbb{Z})$, however this presents the first challenge of formalising the proof. Given Isabelle's foundations in simple type theory, some creativity is required to define a finite field type. For the purpose of this proof, we went with the formalisation presented in the Berlekamp Zassenhaus AFP entry [8]. This uses the lifting and transfer methodology [16] to prove the resulting type *'a mod-ring* is a field, which is required for our linear bound argument lemma. Using the *lift-01-mat* definition from Section (6.1), it is possible to easily lift our incidence matrix to this type.

The formal statement of the lemma is within the odd clubs locale:

```
lemma upper-bound-clubs:
  assumes CARD('b::prime-card) = 2
  shows b ≤ v
proof −
  have cb2: CARD('b) = 2 using assms by simp
  define N2 :: 'b mod-ring mat where N2 ≡ lift-01-mat N
  show ?thesis proof (intro lin-bound-argument2[of N2])
    show distinct (cols (N2)) ⟨proof⟩
    show n2cm:N2 ∈ carrier-mat v b ⟨proof⟩
    show ⋀f. vec v (λi. ∑j = 0..<b. f (col N2 j) ∗ (col N2 j) $ i) = 0_v v ⟹
     ∀v∈set (cols N2). f v = 0
    proof (auto)
      fix f v  assume vin: v ∈ set (cols N2)
      assume eq0: vec v (λi. ∑j = 0..<length Bs. f (col N2 j) ∗ (col N2 j) $ i) = 0_v v
       ⟨Isar linear independence proof details omitted⟩
      then show f v = 0 ⟨proof⟩
    qed
  qed
qed
```

The proof sketch above clearly shows the structure of the formal proof using the linear bound lemma from Section (5.2). The initial part of the formalisation applies the *lift-01-mat* definition to establish *N2*, an incidence matrix of the type *'b mod-ring mat*. We can then apply the linear bound method directly, resulting in three goals. The first two goals are straightforward using lemmas on the *lift-01-mat* definition and known facts on the incidence matrix N given the locale context. The proof of the final goal closely follows the mathematical proof outlined earlier. The details omitted from the sketch above include manipulating a summation and utilising the formal results on the scalar product of two incidence vectors.

6.4 Generalised Fisher's Inequality

The generalised version of Fisher's inequality was first proved by Majumdar [20], after work by Bose, de Bruijn and Erdős on Fisher's initial statement. It was Bose [4] who first presented the linear algebraic approach to prove a more general version of the theorem, which Majumdar then built on. Notably, the generalised version of Fisher's inequality is presented on families of intersecting sets, rather than using design theoretic terminology.

▶ **Theorem 8.** *Let $A_1, ..., A_m$ be distinct subsets of $1, ..., n$ such that $|A_i \cap A_j| = k$ for some fixed $1 \leq k \leq n$ and every $i \neq j$. Then $m \leq n$.*

This is a familiar statement in comparison to both earlier variations, clearly putting a bound on any incidence set system with the constant intersect property. It is both more general than the uniform version of Fisher's, instead only constraining the intersect number rather than the points index, and has no condition on the size of blocks as in the odd town statement, while the intersect condition is stronger.

The proof we follow was presented by Jukna [17], building on initial work by Babai and Frankl [2]. On paper, the proof takes up less than half a page and is dominated by a linear independence proof. It utilises similar results on the scalar product of incidence vectors as in the odd town proof, however as we are no longer working in $(\mathbb{Z}/2\mathbb{Z})$, the calculations

require more creativity to prove that each coefficient must be equal to 0 in an arbitrary linear combination equal to the 0 vector. We begin by taking the scalar product of this linear combination with itself (rather than a singular incidence vector as in the odd town proof). Through manipulating the resultant summations, we are able to arrive at an equation where each term must clearly be positive. Using the intersection condition $|A_i| \geq k$ for all i and $|A_i| = k$ for at most one i, we are able to conclude that each coefficient must be 0.

Unlike the odd-town proof, we first prove trivial cases for when there are less than two blocks or a zero intersect number. However, we are similarly able to once again use the *lift-01-mat* definition to establish an incidence matrix with real elements *NR*. We proceed with the main part of the proof by applying the same linear bound argument.

As in the odd town case, the proofs of the first two properties are quick. While it would be possible to simplify these properties out entirely, it is useful to be able to reference these facts directly in the linear independence proof. Previously proven facts on the relationship between the scalar product, block size and intersection numbers can also be reused given lemmas on the preservation of incidence matrix properties under the *lift-01-mat* definition. The linear independence proof itself is more complicated than the odd town proof, having two main components. Firstly, we manipulate a summation into a split form through the initial scalar product step, resulting in the *finally* statement in the proof sketch below. We then reason on why the coefficients must evaluate to 0 using this split form, through the lemma *sum-split-coeffs-0*. Here, we can see the benefits of using the modified linear bound lemma, as summation reasoning for this and the odd town proof has minimal repetition. A sketch of the final proof in Isabelle, located in the *simp-ordered-const-intersect-design* locale, is given below:

```
theorem general-fishers-inequality: b ≤ v
proof (cases m = 0 ∨ b = 1)
  case True then show ?thesis ⟨proof⟩
next
  case False
  define NR :: real mat where NR ≡ lift-01-mat N
  show ?thesis proof (intro lin-bound-argument2[of NR])
    show distinct (cols NR) ⟨proof⟩
    show nrcm: NR ∈ carrier-mat v b ⟨proof⟩
    show ⋀f. vec v (λi. ∑j = 0..<b. f (col NR j) ∗ (col NR j) $ i) = 0_v v ⟹
     ∀v∈set (cols NR). f v = 0
    proof (intro ballI)
      fix f v  assume vin: v ∈ set (cols NR)
      assume eq0: vec v (λi. ∑j = 0..<b. f (col NR j) ∗ col NR j $ i) = 0_v v
      define c where c ≡ (λ j. f (col NR j))
      ⟨Isar linear independence proof details omitted⟩
      finally have sum-rep: 0 = (∑ j ∈ {0..<b} . (c j)^2 ∗ ((card (ℬs ! j))− (int m))) +
        m ∗ ((∑ j ∈ {0..<b} . c j)^2) by (simp add: algebra-simps)
      thus f v = 0 using sum-split-coeffs-0[of j′ c] ⟨proof⟩
    qed
  qed
qed
```

6.5 Dual of Generalised Fisher's

A brief analysis of the uniform version of Fisher's demonstrates that it is in fact a restricted version of the dual of the generalised version, with the inequality flipped. In design theory, it is significantly more common to reason about PBDs than it is to reason about designs with constant intersect numbers. Using the dual design formalisation outlined in Section (3.6), we've previously proven that the dual of a PBD with v points and b blocks is a constant intersect design with b points and v blocks. With the added condition of incompleteness, the dual is a simple constant intersect design, which is the context we proved the generalised version of Fisher's inequality in. Using locale interpretation, it is simple to apply Fisher's inequality to get the dual bound within the pairwise balanced locale, as desired:

corollary general-nonuniform-fishers:
assumes $\Lambda > 0$ **and** $\bigwedge$ bl. bl $\in\#$ $\mathcal{B} \Longrightarrow$ incomplete-block bl
shows v $\leq$ b

Note that this technically implies the uniform version of Fisher's on BIBDs which we started with. However, the rank argument remains a valuable tool, and hence we include both versions of the proof in our final formalisation.

7 Discussion

There are several valuable discussion points from this formalisation, of which we focus on two areas. Firstly, the challenges in formalising proofs utilising techniques from multiple mathematical fields and past formalisations, and secondly a comparison of the linear algebraic and traditional combinatorial proof approaches in a formal environment.

7.1 Integrating Multiple Formal Libraries

Modern mathematics routinely deals with proofs which draw on unexpected techniques from other mathematical fields. This is particularly true of combinatorics, as discussed in Section (1), where a barrier to formalisation in the past has been the need for established libraries in other fields of mathematics first. As formal libraries expand, it is anticipated it will routinely become more common to draw on formal results from a variety of fields and past formalisations. This formalisation was at the intersection of just two fields, using straightforward results from linear algebra as applied to combinatorics. Yet it highlights some of the barriers that still exist, particularly in older well established proof assistants such as Isabelle which are both benefited and hindered by decades of contributions. We summarise the challenges as follows:

(i) Managing multiple representations of the same concept in established libraries, and the transfer of results between these representations.
(ii) The flexibility of these representations for extensions and future applications.
(iii) Finding the necessary results as needed, including often routine lemmas.

Point (i) is likely the most relevant factor from a mathematician's perspective. In this formalisation matrices, vectors, vector spaces, and prime fields are notable examples. For a less experienced user, one would imagine that even the initial choice between representations may prove a barrier to proving results utilising these theories. In Isabelle, a survey of the AFP shows the JNF library [25] is quickly becoming the matrix library of choice for future formalisations, and further documentation summarising the current usage of certain libraries would likely prove useful. The transfer of results between representations remains

crucial, both for existing libraries, and in the case of the creation of a new, more effective representations for a structure. The lifting and transfer libraries [16] are a powerful tool here, and are well used in the matrix and vector libraries, however can be fiddly to set up. This formalisation experimented with using transfer rules to move between integer matrices under modular arithmetic and matrices of type *'a mod-ring*, a type originally defined using lift definitions [8]. Ultimately, a different approach was used and it was simpler to just prove the few equivalence results required directly.

The flexibility of past formalisations is essential to reducing duplication. In Isabelle, locales have proven to be valuable towards these efforts. This was again demonstrated in this formalisation, by the ease at which our existing incidence system locales from past work [9] could be extended and adapted. Similarly, they enable the transfer of results between vector space definitions. However, one notable limitation of locales in this formalisation was the use of definitions outside of the locale context. Inherited definitions could not be accessed directly, requiring an interpretation of the locale, while others could be such as *vec-space.rank*. To overcome this in our methodology, we defined a trivial equivalent definition, which ideally would not be needed.

Lastly (iii) addresses one of the most significant challenges of this formalisation, which was both determining the existence of and sourcing, sometimes trivial, results. In particular, for linear algebra in Isabelle, results are scattered across many AFP entries. Numerous entries which build on the JNF Matrix representation include a theory with a significant number of basic but useful results on vectors and matrices, but these can be difficult to find. This also means AFP entries have a growing number of dependencies and this formalisation will likely add yet another entry to the convoluted network of linear algebra results. Search tools aiming to answer this research question such as SErAPIS proved extremely useful in navigating these libraries when beginning the formalisation project.

7.2 Formalising the Linear Algebraic Method

The linear algebraic method is ultimately a very simple concept, but one that can be quite difficult to apply. Here we discuss how working with linear algebra in a formal environment compares to a traditional combinatorial approach and the joint focus on both proof techniques and theorems.

Firstly, working with matrix and vector representations of incidence systems had both advantages and challenges from a formalisation perspective. The main challenge lay in the considerable setup for reasoning to be transferable between the set and matrix representations. Of the theories in the final library, the incidence matrix theory by far has the most lines of code. These basic results were usually obvious, but could be tricky to prove. An initial review of the formalisation found multiple lemmas with similar statements, however the slight variations on these statements often proved useful from a proof automation standpoint in different contexts. Proof automation, specifically sledgehammer, also proved to be much less effective on matrix and vector representations than set theoretic definitions, with factors such as a much larger search space of dependencies and added assumptions on valid indexes possibly contributing to this.

Intuitive facts around linear algebraic representations, which often are what is appealing about a linear algebra proof on paper, did not always translate simply into formalisation. For example, it was common for proofs to use simple statements such as "the number of ones in an incidence vector is the size of the block it represents". In order to reason on these facts, we needed to lift a number of basic definitions of concepts already defined on lists and sets, such as counting the occurrences of a number or performing a summation over a

vector. Formalising the properties of an incidence system using a matrix based definition did require extra effort in equivalence proofs, but also significantly cleaned up formal reasoning on these properties in more complex proofs at later stages. With this setup now in place, linear algebraic proofs could also potentially offer a number of benefits for formalisation. Traditional counting proofs can be very intuitive, and hard to formalise, whereas a proof of linear independence for example tends to be much more calculation based, which is easier to formalise. Additionally, linear algebraic techniques do benefit from much more significant and established libraries.

The joint focus on proof techniques and their application to formalising particular theorems presents several advantages demonstrated in this paper. The techniques in this formalisation are conceptually simple, and by providing a number of general lemmas we aim to enable formalisations using these techniques without the need to gain a deeper understanding of the complexities of the Isabelle linear algebra libraries. While there are limitations on how general these techniques can be, when applicable they provide an easy way to structure proofs of a similar nature, and reduce duplication of work, as demonstrated by their application to formalising Fisher's inequality. This application was essential in turn to refine the formal techniques, demonstrating the benefit of dually focussing on both aspects during the formalisation process. Having now established these general lemmas, it would be interesting to further experiment with applying them to other theorems in extremal combinatorics such as the well known Frankl-Wilson theorem [11]. Chapter 13 in Jukna's textbook provides inspiration for other applications of the linear algebra bound method across several different types of combinatorial structures [17].

8 Conclusion

In summary, this paper presented three primary contributions to the formalisation of combinatorics. Firstly, the formalisation of incidence matrices, providing a linear algebraic representation and basic reasoning techniques for future formal proofs on incidence systems. Secondly, we formalise general lemmas for the rank argument and linear algebra method as used in combinatorial settings, aiming to improve accessibility for utilising these techniques in a formal environment. Finally, these techniques were demonstrated through the first formalisations of a number of variations of Fisher's inequality, a consequential theorem for both set systems and the linear algebraic method in combinatorics more broadly. The final formalisation repository is available online, and will be submitted to the Isabelle AFP.

For future work, it would be interesting to extend these techniques to more advanced settings, formalise other linear algebraic techniques and types of linear algebraic representations, and explore the potential for automation of some of the techniques involved. Lastly, Fisher's inequality itself is an important result which has several further extensions as well as applications to other interesting theorems yet to be formalised.

References

1 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*, volume 8901, pages 1–18. Springer International Publishing, Cham, 2014.

2 László Babai and Péter Frankl. *Linear Algebra Methods in Combinatorics*. Department of Computer Science, University of Chicago, 2.1 edition, 2020. URL: `https://people.cs.uchicago.edu/~laci/CLASS/HANDOUTS-COMB/BaFrNew.pdf`.

3 Clemens Ballarin. Locales and Locale Expressions in Isabelle/Isar. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs*, Lecture Notes in Computer Science, pages 34–50, Berlin, Heidelberg, 2004. Springer.

4 R. C. Bose. A Note on Fisher's Inequality for Balanced Incomplete Block Designs. *The Annals of Mathematical Statistics*, 20(4):619–620, December 1949.

5 Richard A. Brualdi and Herbert J. Ryser. *Combinatorial Matrix Theory*. Encyclopedia of Mathematics and Its Applications. Cambridge University Press, Cambridge, 1991.

6 Boris Bukh. Lecture notes in algebraic Methods in Combinatorics: Rank argument, 2014. URL: `http://www.borisbukh.org/AlgMethods14/rank_notes.pdf`.

7 C. J Colbourn and Jeffrey H. Dinitz. *Handbook of Combinatorial Designs / Edited by Charles J. Colbourn, Jeffrey H. Dinitz*. Discrete Mathematics and Its Applications. Chapman & Hall/CRC, Boca Raton, Fla. ; London, 2nd ed. edition, 2007.

8 Jose Divasón, Sebastiaan J. C. Joosten, René Thiemann, and Akihisa Yamada. A Verified Implementation of the Berlekamp–Zassenhaus Factorization Algorithm. *Journal of Automated Reasoning*, 64(4):699–735, April 2020.

9 Chelsea Edmonds and Lawrence C. Paulson. A Modular First Formalisation of Combinatorial Design Theory. In *CICM*, 2021.

10 R. A. Fisher. An Examination of the Different Possible Solutions of a Problem in Incomplete Blocks. *Annals of Eugenics*, 10(1):52–75, 1940.

11 P. Frankl and R. M. Wilson. Intersection theorems with geometric consequences. *Combinatorica*, 1(4):357–368, December 1981.

12 C. D. Godsil. Tools from Linear Algebra. In Lovász L Graham RL, Grötschel M, editor, *Handbook of Combinatorics*, volume 2. Elsevier, Amsterdam, 1996.

13 Georges Gonthier. The Four Colour Theorem: Engineering of a Formal Proof. In Deepak Kapur, editor, *Computer Mathematics*, Lecture Notes in Computer Science, pages 333–333, Berlin, Heidelberg, 2008. Springer.

14 W. T. Gowers. Dimension arguments in combinatorics, July 2008. URL: `https://gowers.wordpress.com/2008/07/31/dimension-arguments-in-combinatorics/`.

15 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`.

16 Brian Huffman and Ondřej Kunčar. Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Georges Gonthier, and Michael Norrish, editors, *Certified Programs and Proofs*, volume 8307, pages 131–146. Springer International Publishing, Cham, 2013.

17 Stasys Jukna. *Extremal Combinatorics*. Texts in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

18 Jonas Keinholz. Matroids. *Isabelle Archive of Formal Proofs*, November 2018. URL: `https://www.isa-afp.org/entries/Matroids.html`.

19 Daniel Kroes, Jacob Naranjo, Jiaxi Nie, Jason O'Neill, Nicholas Sieger, Sam Sprio, and Emily Zhu. Lecture notes: Linear Algebra methods in Combinatorics, 2019. URL: `https://mathweb.ucsd.edu/~sspiro/Abacus/AbacusF19.pdf`.

20 Kulendra N. Majumdar. On some Theorems in Combinatorics Relating to Incomplete Block Designs. *The Annals of Mathematical Statistics*, 24(3):377–389, September 1953.

21 Lars Noschinski. A Graph Library for Isabelle. *Mathematics in Computer Science*, 9(1):23–39, March 2015.

22 Steven Obua and Technische Universität München. Proving bounds for real linear programs in isabelle/HOL. In *Theorem Proving in Higher Order Logics (TPHOLs 2005), Volume 3603 of Lect. Notes in Comp. Sci*, pages 227–244. Springer, 2005.

23 Leonard H. Soicher. Design GAP Manual. URL: `https://www.gap-system.org/Manuals/pkg/design-1.7/doc/manual.pdf`.

24 Douglas Stinson. *Combinatorial Designs: Constructions and Analysis*. Springer-Verlag, New York, 2004.

25 René Thiemann and Akihisa Yamada. Formalizing Jordan normal forms in Isabelle/HOL. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, pages 88–99, St. Petersburg, FL, USA, January 2016. Association for Computing Machinery.

26 Markus Wenzel. *Isabelle, Isar – A Versatile Environment for Human Readable Formal Proof Documents*. PhD thesis, Technical University Munich, Germany, 2002. URL: `http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.pdf`.

Synthetic Kolmogorov Complexity in Coq

Yannick Forster
Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
Inria, Gallinette Project-Team, Nantes, France

Fabian Kunze
Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

Nils Lauermann
Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
University of Cambridge, Cambridge, United Kingdom

Abstract

We present a generalised, constructive, and machine-checked approach to Kolmogorov complexity in the constructive type theory underlying the Coq proof assistant. By proving that nonrandom numbers form a simple predicate, we obtain elegant proofs of undecidability for random and nonrandom numbers and a proof of uncomputability of Kolmogorov complexity.

We use a general and abstract definition of Kolmogorov complexity and subsequently instantiate it to several definitions frequently found in the literature.

Whereas textbook treatments of Kolmogorov complexity usually rely heavily on classical logic and the axiom of choice, we put emphasis on the constructiveness of all our arguments, however without blurring their essence. We first give a high-level proof idea using classical logic, which can be formalised with Markov's principle via folklore techniques we subsequently explain. Lastly, we show a strategy how to eliminate Markov's principle from a certain class of computability proofs, rendering all our results fully constructive.

All our results are machine-checked by the Coq proof assistant, which is enabled by using a synthetic approach to computability: rather than formalising a model of computation, which is well known to introduce a considerable overhead, we abstractly assume a universal function, allowing the proofs to focus on the mathematical essence.

2012 ACM Subject Classification Theory of computation → Constructive mathematics; Theory of computation → Type theory

Keywords and phrases Kolmogorov complexity, computability theory, random numbers, constructive matemathics, synthetic computability theory, constructive type theory, Coq

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.12

Supplementary Material *Software (Source Code)*:
`https://github.com/uds-psl/coq-synthetic-computability/tree/kolmogorov`
archived at `swh:1:dir:172c753027deabef87ce303e73dc98fc14745a89`

Funding *Yannick Forster*: received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101024493.

Acknowledgements We want to thank Dominik Kirst and Gert Smolka for many fruitful discussions.

1 Introduction

Kolmogorov complexity[1] as a subject of computability and information theory is usually presented in the abstract rather than using spelled-out definitions (e.g. [42, 30]). This is no oversight: Kolmogorov complexity is a general concept, invariant in the concrete definition as long as some structural properties are fulfilled. The Kolmogorov complexity of a number x is the length of the shortest bitstring s s.t. decoding s terminates with x, based on a certain decoding function called *description mode* (which can and will be left abstract at first).

[1] The origins of the complexity notion are complicated, see Li and Vitányi's book for an overview [30, §1.13]. It is often also called Solomonoff-Kolmogorov-Chaitin complexity, crediting the influential work by Ray Solmonoff [44, 45, 46], Andrej Kolmogorov [24, 25], and Gregory Chaitin [6]

In terms of (the abstraction away from) details, presentations of Kolmogorov complexity are well in line with presentation of other areas of computability theory, which are also usually given conceptually rather than with spelled out definitions. This is due to extensive use of the Church-Turing thesis, stating that every intuitively computable function is in fact computable in a fixed (and thus any) Turing-complete model of computation. Thus, presentations of computability theory can abstract away from the chosen model, and definitions could be but are not spelled out using many different models like the λ-calculus, Turing machines, μ-recursive functions, counter machines, or more elaborate models like `while`-based simply typed programming languages.

Presentations of Kolmogorov complexity take generality one step further: Not only could they be instantiated to various models of computation, even the concrete definition of a description mode can be left abstract and instantiated to multiple possible definitions per model of computation.

Recently, one such particular definition for the model of the full λ-calculus was given by Catt and Norrish [5] in HOL4. They prove various inequalities w.r.t. Kolmogorov complexity and that Kolmogorov complexity is not a computable function. Even the formal definition of Kolmogorov complexity in their setting is non-trivial and requires both classical logic (i.e. a variant of the law of excluded middle) and a unique choice operator. Subsequently, various constructions on the λ-calculus become relevant, which they acknowledge to be tedious.

This tedium is well known amongst different machine-checked proofs in computability theory: Results like Rice's theorem [39] or the existence of universal machines are feasible to machine check, but more involved results like the Myhill isomorphism theorem [32] (one-one equivalence gives rise to a recursive isomorphism), the existence of Post's simple and hypersimple sets [37] (there are enumerable, undecidable many-one/truth-table degrees different from the halting problem), the Kleene-Post theorem [22] (stating that there are incomparable Turing degrees), solutions to Post's problem [31, 18] (there is an enumerable, undecidable Turing degree different from the halting problem) or Post's theorem [38] (relating Turing jumps and the arithmetical hierarchy[2]) are out of reach for direct translations of the textbook setting based on models of computation to proof assistants: Dealing with concrete programs in low-level models of computations outweighs the (also involved) mathematical arguments underlying the proofs to a level that mechanisation becomes infeasible.

To circumvent the tedium of models of computation for machine-checked proofs, *synthetic* approaches have proved successful: Synthetic undecidability has allowed machine-checked undecidability proofs for various landmark results [15, 28, 21, 9, 8], based on the observation that a problem is undecidable if an assumed Coq function deciding the problem could be used to construct e.g. a decider of the Turing machine halting problem. For abstract results in computability, Forster [13] presents an axiomatic setting allowing to prove Rice's theorem. Forster, Smolka, and Jahn [14] work in this setting and construct simple and hypersimple sets, allowing to prove that one-one, many-one, and truth-table reducibility all differ.

In this setting, based on the axiom *Synthetic Church's Thesis*, one assumes a function $\phi\colon \mathbb{N} \to \mathbb{N} \rightharpoonup \mathbb{N}$, where a call $\phi_c x$ morally represents the c-th program in a fixed model of computation evaluated on input x, but the definition of ϕ is not explicitly given, and a *universality property* of ϕ w.r.t. all functions of type $\mathbb{N} \rightharpoonup \mathbb{N}$, i.e. that $\forall f\colon \mathbb{N} \rightharpoonup \mathbb{N}.\exists c.\forall x.\phi_c x \equiv fx$, and an S_n^m operator for ϕ like in the S_n^m theorem.

The observation that computability theory can be developed in such a setting goes back to Richman and Bridges [40, 3], who develop synthetic computable analysis in Bishop-style constructive logic, and has also been employed by Bauer [1, 2] in the internal logic of

[2] The given reference is usually used for Post's theorem, but just constitutes a short abstract containing neither the theorem or a proof. Standard textbook references give modern proofs [41, 34].

the effective topos. In contrast to the work of Richman, Bridges, and Bauer, developing synthetic computability in the constructive type theory underlying the Coq proof assistant is compatible with classical logic, i.e. the law of excluded middle can be assumed alongside Synthetic Church's Thesis without immediately introducing an inconsistency.

Synthetic computability in the Calculus of Inductive Constructions (CIC [7, 35, 36]), the constructive type theory underlying the Coq proof assistant [47], is based on two facts: First, that all functions definable in CIC can be shown computable in CIC (thus allowing the assumption of a universal function, implementable as the universal μ-recursive function). Secondly, that CIC has a strict separation between function spaces such as $\mathbb{N} \to \mathbb{N}$ or $\mathbb{N} \rightharpoonup \mathbb{N}$ and logic, carried out in the impredicative universe of propositions $\mathbb{P}$. Thus, assuming the law of excluded middle (LEM) does not leak into the function space and does not allow the definition of an uncomputable function. In this setting, one has the choice between three equivalent axioms to develop results: SCT, based on total functions $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$, EPF, based on partial functions $\mathbb{N} \to \mathbb{N} \rightharpoonup \mathbb{N}$, and EA, based on parametrically enumerable predicates $\mathbb{N} \to \mathbb{N} \to \mathbb{P}$. Forster, Jahn, and Smolka [14] use EA to construct Post's simple predicate.

Coming back to Kolmogorov complexity, the noncomputability of Kolmogorov complexity can also be proved by showing that nonrandom numbers (numbers which can be efficiently described, i.e. which are longer than their Kolmogorov complexity) form a simple predicate. We give a machine-checked proof of this result using the definition of simple predicates by Forster, Jahn, and Smolka [14] based on the axiom EPF [13], which seems most natural for Kolmogorov complexity due to the frequent use of partial functions.

We structure the paper to be instructive for similar future endeavours: First, we give the high-level proof idea in Section 2, without formal definitions and without bothering about the use of classical logic. Then, we introduce the relevant aspects of CIC in Section 3: Total and partial functions (treated as computable functions), partial relations (which are not necessarily computable), total relations (also not provably computable, but where a constructive totality proof is possible if and only if the relation is computable), and classically total relations (which can be proved total using classical logic and are not necessarily computable).

We recap the formalities of synthetic computability in CIC in Section 4 and discuss provable versions of choice principles in Section 5. We then observe that Kolmogorov complexity has to be formalised as a partial, classically total relation and introduce the notion formally in Section 6. In Section 7 we introduce nonrandom and random numbers, suitably formalised to allow a constructive enumerability proof, and show that the nonrandom numbers form a simple predicate. For infinity of random numbers, we rely on the definition by Forster, Jahn, Smolka [14] and explain how to insert double negations in the formalised statements of the intuitive proofs to get rid of uses of the law of excluded middle. To show that nonrandom numbers are immune (a necessary condition to show nonrandom numbers simple) we rely on Markov's principle (MP), which simplifies the proof.

We then discuss the role of MP in Section 8 and show a general technique how the specific way how MP is used in the proof, which also occurs in other proofs, can be eliminated to obtain a fully constructive proof that nonrandom numbers form a simple predicate.

In Section 9 we then instantiate our general definition of Kolmogorov complexity to the definition used by Catt and Norrish (while still staying in the synthetic setting). We also instantiate to a definition similar to the definition used in Odifreddi's book [34] in Section 10.

We claim that we obtain elegant proofs of all results and that our proofs do not lose elegance either due to formalising, due to mechanising them in a proof assistant, or due to presenting them in constructive logic. The machine-checked development[3] is accessible to any Coq user and no knowledge of tedious details regarding models of computation is necessary.

[3] The development is hyperlinked with the present paper, always indicated by a clickable -symbol.

2 Intuition: Nonrandom numbers form a simple predicate

The Kolmogorov complexity $\mathcal{C}(x)$ of a number x is the size of its shortest description. We call a partial function $\delta : \mathbb{N} \rightharpoonup \mathbb{N}$ a *description mode*. We call y a description of x if $\delta y \triangleright x$.

The size of descriptions is the length of its interpretation as bitstring. We write $|n|$ for the length of the bit string representing n. We will assume that the interpretation as bitstring works such that the size of n is at most as long as the logarithm (by 2) of n plus 1.

As a consequence, we obtain the following.

▶ **Fact 1.** *For all d, there is n with $n > |n| + d$.*

▶ **Fact 2.** *There are exactly 2^k descriptions with size k and, $2^k - 1$ descriptions with size less than k.*

Proof. The first follows since there are exactly 2^k bit strings of length k. The second is by usual arithmetical arguments. ◀

We only assume one property of δ, namely that it is optimal up to a constant. I.e. for every computable function δ' there is a constant d s.t.

$$\forall y' x.\ \delta' y' \triangleright x \to \exists y.\ \delta y \triangleright x \wedge |y| < |y'| + d.$$

Using the optimality on $\delta' y := y$ we obtain that δ is surjective:

▶ **Fact 3.** *Every number has a description.*

A number is nonrandom if its shortest description is shorter than the number itself. Formally, $\mathcal{N}(x) := \mathcal{C}(x) < x$. Intuitively (but because we do not go into detail how δ is defined, not formally), the following number is nonrandom:

001 001 001 001 001 001 001 001 001 001 001 001 001 001 001 001 001

This is because it can be written more compactly as "repeat 001 for 17 times". On the other hand, the terminology random seems intuitive when looking at a number like

833 256 052 501 225 207 711 246 773 401 191 432 051 794 357 674 718

We now prove that nonrandom numbers form a simple predicate, a result that first appeared in [50] and is there attributed to Janis Barzdins. A predicate is simple if it is enumerable and its complement is infinite but does not contain an enumerable, infinite subpredicate.

▶ **Lemma 4.** *Nonrandom numbers are enumerable.*

Proof. Try all possible inputs y and evaluate δy for all possible step numbers n. If δy evaluates to a value x in n steps and $|y| < |x|$, then output x. ◀

▶ **Lemma 5.** *Random numbers are infinite.*

Proof. Suppose all 2^k many numbers x_i of size k would be nonrandom. This means that they would have 2^k many distinct descriptions y_i, all with size smaller than k. But there are at most $2^k - 1$ numbers of size smaller than k. Contradiction. ◀

The third part of the definition of simpleness, which is often called immunity, is easiest to explain with some more formulas.

▶ **Lemma 6.** *There is no enumerable infinite sub-predicate of random numbers.*

Proof. Assume such a predicate q. Since q is infinite, in particular, for every k it contains x with $|x| > k$. Now since q is enumerable, we can define a computable function f computing these x, i.e. for all k, fk is in q. Since q is a subpredicate of random numbers, in particular fk is random.

$$\forall k.\ k < |fk|.$$

Since δ is optimal, we have a constant d and descriptions y_k for every fk at most as long as $|k| + d$, i.e.:

$$\forall k.\ |y_k| \leq |k| + d.$$

Since fk is random, its size is shorter than its shortest description, and in particular it is shorter than its description y_k, i.e.

$$\forall k.\ |fk| \leq |y_k|.$$

But now $\forall k.\ k < |fk| \leq |y_k| \leq |k| + d$, a contradiction to Fact 1. ◀

▶ **Corollary 7.**

1. *Nonrandom numbers are simple.*
2. *Both random and nonrandom numbers are undecidable.*
3. *Kolmogorov complexity is noncomputable.*

Proof. Simple predicates can be shown undecidable [14]. If there would be a function f computing Kolmogorov complexity, $\lambda x.fx < |x|$ would decide nonrandomness. ◀

3 Constructive Logic in Coq's Type Theory

In the previous section, we informally defined Kolmogorov complexity as the size of the *least* description. Since every number has a description, Kolmogorov complexity was treated as a total function. Later, we showed that this total function is however not computable.

When we now want to define Kolmogorov complexity in CIC, we cannot define it as a function, because all definable functions in CIC always are, by definition, computable. Thus, we will define Kolmogorov complexity as a functional relation. While in set theory the notion of a function and a functional relation are conflated they are strictly separate in CIC.

We make use of this separation and introduce the following general leastness-predicate transformer: For every predicate $p : \mathbb{N} \to \mathbb{P}$ one can define the predicate $y\ is\ (\boldsymbol{\mu} x.\ px)$ being satisfied exactly if y is the least number satisfying p.

$$y\ is\ (\boldsymbol{\mu} x.\ px) := py \wedge \forall y'.\ py' \to y \leq y'$$

We can then prove that the predicate is functional.

Fact 8. $y_1\ is\ (\boldsymbol{\mu} x.\ px) \to y_2\ is\ (\boldsymbol{\mu} x.\ px) \to y_1 = y_2$

As we have defined it, we cannot prove that whenever p is satisfied there is a least element satisfying it. To see this, we can look again at Kolmogorov complexity: After defining Kolmogorov complexity using $\boldsymbol{\mu}$, we could then show that Kolmogorov complexity is a total relation. This, however, is impossible: A constructive totality proof $\forall x.\exists y.$ would correspond via the Curry-Howard interpretation to a definable function computing Kolmogorov complexity, which we know cannot exist because Kolmogorov complexity is uncomputable.

Since we know that a totality proof cannot be constructive, we will use classical totality: A relation $X \to Y \to \mathbb{P}$ is classically total if $\forall x.\neg\neg\exists y.\ Rxy$.

When proving the $\neg\neg\exists$ part of classical totality, one can use classical logic. This is because one can think of double negation as a modality with the following proof rules.

Fact 9. *The following hold:*

1. $P \to \neg\neg P$
2. $\neg\neg P \to (P \to \neg\neg Q) \to \neg\neg Q$
3. $(P \vee \neg P \to \neg\neg Q) \to \neg\neg Q$

Item 1 states that the double negation modality can be left at any time (it has the type of the return of a monad). 2 states how to combine different double negated proofs (it has the type of the bind of a monad). 3 states that in the double negation modality, arbitrary case disctinctions are allowed.

We can indeed prove that whenever p is satisfied it does not not have a least element:

Fact 10. $py \to \neg\neg\exists y.\ y\ is\ (\boldsymbol{\mu} x.\ px)$

The well-known axioms of the law of excluded middle LEM and Markov's principle MP can be thought of as extending the proof rules for the double negation modality.

$$\mathsf{LEM} := \forall P{:}\mathbb{P}.\ P \vee \neg P \qquad \mathsf{MP} := \forall f{:}\mathbb{N} \to \mathbb{B}.\ \neg\neg(\exists n.\ fn = \mathsf{true}) \to \exists n.\ fn = \mathsf{true}$$

MP allows entering the double negation modality whenever the goal is a Σ^0_1 formula. The following states that LEM allows entering the double negation modality for every goal.

Fact 11. $\mathsf{LEM} \to \neg\neg Q \to Q$

Using LEM, one can prove that least elements for inhabited predicates always exist. Furthermore, in constructive mathematics, it is standard practice to also analyse whether made classical assumptions are necessary, and in this case one can indeed prove that the existence of least elements for inhabited predicates implies LEM.

Lastly, we need a good, constructive definition of infinite predicates. The definition cannot be too weak, which might mean that we cannot show that simple predicates are undecidable. The definition cannot be too strong either, which might mean that it is impossible to establish random numbers to be infinite. We can here just follow Forster, Jahn, and Smolka [14] in using non-finite predicates. We use the type of lists over X, written $\mathbb{L}X$, with [] being the empty list and $x :: l$ being the list with element x added to a list l. A predicate $p{:}X \to \mathbb{P}$ is *finite* if $\exists l{:}\mathbb{L}X.\ \forall x.\ px \leftrightarrow x \in l$. A predicate $p{:}X \to \mathbb{P}$ is *subfinite* if $\exists l{:}\mathbb{L}X.\ \forall x.\ px \to x \in l$. A predicate is *non-finite* if it is not finite.

Fact 12. *A predicate is non-finite if and only if it is not subfinite.*

Non-finiteness for predicates on natural numbers can be expressed as a $\forall\neg\neg\exists$ version of having elements of arbitrary size.

Fact 13. *Let $f{:}\mathbb{N} \to \mathbb{N}$ be a size function which assigns only finitely many elements the same size.*

A predicate $p : \mathbb{N} \to \mathbb{P}$ is infinite if and only if

$$\forall n.\ \exists x.\ fx \geq n \wedge px.$$

4 Synthetic Computability Theory

Computability theory is usually presented with respect to an effective enumeration ϕ of all computable function, i.e. ϕ_c denotes the c-th computable function, and for every computable function f there is a code c s.t. f and ϕ_c agree. By appeal to the (informal) Church Turing thesis, ϕ has the following informal universal property:

$$\forall f : \mathbb{N} \rightharpoonup \mathbb{N}.\ f \text{ is intuitively computable} \to \exists c. \forall x v.\ \phi_c x \rhd v \leftrightarrow f x \rhd v.$$

In synthetic computability theory [13], one uses essentially the same approach, but (1) abstracts against an abstract function ϕ rather than a concrete enumeration, and (2) assumes all functions $\mathbb{N} \rightharpoonup \mathbb{N}$ of the meta-theory to be intuitively computable. Thus, we obtain an axiom assuming a function $\phi : \mathbb{N} \to \mathbb{N} \rightharpoonup \mathbb{N}$ with the following formal universality property:

$$\forall f : \mathbb{N} \rightharpoonup \mathbb{N}.\ \exists c. \forall x v.\ \phi_c x \rhd v \leftrightarrow f x \rhd v$$

This axiom is called EPF in [11] and is a consequence of the axiom EPF in [13].[4] Consistency of such an axiom is a consequence of the consistency of the axiom CT in CIC, see [13, Appendix A] and [11].

The first simplification entailed by synthetic computability becomes obvious when defining standard notions of computability, which do not need to rely on a model of computation [15]. A predicate $p : X \to \mathbb{P}$ is

- *decidable* if there exists a decider: $\exists f : X \to \mathbb{B}. \forall x.\ px \leftrightarrow fx = \mathsf{true}$
- *enumerable* if there exists an enumerator: $\exists f : \mathbb{N} \to \mathbb{O}X. \forall x.\ px \leftrightarrow \exists n.\ fn = \mathsf{Some}\, x$

Here, $\mathbb{O}X$ is the type with constructors None and Some x where $x : X$ and $\mathbb{B}$ has elements true and false.

The intuition behind decidability should be immediately clear. Enumerability captures the definition that a predicate is "r.e." ("recursively enumerable") if it is the co-domain of a function. Equivalent domain-based definitions can be given [12] but we do not require them.

If a predicate p is enumerable we write $\mathcal{E}p$. We write $\mathcal{E}R$ for a relation $R : \mathbb{N} \to \mathbb{N} \to \mathbb{P}$ if the predicate $\lambda(x, y).\ Rxy$ is enumerable. We call a type X *discrete* if $\lambda(x_1, x_1).\ x_1 = x_2$ is decidable and *enumerable* if $\lambda x : X.\ \top$ is enumerable.

One can give various definitions of partial functions in CIC with enumerable graph [12, §4.5]. We abstractly use a function type $\mathbb{N} \rightharpoonup \mathbb{N}$ and write $fx \rhd y$ if $f : \mathbb{N} \rightharpoonup \mathbb{N}$ on input x evaluates to y. This type can for instance be inhabited by stationary functions $\mathbb{N} \to (\mathbb{N} \to \mathbb{O}\mathbb{N})$, i.e. where $fxn_1 = \mathsf{Some}\, x \to \forall n_2 \geq n_1.\ fxn_2 = \mathsf{Some}\, x$. We just need the following fact and only need more constructions for partial functions in Fact 37.

Fact 14. *One can define an enumeration function* graph : $(\mathbb{N} \rightharpoonup \mathbb{N}) \to \mathbb{N} \to \mathbb{O}(\mathbb{N} \times \mathbb{N})$ *s.t.*

$$\forall f x y.\ fx \rhd y \leftrightarrow \exists n.\ \mathsf{graph}\, f n = \mathsf{Some}\, (x, y).$$

Following Forster, Jahn, and Smolka [14] one can also define reducibility, e.g. many-one, one-one, or truth-table reducibility. We refrain from doing so here and just use their definition of simpleness. A predicate $p : \mathbb{N} \to \mathbb{P}$ is *simple* if

1. p is enumerable,
2. the complement of p is not finite,
3. the complement of p has no non-finite, enumerable sub-predicate q (i.e. $\forall x.\ qx \to \neg px$).

[4] This is not a typo, both axioms are unfortunately indeed named the same.

5 Choice Functions

In computability theory, unbounded search is a crucial tool to define functions. In CIC, we have two options for unbounded search: First, we can implement an unbounded search operator yielding partial functions (e.g. of type $\mathbb{N} \rightharpoonup \mathbb{N}$). Secondly, we can make use of the fact that a choice principle for enumerable relations is provable. This means that for enumerable relations (often also called Σ_1^0-relations) one can extract a function from a totality proof.

Fact 15. *Every total enumerable relation $R : \mathbb{N} \to \mathbb{N} \to \mathbb{P}$ has a choice function:*

$$\mathcal{E}R \to (\forall x.\exists y.\ Rxy) \to \exists f.\forall x.\ Rx(fx)$$

Proof. Due to the fact that whenever $\exists n : \mathbb{N}.\ fn = \mathsf{true}$ also $\Sigma n : \mathbb{N}.\ fn = \mathsf{true}$, discovered independently by Benjamin Werner and Jean-François Monin and part of Coq's standard library as *constructive ground description* operator. ◀

As discussed it is however not always possible to prove totality of a relation without classical logic. For enumerable relations, MP suffices to obtain full classical power for the totality proof.

Fact 16. *Given* MP*, every classically total enumerable relation $R : \mathbb{N} \to \mathbb{N} \to \mathbb{P}$ is total:*

$$\mathsf{MP} \to \mathcal{E}R \to (\forall x.\neg\neg\exists y.\ Rxy) \to \forall x.\exists y.\ Rxy$$

Corollary 17. *Given* MP*, every classically total enumerable relation $R : \mathbb{N} \to \mathbb{N} \to \mathbb{P}$ has a choice function:*

$$\mathsf{MP} \to \mathcal{E}R \to (\forall x.\neg\neg\exists y.\ Rxy) \to \exists f.\forall x.\ Rx(fx)$$

6 Kolmogorov Complexity

Recall that the Kolmogorov complexity of a number x is the size of the least description of x. To formally capture the notion, we need to formally capture *size* and *description*.

We will model descriptions using the type of natural numbers. To define the size of a description we parameterise our development against a bijection between natural numbers and binary strings, i.e. against functions $\lceil\cdot\rceil : \mathbb{N} \to \mathbb{LB}$ and $\lfloor\cdot\rfloor : \mathbb{LB} \to \mathbb{N}$ s.t.

1. $\forall l.\ \lceil\lfloor l \rfloor\rceil = l$
2. $\forall n.\ \lfloor\lceil n \rceil\rfloor = n$
3. $\forall n.\ \mathsf{len}(\lceil n \rceil) \leq \mathsf{log}_2(n) + 1$
4. $\forall xy.\ x \leq y \to \mathsf{len}(\lceil x \rceil) \leq \mathsf{len}(\lceil y \rceil)$

We model binary strings formally as lists of booleans $\mathbb{LB}$, for instance $[]$, $[\mathsf{true}, \mathsf{false}]$, $[\mathsf{false}, \mathsf{false}, \mathsf{false}]$ are then binary strings. We write $\mathsf{len}(l)$ for the length of a binary string.

To see that such a bijection exists, we can define one which identifies a natural number n with the binary expansion of $n + 1$ with the most significant bit left out. I.e. this bijection associates the following numbers and lists:

$$0 \sim [] \qquad 1 \sim [\mathsf{false}] \qquad 2 \sim [\mathsf{true}] \qquad 3 \sim [\mathsf{false}, \mathsf{false}] \qquad 4 \sim [\mathsf{false}, \mathsf{true}] \qquad \ldots$$

It is then routine to prove the above properties, we do not go into detail here.

We now turn back to working with any bijection fulfilling the above four properties and define the *size* of a number n as the length of its interpretation as binary string, i.e.

$$|n| := \mathsf{len}(\lceil n \rceil).$$

We can use concatenation to define a pairing function on natural numbers:

$$\langle x, y \rangle := \lfloor \lceil x \rceil +\!\!+ \lceil y \rceil \rfloor$$

Note that $\langle x, y \rangle$ is surjective, but injectivity or the existence of a (left-)inverse function depend on the concrete definition of $\lceil \cdot \rceil$ and $\lfloor \cdot \rfloor$, because x and y are not necessarily reconstructible from $\lceil x \rceil +\!\!+ \lceil y \rceil$. We however do not rely on either.

Fact 18. $|\langle x, y \rangle| = |x| + |y|$

Since the size behaves logarithmically, there (classically) exist numbers n which are larger than $|n|$. Formally:

Fact 19. *For all d, $\neg\neg\exists n.\ n > |n| + d$. Equivalently: For all d, $\neg\forall n. n \leq |n| + d$.*

The following fact will become important later, because it restricts the amount of possible descriptions with a certain size.

Fact 20. *There are exactly 2^k distinct lists of booleans x with $\mathsf{len} x = k$:*

$$\exists l{:}\,\mathbb{L}(\mathbb{LB}).\ (\forall x.\ x \in l \leftrightarrow \mathsf{len}(x) = k) \wedge \#l \wedge \mathsf{len}(l) = 2^k$$

We write $\#l$ to indicate that l does not contain any duplicates. One can find various definitions of what constitutes a description of a number x. We will just assume a *description mode* $\delta{:}\,\mathbb{N} \rightharpoonup \mathbb{N}$. As in the intuitive explanation, we will assume that δ is optimal, i.e. that

$$\forall \delta'{:}\,\mathbb{N} \rightharpoonup \mathbb{N}.\exists d{:}\,\mathbb{N}.\forall y' x.\ \delta' y' \triangleright x \to \exists y.\ \delta y \triangleright x \wedge |y| < |y'| + d.$$

As before, using optimality on the identity function, we can obtain that every number has a description.

Fact 21. $\forall x.\exists y.\ \delta y \triangleright x$

In Sections 9 and 10 we give several possible definitions for description modes frequently found in the literature.

We define the Kolmogorov complexity $\mathcal{C}(x)$ of a number x to be the least number s s.t. s is a size of a description of x, i.e. we define a functional relation $\mathcal{C}{:}\,\mathbb{N} \to \mathbb{N} \to \mathbb{P}$ as follows

$$\mathcal{C}(x)s := s \mathit{\ is\ } \boldsymbol{\mu} s.\ \exists y.\ s = |y| \wedge \delta y \triangleright x$$

We also write $s \mathit{\ is\ } \mathcal{C}(x)$ for $\mathcal{C}(x)s$.

Surjectivity of δ suffices to prove that $\mathcal{C}(x)$ is classically total.

Fact 22. $\forall x.\ \neg\neg\exists s.\ s \mathit{\ is\ } \mathcal{C}(x)$

We can also use optimality of δ to prove that a definition of $\mathcal{C}$ using any other description operator δ' yields a similar definition of Kolmogorov complexity, only differing in a constant. In this so-called *invariance theorem* we write $\mathcal{C}_\delta$ and $\mathcal{C}_{\delta'}$ for the two respective definitions of Kolmogorov complexity.

Theorem 23. *If δ is an optimal description mode we have that for every $\delta'{:}\,\mathbb{N} \rightharpoonup \mathbb{N}$ there exists $d{:}\,\mathbb{N}$ s.t. whenever s is $\mathcal{C}_\delta(x)$ and s' is $\mathcal{C}_{\delta'}(x)$ we have $s \leq s' + d$.*

7 Nonrandom Numbers

In this section, we introduce nonrandom numbers and prove that they form a simple predicate. Recall that a number is nonrandom if it has a description shorter than its own size. Formally, we have three options to define nonrandom numbers:

$$\mathcal{N}_1(x) := \exists s.\ s\ is\, \mathcal{C}_\delta(x)\ \wedge s < |x|$$
$$\mathcal{N}_2(x) := \exists y.\ \delta y \triangleright x\qquad \wedge |y| < |x|$$
$$\mathcal{N}_3(x) := \forall s.\ s\ is\, \mathcal{C}_\delta(x) \to\ \ s < |x|$$

Note that we cannot simply use $\mathcal{C}_\delta(x) < |x|$, because $\mathcal{C}$ is not a total function. Options $\mathcal{N}_1$ and $\mathcal{N}_3$ capture the two possible ways to talk about the values of a functional relation. Option $\mathcal{N}_2$ circumvents mentioning $\mathcal{C}$, because if some description y has smaller size than x, then surely the smallest description (i.e. $\mathcal{C}_\delta(x)$) has smaller size as well. All definitions have different constructive strength, but are classically equivalent.

Fact 24. $\mathcal{N}_1(x) \to \mathcal{N}_2(x)$, $\mathcal{N}_2(x) \to \mathcal{N}_3(x)$, $\mathcal{N}_3(x) \to \neg\neg\mathcal{N}_1(x)$.

Proof. Relying on surjectivity of δ. ◀

Since only $\mathcal{N}_2$ can be proved enumerable constructively, we choose it as definition.

$$\mathcal{N}(x) := \exists y.\ \delta y \triangleright x \wedge |y| < |x|$$

Fact 25. *$\mathcal{N}$ is enumerable.*

Proof. Let $f : \mathbb{N} \to \mathbb{O}\mathbb{N}$ enumerate the graph of δ, i.e. $\delta y \triangleright x \leftrightarrow \exists n.\ fn = \mathsf{Some}\,(x, y)$. Then $\lambda n.$ **if** fn **is** $\mathsf{Some}\,(x', y)$ **then if** $x' = x \wedge |y| < |x|$ **then** $\mathsf{Some}\, y$ **else** None **else** None enumerates $\mathcal{N}$. ◀

The complement of nonrandom numbers is formed by *random* or *incompressible* numbers:

$$\mathcal{R}(x) := \forall y.\ \delta y \triangleright x \to\quad |x| \le |y|$$

We emphasise that, counter-intuitively perhaps given the naming, random numbers are constructively the complement of nonrandom numbers (rather than the other way around):

Fact 26. $\mathcal{R}(x) \leftrightarrow \neg\mathcal{N}(x)$.

Classically, a number is random if it is shorter than or as long as its shortest description, again for surjective description functions:

Fact 27. *Given $\forall x.\exists y.\ \delta y \triangleright x$ we have Let $\mathcal{R}_1(x) := \exists s.\ s\ is\, \mathcal{C}_\delta(x) \wedge |x| \le s$ and $\mathcal{R}_3(x) := \forall s.\ s\ is\, \mathcal{C}_\delta(x) \to |x| \le s$. Then $\mathcal{R}_1(x) \to \mathcal{R}(x)$, $\mathcal{R}(x) \to \mathcal{R}_3(x)$ and $\mathcal{R}_3(x) \to \neg\neg\mathcal{R}_1(x)$.*

We have already proved that $\mathcal{N}$ is enumerable. To prove that it is simple, we need to prove that $\mathcal{R}$ (being the complement of $\mathcal{N}$) is not finite and to prove that $\mathcal{R}$ has no enumerable, non-finite subpredicate. We will do so in the following sections.

7.1 The Random Numbers are Non-Finite

Recall Fact 20 stating that there are exactly 2^k distinct lists of numbers $l : \mathbb{LB}$ with length k. We can immediately lift this to the following:

Corollary 28. *There are exactly 2^k distinct numbers y with $|y| = k$:*

$$\exists l : \mathbb{LN}.\ (\forall y.\ y \in l \leftrightarrow |y| = k) \land \# l \land \mathsf{len}(l) = 2^k$$

Corollary 29. *There are at most $2^k - 1$ distinct numbers y with $|y| < k$:*

$$\forall l : \mathbb{LN}.\ (\forall x.\ x \in l \to |y| < k) \to \# l \mathsf{len}(l) < 2^k - 1$$

Lemma 30. *Not every number of size k is nonrandom:*

$$\forall k : \mathbb{N}. \neg \forall x : \mathbb{N}.\ |x| = k \to \mathcal{N}(x)$$

Proof. Suppose all 2^k many numbers of size k would be nonrandom, i.e. we have distinct x_i for $0 \le i < 2^k$ s.t. there is y_i with $|y_i| < |x_i| = k$ and $\delta y_i \triangleright x_i$.

Note that we have $|y|_i < k$ and all y_i are distinct (because $y_i = y_j$ implies $x_i = x_j$). But by the last corollary, there are at most $2^k - 1$ numbers y with $|y| < k$. Contradiction. ◀

Corollary 31. *There classically exist random numbers of every size:*

$$\forall k. \neg\neg \exists x.\ |x| = k \land \mathcal{R}(x)$$

Corollary 32. *$\mathcal{R}$ is not finite.*

7.2 The Nonrandom Numbers are Simple

Recall that we have assumed δ to be an optimal description function. Restricting optimality to total functions yields the following.

Fact 33. $\forall f : \mathbb{N} \to \mathbb{N}. \exists d : \mathbb{N}. \forall x : \mathbb{N}. \exists y_x.\ \delta y_x \triangleright f x \land |y_x| < |x| + d$

We only need this fact, not full optimality, for the following central result.

Lemma 34. *Given* MP*, $\mathcal{R}$ does not have an enumerable, non-finite subpredicate.*

Proof. Let q be enumerable, non-finite, and $\forall x.\ qx \to \mathcal{R}(x)$. We have to obtain a contradiction.

Since q is non-finite the relation $\lambda kx.\ k < |x| \land qx$ is classically total, i.e. $\forall k. \neg\neg \exists x.\ k < |x| \land qx$. Thus, by MP and Corollary 17 there is a function $f : \mathbb{N} \to \mathbb{N}$ s.t. for all $k < |fk|$ (1) and $q(fk)$. In particular, due to the latter and since q is a subpredicate of random numbers, $\forall ky.\ \delta y \triangleright fk \to |fk| \le |y|$ (2).

By optimality for total functions there exists d such that for all k there is y_k with $\delta y_k \triangleright fk$ and $|y_k| < |k| + d$ (3). Thus we have for all k:

$$k \overset{(1)}{<} |fk| \overset{(2)}{\le} |y_k| \overset{(3)}{\le} |k| + d.$$

This is a contradiction to Fact 19. ◀

Theorem 35. *Given* MP*, $\mathcal{N}$ is simple.*

▶ **Corollary 36.** *The following hold given* MP*:*

1. *$\mathcal{R}$ is undecidable.*
2. *$\mathcal{N}$ is undecidable.*
3. *$\mathcal{C}$ is uncomputable.*

8 The Role of Markov's Principle

Markov's principle MP is used to prove that $\mathcal{R}$ has no enumerable, non-finite subpredicate in the proof of simpleness of $\mathcal{R}$.

The simpleness proof of nonrandom numbers uses MP to obtain a choice function f for the classically total, enumerable relation $\lambda kx.\ k < |x| \wedge qx$. Then optimality is applied to f to obtain an upper bound for $\mathcal{C}_\delta(fx)$, i.e. d with $\forall x.\ \mathcal{C}_\delta(fx) < |x| + d$. However, optimality would readily apply to partial functions f.

Enumerable relations can equivalently be characterised using partial choice functions. This approach is not very common in type theory because partial functions are usually avoided, but in the setting of Kolmogorov complexity we have been relying on partial functions all along. We say that a partial function $f : X \rightharpoonup Y$ is a partial choice function for a relation $R : X \to Y \to \mathbb{P}$ if values computed by f are in the relation. A partial choice function has the same domain as R if $\forall x.(\exists y.\ Rxy) \leftrightarrow \exists y.\ fx \rhd y$. For functional relations, the conditions are equivalent to $Rxy \leftrightarrow fx \rhd y$. A relation is enumerable if and only if it has a partial choice function with the same domain.

Fact 37. *Every enumerable relation $R{:}\,\mathbb{N} \to \mathbb{N} \to \mathbb{P}$ has a partial choice function with the same domain, and vice versa:*

$$\mathcal{E}R \leftrightarrow \exists f{:}\,\mathbb{N} \rightharpoonup \mathbb{N}.\ (\forall xy.\ fx \rhd y \to Rxy) \wedge \forall x.\ (\exists y.\ Rxy) \to (\exists y.\ fx \rhd y)$$

If R is functional, then $\mathcal{E}R \leftrightarrow \exists f{:}\,\mathbb{N} \rightharpoonup \mathbb{N}.\ \forall xy.\ fx \rhd y \leftrightarrow Rxy$.

We can then use optimality to prove the following result applying to every classically total, enumerable relation R.

Lemma 38. *Let $R{:}\,\mathbb{N} \to \mathbb{N} \to \mathbb{P}$ be a classically total, enumerable relation. Then*

$$\exists d{:}\,\mathbb{N}.\forall x{:}\,\mathbb{N}.\exists y_x.\neg\neg\exists v_x.\ Rxv_x \wedge \delta y_x \rhd v_x \wedge |y|_x < |x| + d.$$

Proof. By Fact 37 R has a partial choice function $f{:}\,\mathbb{N} \rightharpoonup \mathbb{N}$ s.t. $fx \rhd v_x \to Rxv_x$ and $Rxv_x \to \exists v'_x.\ fx \rhd v'_x$. Optimality of δ used on f immediately yields the result. ◀

The rest of the proof then proceeds exactly as before:

Lemma 39. *$\mathcal{R}$ does not have an enumerable, non-finite sub-predicate.*

Proof. Let q be enumerable, non-finite, and $\forall x.\ qx \to \mathcal{R}(x)$. We have to obtain a contradiction.

Since q is non-finite $\lambda ky.\ y\ is\ \boldsymbol{\mu} x.\ \ k < |x| \wedge qx$ is classically total, i.e. for all k, there classically exists x_k s.t. $k < |x_k|$ (1) and qx_k. In particular, due to the latter, x_k is random: $\forall y.\ \delta y \rhd x_k \to |x_k| \leq |y|$.

By Lemma 38 there exists d such that for all k there exist y_k and there classically exists x_k with $k < |x_k|$ (1), qx_k, thus $\forall y.\ \delta y \rhd x_k \to |x_k| \leq |y|$ (2), $\delta y_k \rhd x_k$ and $|y_k| < |k| + d$ (3).

We now prove $\forall k.\ k < |k| + d$, a contradiction by Fact 19.

Let k be given. We prove $k < |k| + d$. Since $<$ is decidable, we can obtain y_k and x_k with properties (1) - (3) from above and have the wanted contradiction to Fact 19:

$$k \overset{(1)}{<} |x_k| \overset{(2)}{\leq} |y_k| \overset{(3)}{<} |k| + d$$ ◀

Note that overall, it would have been shorter to just explain the proof without MP in the paper. However, the proof becomes arguably a little more indirect by eliminating MP. We consciously decided to show both versions, because we think that being able to transparently compare both approaches can be helpful to eliminate uses of MP from other theorems not exclusively but especially in computability theory.

9 Universal codes

We now instantiate the definition of Kolmogorov complexity to a definition based on universal codes as used by Catt and Norrish [5]. We call a code u *universal* if $\forall c.\ \exists x.\ \forall y.\ \phi_c y \equiv \phi_u \langle x, y\rangle$ and define

$$\delta_u x := \phi_u x \qquad \mathcal{C}_u(x) s := s \text{ is } \boldsymbol{\mu} s.\ s = |y| \wedge \delta_u y \triangleright x \qquad \mathcal{N}_u x := \exists y.\ \delta_u y \triangleright x \wedge |y| < |x|$$

We need to prove optimality of δ_u for partial functions to deduce simpleness of $\mathcal{N}$:

Lemma 40. *If u is universal then δ_u is optimal:*

$$\forall f\colon \mathbb{N} \rightharpoonup \mathbb{N}.\exists d\colon \mathbb{N}.\forall x\colon \mathbb{N}.\ \exists y_x.\ \forall v_x.\ f x \triangleright v_x \to \phi_u y_x \triangleright v_x \wedge |y_x| < |x| + d.$$

Proof. By universality of u we have c s.t. $\lambda y.\ \phi_u \langle c, y\rangle$ computes f.

Pick $d := |c| + 1$. Now let x be given. Pick $y_x := \langle c, x\rangle$. Let v_x be given and $f x \triangleright v_x$. We have $\phi_u y_x = \phi_u \langle c, x\rangle \triangleright v_x$ and $|y_x| = |\langle c, x\rangle| = |c| + |x| < |x| + d$ as wanted. ◀

Lastly, it is easy to show that there exists a universal code based on the universality of ϕ:

Lemma 41. *If ϕ is universal, there exists a universal code u.*

Proof. Recall that $\langle x, y\rangle := \lfloor \lceil x \rceil + \lceil y \rceil \rfloor$. We can define

$$I(x\colon \mathbb{N}) := \lfloor [\underbrace{\mathsf{true}, \dots, \mathsf{true}}_{x \text{ times}}, \mathsf{false}] \rfloor$$

$$D_1 n := \lfloor \mathsf{first}\ i\ (\mathsf{skip}\ (i+1)\ \lceil n \rceil) \rfloor \text{ where } i \text{ is the index of the first } \mathsf{false} \text{ in } \lceil l \rceil$$

$$D_2 n := \lfloor \mathsf{skip}\ (2 \cdot i + 1)\ \lceil n \rceil \rfloor \text{ where } i \text{ is the index of the first } \mathsf{false} \text{ in } \lceil l \rceil$$

We have that $D_1 \langle \langle Ix, x\rangle, y\rangle = x$ and $D_2 \langle \langle Ix, x\rangle, y\rangle = y$.

Now let $f x := \phi_{D_1 x}(D_2 x)$ and u be its code w.r.t. ϕ. u is universal because given c we can pick $x := \langle Ic, c\rangle$ and have that $\phi_c y \equiv \phi_u \langle x, y\rangle$. ◀

We can construct a universal code u s.t. this definition of Kolmogorov complexity and nonrandom numbers agree with the ones used by Sipser [43], who treats both the code and the input to ϕ as the description.

Lemma 42. *If pairing is computabily invertible (i.e. given a function* inv *with* $\mathsf{inv}\langle x, y\rangle = (x, y)$*), there is a universal code u s.t. $\mathcal{C}_u(x)$ is equivalent to $\lambda y.\ y$ is $\boldsymbol{\mu} s.\ \exists c e.\ s = |c| + |e| \wedge \phi_c e \triangleright x$ and $\mathcal{N}_u x$ is equivalent to $\exists c\ e.\ \phi_c e \triangleright x \wedge |c| + |e| < |x|$.*

Proof. Let a code u be strongly universal if $\forall c i.\ \phi_c i \equiv \phi_u \langle c, i\rangle$. It is immediate that every strongly universal code is universal and that every strongly universal code fulfils the properties.

Thus, it suffices to prove that there exists a strongly universal code: Define

$$f x := \textbf{let}\ (c, i) := \mathsf{inv}\ x\ \textbf{in}\ \phi_c i.$$

By universality of ϕ there exists a code u s.t. $\forall x.\ \phi_u x \equiv f x$. I.e. $\phi_c i \equiv f \langle c, i\rangle \equiv \phi_u \langle c, i\rangle$. ◀

Catt and Norrish define pairing as $\lfloor Ix + \lceil x \rceil + \lceil y \rceil \rfloor$ with I being the function from the proof of Lemma 41, which makes pairing immediately computationally invertible. For our definition of pairing, we do not know how to define such an inversion operation, but the whole development in this paper does not depend on the concrete definition of pairing and could as well be carried out with Catt and Norrish's definition.

10 Fixed inputs

Odifreddi [34] defines the Kolmogorov complexity of x as the smallest c such that the c-th Turing machine (w.r.t. an effective enumeration of Turing machines) outputs x when ran on a blank tape. That is, he defines the description operator to simulate on input c the c-th Turing machine on blank tape, and then defines the size of a Turing machine as exactly its code. Thus, there is exactly one Turing machine of size k, rather than exponentially many.

This non-conventional approach allows Odifreddi to obtain an elegant proof of simpleness of nonrandom numbers using Rogers' fixed-point theorem. However, Odifreddi's definition is not invariant w.r.t. other definitions (in the sense of Theorem 23) since there are strings with Kolmogorov complexity exponentially larger than using our definition.

However, one can use the first part of the definition to instantiate our definition as well. That is, we can define δ in terms of ϕ for an arbitrary fixed input i.

$$\delta c := \phi_c i \qquad \mathcal{C}(x)s := s \; is \; \boldsymbol{\mu} s.\; \exists c.\; s = |c| \land \phi_c i \rhd x \qquad \mathcal{N}x := \exists c.\; \phi_c i \rhd x \land |c| < |x|$$

The assumed universality of ϕ does not suffice to prove that $\mathcal{N}$ is simple using this definition. We need to adapt it in order to make statements on the size of γx for certain functions as follows.

Theorem 43. *If* $\forall f : \mathbb{N} \rightharpoonup \mathbb{N}.\; \exists \gamma : \mathbb{N} \to \mathbb{N}. \exists d : \mathbb{N}.\; \forall x i.\; \phi_{\gamma x} i \equiv f x \land (\exists y.\; f x \rhd y) \to |\gamma x| < |x| + d$, *then* $\mathcal{N}$ *is simple.*

Note that this is strictly stronger than the universality condition, but it can similarly be proved consistent by showing that a formulation of CT_{L} as discussed by Forster [13, §7, Appendix A] also implies this version.

11 Related Work

Vitanyi [48] attributes the first complete, spelled-out undecidability proof to Zvonkin and Levin in 1970 [50]. Zvonkin and Levin themselves refer to Kolmogorov who claimed the uncomputability of Kolmogorov complexity in 1965 by pointing to the undecidability of the halting problem [25], however without a real proof. Furthermore, the paper by Zvonkin and Levin seems to contain the first proof that that the nonrandom numbers are simple, but attribute the result to Janis Barzdins.

Other definitions of Kolmogorov complexity. Odifreddi [34] defines Kolmogorov complexity as $\mathcal{C}(x) := \boldsymbol{\mu} c.\; \phi_c 0 \rhd x$, i.e. assuming $|c| = c$. Our theory cannot be instantiated to such definitions of Kolmogorov complexity, because it contradicts property **3** of the encoding of numbers as binary strings. In particular, there are just k many numbers x with $|x| < k$ and just one number with exactly size k, so our proof cannot be used directly. Odifreddis' proof seems to be about as complicated as our proof, but not considering a size function allows a very elegant use of Roger's fixed-point theorem [41], of which Forster [13] gives a machine-checked proof. Thus, everything would be in place to also give a machine-checked proof that nonrandom numbers form a simple predicate based on Odifreddi's definition in our setting, but we leave this to future work.

Other machine-checked proofs regarding Kolmogorov complexity. In Section 9 we discuss a synthetic version of the definition used by Catt and Norrish [5].

There are various differences in the approaches: First, Catt and Norrish define ϕ in terms of the λ-calculus. In order to obtain c s.t. ϕ_c computes f they have to explicitly construct a λ-term computing f. Terms of the λ-calculus constitute about 200 lines of code of their HOL4 formalisation (not accounting for proofs about these terms) which has approximately 6000 lines in total. For comparison of the magnitude, our whole development has around 800 lines of Coq code. Note that we just give these numbers for comparison of magnitude and do not want to relate the complexity of Coq code and HOL4 code via line numbers.

Catt and Norrish use an operation MIN_SET computing the least element of a non-empty set to define a Kolmogorov complexity function. In general, such an operation requires at least a unique choice operator and the law of excluded middle. We instead work with functional relations, which poses no problems and avoids the operator. They also use classical logic, i.e. rely on LEM. While classical logic would be available in our setting, our result also shows that it is not needed. Classical logic is only available in our synthetic setting because we avoid (unique) choice operators [11]. Note that Catt and Norrish do not consider random or nonrandom numbers and prove instead that $\mathcal{C}$ is uncomputable. This is a direct consequence of the proof that $\mathcal{N}$ is simple, but the proof idea is exactly the same.

Other results on Kolmogorov complexity. Catt and Norrish [5] also prove the Kraft and Kolmogorov-Kraft inequalities and various other information-theoretic properties from the book by Hutter [20, Theorem 2.10]. We do not see any hurdles in adding them on top of our development, but leave them for future work as we focus on computability-theoretic properties. Regarding computability Kummer [26] proves that $\mathcal{N}$ is truth-table complete, i.e. that every enumerable predicate p, and in particular the halting problem $\mathcal{K}$ truth-table reduces to $\mathcal{N}$. The proof is highly involved and would pose an interesting challenge for interactive theorem proving. Forster, Jahn, and Smolka [14] give a machine-checked construction of a simple but truth-table complete predicate following Post, which is easier.

Other machine-checked proofs in computability and complexity theory. Norrish [33] proves Rice's theorem in HOL4 based on the full λ calculus, the development is now underlying the work on Kolmogorov complexity with Catt.

Forster and Smolka [17] prove Rice's theorem in Coq based on the weak call-by-value λ calculus. Forster, Kunze, Smolka, and Wuttke [16] prove that the weak call-by-value λ calculus and Turing machines can simulate each other with polynomial overhead in time. Gäher and Kunze [19] prove the Cook-Levin theorem stating that SAT is NP-complete with a definition of NP in the weak call-by-value λ-calculus, but by employing the simulation of the λ-calculus on Turing machines to encode computation as boolean circuits.

Xu, Zhang, and Urban [49] prove that the halting problem of Turing machines is undecidable in Isabelle. Larchey-Wendling [27] proves that every total μ-recursive function gives rise to a Coq function. Carneiro [4] proves Rice's theorem in Lean based on μ-recursive functions. Ramos et al. [10] prove Rice's theorem in PVS based on PVS0 and assuming a fixed-point theorem. Forster, Jahn, and Smolka [14] prove a synthetic version of Myhill's isomorphism theorem and a synthetic computational version of the Cantor-Bernstein theorem without assuming any axioms. Lastly, Forster [13, 12] proves Rice's theorem and Forster, Jahn, and Smolka [14] develop the theory of one-one, many-one, and truth-table degrees and construct simple and hypersimple sets, based on the axiom EA, equivalent to EPF.

12 Discussion

We conclude by discussing several more general points concerning machine-checked proofs, synthetic computability, and constructive mathematics.

For the scope of this project, translating from textbook proofs to the synthetic settings was barely noticable: The look and feel is like working in a textbook setting. Classical logic is available and the axiom of choice, albeit being explicitly acknowledged as foundation e.g. by Rogers [41], does not play any practical role. Instead of using (unique) choice operators to obtain functions one can work with the – often anyways more convenient – (classically) total functional relation directly.

The original textbook proofs on Kolmogorov complexity are heavily classical. The first steps of the present paper were undertaken as part of the third author's Bachelor's thesis [29], and explicitly used the law of excluded middle. The agnosticism of constructive type theory and Coq w.r.t classical provided very helpful in this regard: `Require Import Classical.` turns Coq into a classical proof assistant and basic logical automation tactics like `tauto` use classical logic immediately.

In a second pass, also part of the mentioned Bachelor's thesis, the uses of LEM were analysed one by one and either circumvented by a constructivisation of the statement, or by a use of Markov's principle where no immediate constructivisation was possible. For instance, the usual definition "There exist members of every size." $(\forall s.\exists n \geq s.\ pn)$ to define infinite sets has to be translated to the constructively weaker definition of infinity, "For every size, the subset of members of this size is not non-empty." $(\forall n.\neg(\neg\exists n.\ n \geq s \wedge pn))$. Here, Coq serves as a true proof *assistant*: It helps in keeping track of details and assumptions and keeps its user honest by forcing them to apply classical axioms explicitly. Manually constructivising proofs on paper would be more time-intensive, as proofs have to be checked repeatedly on axiom usage to be sure.

In the case of Kolmogorov complexity, obtaining fully constructive proofs directly from textbook proofs seems unfeasible. Even obtaining proofs based solely on MP requires a great deal of experience with constructive proofs. Thus it is crucial that both Coq and our setting of synthetic computability are compatible with classical reasoning: First working up to LEM, then up to MP and only then re-working the proofs to not use MP turned out to be feasible.

Overall, our paper can be seen as a case study substantiating various points often found in papers on interactive theorem proving:

First, giving a machine-checked proof has benefits distinct from the question of correctness. The ITP forces users to think about the most concise and elegant ways to express definitions, statements, and proofs and to isolate the properties of structures necessary to prove the wanted results. In our case, we can isolate that the assumption of an optimal description function suffices to prove the existence of a simple predicate, no other properties of the description function or the subsequent definition of Kolmogorov complexity are needed.

Secondly, thinking about constructive proofs has benefits distinct from the question which axioms are necessary for a result. Constructive logic enforces direct arguments, thus making presentations easier to follow, and makes proofs by contradiction explicit already in theorem statements, thus making presentations more transparent.

Thirdly, having the *possibility* to assume classical axioms is vital. Nowadays, few (if any) constructive mathematicians outright reject axioms like the law of excluded middle, but rather emphasise the benefits for presentations on the one hand, and on the other hand the academic curiosity motivating the analysis of which assumptions constitute the minimal set of assumptions necessary for any given theorem. The circumstance that our setting of

synthetic computability theory allows classical axioms, in contrast to previous approaches forming an anti-classical endeavour, was crucial for the present paper, which as already mentioned started out as a Bachelor's thesis project: Due to the pioneering work by Catt and Norrish it was clear that a machine-checked proof based on classical axioms is in reach, the first task was just to translate it to the simpler setting of synthetic computability theory. Probably, we would not have dared posing a simultaneous constructivisation and translation to the synthetic setting for a thesis project on Bachelor's level.

References

1 Andrej Bauer. First steps in synthetic computability theory. *Electronic Notes in Theoretical Computer Science*, 155:5–31, 2006. doi:10.1016/j.entcs.2005.11.049.

2 Andrej Bauer. On fixed-point theorems in synthetic computability. *Tbilisi Mathematical Journal*, 10(3):167–181, 2017. doi:10.1515/tmj-2017-0107.

3 Douglas Bridges and Fred Richman. *Varieties of constructive mathematics*, volume 97. Cambridge University Press, 1987. doi:10.1017/CBO9780511565663.

4 Mario Carneiro. Formalizing Computability Theory via Partial Recursive Functions. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:17, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ITP.2019.12.

5 Elliot Catt and Michael Norrish. On the formalisation of Kolmogorov complexity. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, January 2021. doi:10.1145/3437992.3439921.

6 Gregory J. Chaitin. On the simplicity and speed of programs for computing infinite sets of natural numbers. *Journal of the ACM*, 16(3):407–422, July 1969. doi:10.1145/321526.321530.

7 Thierry Coquand and Gérard P Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988. doi:10.1016/0890-5401(88)90005-3.

8 Andrej Dudenhefner. The undecidability of system F typability and type checking for reductionists. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 – July 2, 2021*, pages 1–10. IEEE, 2021. doi:10.1109/LICS52264.2021.9470520.

9 Andrej Dudenhefner. Constructive many-one reduction from the halting problem to semi-unification. In Florin Manea and Alex Simpson, editors, *30th EACSL Annual Conference on Computer Science Logic, CSL 2022, February 14-19, 2022, Göttingen, Germany (Virtual Conference)*, volume 216 of *LIPIcs*, pages 18:1–18:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.CSL.2022.18.

10 Thiago Mendonça Ferreira Ramos, Ariane Alves Almeida, and Mauricio Ayala-Rincón. Formalization of rice's theorem over a functional language model. Technical report, University of Brasília, 2020. URL: https://www.mat.unb.br/~ayala/RiceThFormalization.pdf.

11 Yannick Forster. Church's Thesis and Related Axioms in Coq's Type Theory. In Christel Baier and Jean Goubault-Larrecq, editors, *29th EACSL Annual Conference on Computer Science Logic (CSL 2021)*, volume 183 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CSL.2021.21.

12 Yannick Forster. *Computability in Constructive Type Theory*. PhD thesis, Saarland University, 2021. doi:10.22028/D291-35758.

13 Yannick Forster. Parametric church's thesis: Synthetic computability without choice. In Sergei N. Artëmov and Anil Nerode, editors, *Logical Foundations of Computer Science - International Symposium, LFCS 2022, Deerfield Beach, FL, USA, January 10-13, 2022, Proceedings*, volume 13137 of *Lecture Notes in Computer Science*, pages 70–89. Springer, 2022. doi:10.1007/978-3-030-93100-1_6.

14 Yannick Forster, Felix Jahn, and Gert Smolka. A Constructive and Synthetic Theory of Reducibility: Myhill's Isomorphism Theorem and Post's Problem for Many-one and Truth-table Reducibility in Coq (Full Version), February 2022. preprint. URL: https://hal.inria.fr/hal-03580081.

15 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 - CPP 2019*. ACM Press, 2019. doi:10.1145/3293880.3294091.

16 Yannick Forster, Fabian Kunze, Gert Smolka, and Maximilian Wuttke. A mechanised proof of the time invariance thesis for the weak call-by-value λ-calculus. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Dagstuhl, Germany, 2021. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

17 Yannick Forster and Gert Smolka. Weak call-by-value lambda calculus as a model of computation in Coq. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, volume 10499 of *Lecture Notes in Computer Science*, pages 189–206. Springer, 2017. doi:10.1007/978-3-319-66107-0_13.

18 R. M. Friedberg. Two recursively enumerable sets of incomparable degrees of unsovlability (solution of post's problem, 1944. *Proceedings of the National Academy of Sciences*, 43(2):236–238, February 1957. doi:10.1073/pnas.43.2.236.

19 Lennard Gäher and Fabian Kunze. Mechanising complexity theory: The cook-levin theorem in Coq. In *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.ITP.2021.20.

20 Marcus Hutter. *Universal Artificial Intellegence*. Springer Berlin Heidelberg, 2005. doi:10.1007/b138233.

21 Dominik Kirst and Dominique Larchey-Wendling. Trakhtenbrot's theorem in Coq. In *Automated Reasoning*, pages 79–96. Springer International Publishing, 2020. doi:10.1007/978-3-030-51054-1_5.

22 Steven C. Kleene and Emil L. Post. The upper semi-lattice of degrees of recursive unsolvability. *The Annals of Mathematics*, 59(3):379, May 1954. doi:10.2307/1969708.

23 John R. Kline. The April meeting in New York. *Bulletin of the American Mathematical Society*, 54(7):622–648, July 1948. doi:10.1090/s0002-9904-1948-09030-9.

24 Andrei N Kolmogorov. On tables of random numbers. *Sankhyā: The Indian Journal of Statistics, Series A*, pages 369–376, 1963. doi:10.1016/S0304-3975(98)00075-9.

25 Andrei N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems of information transmission*, 1(1):3–11, 1965.

26 Martin Kummer. On the complexity of random strings. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 25–36. Springer, 1996. doi:10.1007/3-540-60922-9_3.

27 Dominique Larchey-Wendling. Typing total recursive functions in Coq. In *International Conference on Interactive Theorem Proving*, pages 371–388. Springer, 2017. doi:10.1007/978-3-319-66107-0_24.

28 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. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. doi:10.4230/LIPIcs.FSCD.2019.27.

29 Nils Lauermann. A formalization of Kolmogorov complexity in synthetic computability theory. Bachelor's thesis, Saarland University, 2021. URL: https://ps.uni-saarland.de/~lauermann/bachelor.php.

30 Ming Li and Paul Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer New York, 2008. doi:10.1007/978-0-387-49820-1.

31 Albert Abramovich Muchnik. On strong and weak reducibility of algorithmic problems. *Sibirskii Matematicheskii Zhurnal*, 4(6):1328–1341, 1963.

32 John Myhill. Creative sets. *Zeitschr. f. math. Logik und Grundlagen d. Math.*, 1957. doi:10.1002/malq.19550010205.

33 Michael Norrish. Mechanised computability theory. In *ITP 2011*, volume 6898 of *LNCS*, pages 297–311. Springer, 2011. doi:10.1007/978-3-642-22863-6_22.

34 Piergiorgio Odifreddi. *Classical recursion theory: The theory of functions and sets of natural numbers*. Elsevier, 1992.

35 Christine Paulin-Mohring. Inductive definitions in the system Coq rules and properties. In *International Conference on Typed Lambda Calculi and Applications*, pages 328–345. Springer, 1993. doi:10.1007/BFb0037116.

36 Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions, January 2015. URL: https://hal.inria.fr/hal-01094195.

37 Emil L. Post. Recursively enumerable sets of positive integers and their decision problems. *bulletin of the American Mathematical Society*, 50(5):284–316, 1944. doi:10.1090/S0002-9904-1944-08111-1.

38 Emil L. Post. Degrees of recursive unsolvability - preliminary report. In *Bulletin of the American Mathematical Society*, volume 54(7), pages 641–642. American Mathematical Society (AMS), July 1948. doi:10.1090/s0002-9904-1948-09030-9.

39 Henry G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953. doi:10.1090/S0002-9947-1953-0053041-6.

40 Fred Richman. Church's thesis without tears. *The Journal of symbolic logic*, 48(3):797–803, 1983. doi:10.2307/2273473.

41 Hartley Rogers. *Theory of Recursive Functions and Effective Computability*. MIT Press, Cambridge, MA, USA, 1987.

42 A. Shen, V. Uspensky, and N. Vereshchagin. *Kolmogorov Complexity and Algorithmic Randomness*. American Mathematical Society, November 2017. doi:10.1090/surv/220.

43 Michael Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006. doi:10.1145/230514.571645.

44 Ray J. Solomonoff. A preliminary report on a general theory of inductive inference (revision of report v-131). *Contract AF*, 49(639):376, 1960.

45 R.J. Solomonoff. A formal theory of inductive inference. part i. *Information and Control*, 7(1):1–22, March 1964. doi:10.1016/s0019-9958(64)90223-2.

46 R.J. Solomonoff. A formal theory of inductive inference. part II. *Information and Control*, 7(2):224–254, June 1964. doi:10.1016/s0019-9958(64)90131-7.

47 The Coq Development Team. The Coq proof assistant version 8.13.2, January 2021. doi:10.5281/zenodo.4501022.

48 Paul M.B. Vitányi. How incomputable is Kolmogorov complexity? *Entropy*, 22(4):408, April 2020. doi:10.3390/e22040408.

49 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. doi:10.1007/978-3-642-39634-2_13.

50 A K Zvonkin and L A Levin. The complexity of finite objects and the development of the concepts of information and randomness by means of the theory of algorithms. *Russian Mathematical Surveys*, 25(6):83–124, December 1970. doi:10.1070/rm1970v025n06abeh001269.

Verifying a Sequent Calculus Prover for First-Order Logic with Functions in Isabelle/HOL

Asta Halkjær From
DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark

Frederik Krogsdal Jacobsen
DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark

Abstract

We describe the design, implementation and verification of an automated theorem prover for first-order logic with functions. The proof search procedure is based on sequent calculus and we formally verify its soundness and completeness in Isabelle/HOL using an existing abstract framework for coinductive proof trees. Our analytic completeness proof covers both open and closed formulas. Since our deterministic prover considers only the subset of terms relevant to proving a given sequent, we do so as well when building a countermodel from a failed proof. Finally, we formally connect our prover with the proof system and semantics of the existing SeCaV system. In particular, the prover can generate human-readable SeCaV proofs which are also machine-verifiable proof certificates.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Theory of computation → Proof theory; Theory of computation → Program verification

Keywords and phrases Isabelle/HOL, SeCaV, First-Order Logic, Prover, Soundness, Completeness

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.13

Supplementary Material *Software (Isabelle/HOL formalization in the Archive of Formal Proofs)*: `https://www.isa-afp.org/entries/FOL_Seq_Calc2.html` [18]

Acknowledgements We would like to thank Agnes Moesgård Eschen, Alexander Birch Jensen, Anders Schlichtkrull, Simon Tobias Lund and Jørgen Villadsen for comments on drafts. We are very grateful to the anonymous reviewers for their thoughtful comments.

1 Introduction

While there are many automated theorem provers capable of proving theorems involving very large formulas and many lemmas, very few of them have formalized proofs of metatheoretical properties such as soundness and completeness. This leads to issues of trust: how do we know that the answers returned by automated theorem provers are actually correct? And do we know that our automated theorem provers will actually be able to prove what we want them to? Even those provers that can generate proof certificates to support their answers may not always be trustworthy, since some proof techniques lead to proofs that are very difficult to follow for a human, and are thus difficult to check for correctness.

Formalizing the soundness and completeness of a prover provides two crucial benefits. With a soundness result, we know that the prover does not erroneously accept an invalid formula and outputs a wrong proof of the formula. Thus, advanced features and optimizations cannot cause unforeseen flaws in the prover. Completeness of the prover is especially useful in combination with the possibility of generating readable proof certificates. With formalized completeness, we can use the prover as a tool to generate step-by-step proofs of any valid formula, and it can thus also be used to gain understanding, e.g. by students trying to understand why a counter-intuitive formula is valid. While there are some systems with formalized metatheory, they rarely include executable provers, often cannot generate proof certificates, and are often quite limited in their expressive power (cf. Section 1.1).

In this paper, we present an automated theorem prover for first-order logic with functions based on sequent calculus. We formalize its soundness and completeness in Isabelle/HOL. We reuse the syntax and semantics of first-order logic from the Sequent Calculus Verifier (SeCaV) system [16] (Section 2.1). We state the soundness and completeness of the prover with respect to the SeCaV proof system, its semantics and a bounded semantics that we introduce here. The prover can generate human-readable and machine-verifiable SeCaV proofs for valid formulas.

Our formalization instantiates an abstract framework of coinductive proof trees by Blanchette et al. [11] (Section 2.2). By instantiating the framework with concrete functions implementing our sequent calculus, it builds a prover for us (Section 3). By discharging further proof obligations, the framework proves that any proof tree built by our prover is either finite or contains an infinite path with certain properties. We then build either a SeCaV proof from the finite tree (Section 4) or a countermodel from the failed proof attempt (Section 5). As far as we are aware, we are the first to use the framework to prove soundness and completeness of a non-trivial executable prover (as opposed to simply a calculus).

Our prover is deterministic, fair and works on finite sequents. To handle the quantifiers we must thus build our countermodel in a Herbrand universe that contains only the subset of terms that actually appear in the failed proof. This idea is inspired by Ben-Ari's textbook proof [2], where terms are either variables or constants, and by Ridge's Isabelle proof [38], where only variables are considered. We are not aware of any previous formalization of this construction that handles functions. We consider all terms in our Herbrand universe, including those with free variables, yielding completeness for both open and closed formulas.

The prover is free software and the source code is available as supplementary material. This consists of around 3000 lines of Isabelle/HOL and 1300 lines of supplementary Haskell.

We summarize our main contributions:

- A formally verified sound and complete automated theorem prover for full first-order logic with functions.
- An analytic proof of completeness for both open and closed formulas for a deterministic prover via a bounded semantics.
- A method of translating the prover-generated certificates of validity into human-readable and machine-verifiable proofs in SeCaV.
- A concrete application of the abstract completeness framework, and a demonstration of how to obtain soundness and completeness of an actual, executable prover using the framework as a starting point.

We summarize the results and discuss the generated proofs, challenges encountered during the verification, prover limitations and future work in Section 6 before concluding in Section 7.

1.1 Related work

The present paper is a much improved version of the work started in the second author's master's thesis [21]. The Sequent Calculus Verifier (SeCaV) is a well-established proof system, and both soundness and completeness have been proven for the system [19]. The system has been used to teach students in several courses at the Technical University of Denmark [20, 47]. An online tool called the SeCaV Unshortener has been developed to allow input of proofs in a simple format which is then translated to an Isabelle proof [16].

Our prover is based on the abstract completeness framework by Blanchette et al. [10, 11]. The framework contains a simple example prover for propositional logic, and the original application of the framework was in the formalization of the metatheory of the Sledgehammer

tool for automated theorem proving within Isabelle/HOL [8]. Blanchette et al. [11] have used the framework to formalize soundness and completeness of a calculus for first-order logic with equality and in negation normal form. Their search is nondeterministic and they do not generate an executable prover like we do. As such, we improve on their work by using the framework to prove soundness and completeness of an executable prover.

A number of other systems have formally verified metatheories. NaDeA (Natural Deduction Assistant) by Villadsen et al. [49] is a web application that allows users to prove formulas with natural deduction. The metatheory of a model of the system is formalized in Isabelle/HOL, and the application allows export of proofs for verification in Isabelle. The Incredible Proof Machine by Breitner [12] is a web application that allows users to create proofs using a specialized graphical interface. The proof system is as strong as natural deduction, and a model of the system is formalized in Isabelle using the abstract framework by Blanchette et al. [11]. Neither system includes automated theorem provers; they are essentially simple proof assistants designed to aid students in understanding logical systems.

THINKER by Pelletier [35] is a proof system and an attached automated theorem prover. THINKER is a natural deduction system designed to allow for what the author calls "direct proofs", as opposed to proofs based on reduction to a resolution system. THINKER was perhaps the first automated theorem prover designed specifically with "naturality" in mind, as a reaction to the indirectness of resolution-based proof systems. MUSCADET by Pastre [34] is also an automated theorem prover based on natural deduction. The system distinguishes itself by also supporting usage of prior knowledge such as previously proven theorems through a Prolog knowledge base.

While there are many very advanced automated theorem provers such as Vampire [24], Zipperposition [3] and Z3 [13], their metatheory and implementations are rarely formalized. As a first step towards formally verifying modern provers, Schlichtkrull et al. [40] have formalized an ordered resolution prover for *clausal* first-order logic in Isabelle/HOL. Jensen et al. [22] formalized the soundness, but not the completeness, of a prover for first-order logic with equality in Isabelle/HOL. Villadsen et al. [50] verified a simple prover for first-order logic in Isabelle/HOL with the aim of allowing students to understand both the prover and the formalization. That work is based on an earlier formalization by Ridge and Margetson [38], but simplifies both the prover and the proofs to enable easier understanding by students. Neither of these two provers provide support for functions or generation of proof certificates.

Blanchette [5] gives an overview of a number of verification efforts including the metatheory of SAT solvers [6, 14, 29, 30, 43] and certificate checkers [26, 27], SMT solvers [28, 31, 45], the superposition calculus [37], resolution [36, 39, 41], a number of non-classical logics [15, 17, 42, 46, 48], and a wide range of proof systems for classical propositional logic [32, 33]. Some of these efforts are part of the IsaFoL project (Isabelle Formalization of Logic). Part of the goal is to develop "a methodology for formalizing modern research in automated reasoning". Our work points in this direction too, by formally verifying a non-saturation-based prover.

2 Background

In this section, we briefly introduce the two existing things we build on: the Sequent Calculus Verifier (SeCaV) system and the abstract framework by Blanchette et al [11]. In particular, we have not modified these projects in any way for our use, and their designs thus significantly influence the design of our prover.

```
datatype tm = Fun nat (tm list) | Var nat

datatype fm =
  Pre nat (tm list) | Imp fm fm | Dis fm fm | Con fm fm | Exi fm | Uni fm | Neg fm
```

Figure 1 The syntax of the Sequent Calculus Verifier (parentheses added for clarity).

```
definition shift e v x ≡ λn. if n < v then e n else if n = v then x else e (n − 1)

primrec semantics-term and semantics-list where
  semantics-term e f (Var n) = e n
| semantics-term e f (Fun i l) = f i (semantics-list e f l)
| semantics-list e f [] = []
| semantics-list e f (t # l) = semantics-term e f t # semantics-list e f l

primrec semantics where
  semantics e f g (Pre i l) = g i (semantics-list e f l)
| semantics e f g (Imp p q) = (semantics e f g p ⟶ semantics e f g q)
| semantics e f g (Dis p q) = (semantics e f g p ∨ semantics e f g q)
| semantics e f g (Con p q) = (semantics e f g p ∧ semantics e f g q)
| semantics e f g (Exi p) = (∃x. semantics (shift e 0 x) f g p)
| semantics e f g (Uni p) = (∀x. semantics (shift e 0 x) f g p)
| semantics e f g (Neg p) = (¬ semantics e f g p)
```

Figure 2 The semantics of the Sequent Calculus Verifier (# separates the head and tail of a list).

2.1 The Sequent Calculus Verifier

The system is a one-sided sequent calculus for first-order logic with functions. Constants are encoded as functions with arity 0. Figure 1 gives the syntax of terms and formulas as Isabelle/HOL datatypes. The system uses de Bruijn indices to identify variables, while functions and predicates are named by natural numbers. Besides predicates, the system includes implication, disjunction, conjunction, existential quantification, universal quantification, and negation (in that order in Figure 1). Predicates and functions take their arguments as ordered lists of terms, which may be empty. Sequents are ordered lists of formulas. Parameterized datatypes are written in postfix notation, e.g. the type *tm list* of lists containing terms.

The semantics of a formula is due to Berghofer [4], who models the universe as a type variable like we do for now. The interpretation consists of an environment e for variables, a function denotation f and a predicate denotation g. The semantics of the system is standard and defined using the three recursive functions in Figure 2. The semantics of the logical connectives is defined using the connectives from the meta-logic in Isabelle/HOL. The *shift* function handles shifting de Bruijn-indices when interpreting quantifiers. We say that a sequent is valid when, under all interpretations, some formula in the sequent is satisfied.

The system has a number of proof rules, some of which are displayed in Figure 3 (abusing set notation for the membership and inclusion relations on lists – see the formalization for details and the remaining rules). The rules should be read from the bottom up, since we generally work backwards from a sequent we wish to prove. The rules are classified according to Smullyan's uniform notation [44].

The first proof rule, Basic, terminates the branch and applies when the sequent contains both a formula and its negation. Isabelle/HOL allows pattern matching only on the head of a list, so to simplify the specification of this rule, the positive formula must come first.

$$\frac{\text{Neg } p \in z}{\Vdash p, z}\ \textsc{Basic} \qquad \frac{\Vdash z \qquad z \subseteq y}{\Vdash y}\ \textsc{Ext} \qquad \frac{\Vdash p, z}{\Vdash \text{Neg (Neg } p), z}\ \textsc{NegNeg}$$

$$\frac{\Vdash p, q, z}{\Vdash \text{Dis } p\ q, z}\ \textsc{AlphaDis} \qquad \frac{\Vdash \text{Neg } p, z \qquad \Vdash \text{Neg } q, z}{\Vdash \text{Neg (Dis } p\ q), z}\ \textsc{BetaDis}$$

$$\frac{\Vdash p\,[\text{Var } 0/t], z}{\Vdash \text{Exi } p, z}\ \textsc{GammaExi} \qquad \frac{\Vdash \text{Neg } (p\,[\text{Var } 0/\text{Fun } i\ []]), z \qquad i \text{ fresh}}{\Vdash \text{Neg (Exi } p), z}\ \textsc{DeltaExi}$$

Figure 3 Sample proof rules for the Sequent Calculus Verifier (rules omitted here are similar).

The structural Ext rule can be applied to change the position of formulas in a sequent (permutation), duplicate an existing formula (contraction), and remove formulas that are not needed (weakening). It is crucial, since most rules in the system work only on the first formula in a sequent. Duplicating a formula is necessary if a quantified formula needs to be instantiated several times, since γ-rules (starting with Gamma) destroy the original formula.

The NegNeg rule removes a double negation from the first formula in a sequent. It can be considered an α-rule, but we keep it separate from the others because it does not generate two formulas. The AlphaDis rule decomposes disjunctions (and similar for the AlphaImp and AlphaCon rules omitted here). The BetaDis rule decomposes negated disjunctions and requires that two sequents are proven separately, creating branches in the proof tree (and similar for the BetaCon, BetaImp rules omitted here). This essentially moves the connective into the proof tree itself, since both branches now need to be proven separately. The GammaExi rule instantiates an existential quantifier with any term t by substituting t for variable 0 in the quantified formula. The GammaUni rule omitted here is similar. The DeltaExi rule instantiates a negated existential quantifier in the first formula in a sequent with a fresh constant function, with fresh here meaning that the function identifier does not already occur anywhere in the sequent. The fresh constant cannot have any relationship to other terms in the sequent: it is *arbitrary*. Thus we could have used any other term without affecting the validity of the formula, which is exactly what is needed to prove a universally quantified ("there does not exist") formula. The DeltaUni rule omitted here is similar.

The proof system in Figure 3 has been formally verified to be sound and complete with regards to the semantics in Figure 2 by From et al. [19]. We use these results to relate our prover to SeCaV.

2.2 Abstract frameworks for soundness and completeness

Blanchette et al. [11] have formalized an abstract framework to facilitate soundness and completeness proofs by coinductive methods. In particular, they give abstract definitions that can be instantiated to a concrete sequent calculus or tableau prover. They facilitate proofs in the Beth-Hintikka style: the search "builds either a finite deduction tree yielding a proof (...) or an infinite tree from which a countermodel (...) can be extracted." The framework consists of a number of Isabelle/HOL locales that must be instantiated and in return provide various definitions and proofs.

Locales [1, 23] allow the abstraction of definitions and proofs over given parameters. As an example, consider groups in algebra defined by a carrier set, a binary operation and the group axioms. With a locale, these can be specified abstractly and a number of operations

Table 1 The *RuleSystem* locale with premises above the line and important conclusions below.

eff	Effect relation between a rule, a state and a finite set of resulting states.
rules	Stream of rules. The set of these is called R.
S	Set of well formed states.
eff-S	Proof that for any rule in R and proof state in S the *eff*-related states are in S.
enabled-R	Proof that for any state in S, some rule in R is *enabled*, i.e. applies to that state.
mkTree	A function from a stream of rules and a starting state to a tree of states and rules.
wf-mkTree	Proof that the tree generated by *mkTree* is well formed wrt. *eff*.

Table 2 The *PersistentRuleSystem* locale which extends *RuleSystem* from Table 1.

per	Proof that if a rule r in R is enabled in a well formed state s and s' is *eff*-related to s by a rule r' in R distinct from r, then r is enabled in s'.
epath-completeness-Saturated	Proof that for any well formed state s, there exists either a well formed finite tree with s as root or a saturated escape path with s as root.

and results can then be given in the abstract. Later, we can instantiate the locale with a concrete group by providing the carrier set and binary operation, and proving that the group axioms are fulfilled. We then obtain instantiations of the results for our concrete group.

In this section we give an overview of the locales provided by the abstract framework: what they require and what they provide. We have condensed the Isabelle code into four tables for brevity, since the specific details of the framework are not our main focus. The exact definitions can be found in the Archive of Formal Proofs entry by Blanchette et al. [9].

First, two coinductive datatypes are crucial: a *tree* is finitely branching but can be infinitely deep, while a *stream* has no branching but is decidedly infinite (a list with no end).

Tables 1 and 2 cover the two locales *RuleSystem* and *PersistentRuleSystem* which are central for proving completeness. The locale premises are given above each vertical line and the (important) conclusions are given below. The locales require us to prove a number of things about three definitions. First, the *eff* relation specifies the effect of applying a rule to a state in our proof search. By (proof) state we mean a sequent, potentially coupled with additional information. The nodes of our proof tree will be proof states in this sense. Second, *rules* is a stream of rules for the prover to attempt to apply. Third, S is a set of well formed states (in our case simply the set of all states).

For the *RuleSystem* locale we must prove two things about these definitions. First, *eff-S*, that the set of well formed states S is closed under the *eff* relation on rules from the stream *rules*. Second, *enabled-R*, that no matter the proof state we have reached (in S), some rule in *rules* applies. In return we get the function *mkTree* which embodies our prover and a proof, *wf-mkTree*, that the tree produced by this prover is well formed. A tree is well formed (*wf*) when its children are well formed and the set of child states is *eff*-related to the node's state and applied rule.

For the *PersistentRuleSystem* locale, we must additionally prove *per*. This essentially states that rules do not interfere with each other: when we apply a rule, any other rules that were applicable before are still applicable. In return we get a theorem called *epath-completeness-Saturated*. An escape path (*epath*) is an infinite path in a well formed proof tree. Such a path is saturated (*Saturated*) when any rule which is enabled at some point on the path is eventually applied. Thus, this theorem states a completeness property for the *mkTree* function (on valid input): either it returns a well formed finite tree or a tree containing a saturated escape path (from which we can build a countermodel).

Table 3 The *Soundness* locale.

eff, *rules*	As in Table 1 but states are now called sequents.
structure	Set of models.
sat	Satisfaction predicate on sequents and models.
local-soundness	Proof that the validity of a sequent (as given by *sat* and *structure*) follows from the validity of its children (as given by *eff* and *rules*).
soundness	Proof that any finite, well formed tree has a valid root.

Table 4 The *RuleSystem-Code* locale.

eff	Effect *function* from a rule and a state to a finite set of resulting states.
rules	Stream of rules.
i.mkTree	Executable version of the *mkTree* function.

Table 3 covers the *Soundness* locale used to prove the soundness of resulting proof trees. Here, besides *eff* and *rules*, we must state a set of models, *structure*, and a satisfaction predicate, *sat*, on sequents and models. The locale then turns a local soundness proof, *local-soundness*, that validity of a sequent follows from validity of its children, into a global result, *soundness*, that any finite, well formed tree has a valid root.

Finally, to generate code we need to instantiate the locale *RuleSystem-Code* in Table 4, where *eff* must now be a deterministic relation, i.e. a function and *rules* is as before. In return we get an executable version of *mkTree* above, called *i.mkTree*.

RuleSystem-Code provides no guarantees on its own but we use the same underlying function in all four locales. We export this function to Haskell using Isabelle's (unverified) code generation, code lemmas and a few (unverified) custom code-printing facilities. This step moves us from a verified prover inside Isabelle to a prover in Haskell which is based on a verified prover, but which is not itself verified.

3 Prover

In this section we explain the design of the proof search procedure driving our prover. The procedure does not use the proof system of SeCaV directly, but introduces a new set of similar proof rules that apply to entire sequents at once. This obviates the need for the structural Ext rule, which is therefore not present. Additionally, we remove the Basic rule and let the prover close proof branches implicitly.

Before we can define what the rules do, we need a few auxiliary definitions. The function *generateNew* generates a function name that is fresh to a given list of terms. The function *subtermFms* computes the list of terms occurring in a list of functions. We define *subterms* as the list of all terms in a sequent, except that the list contains exactly *Fun 0 []* when it would otherwise be empty. This ensures that we always have some term to instantiate γ-formulas with. The function *sub* implements substitution in a standard way using de Bruijn indices. See the formalization [18] or the original SeCaV work [19] for details. The function *branchDone* computes whether a sequent is an axiom, i.e. whether the sequent contains both a formula and its negation. The prover uses this to determine when a branch of the proof tree is proven and can be closed.

We first define which "parts" of a single formula must be proven for a rule to apply:

```
definition parts :: tm list ⇒ rule ⇒ fm ⇒ fm list list where
  parts A r f ≡ (case (r, f) of
     (NegNeg, Neg (Neg p)) ⇒ [[p]]
   | (AlphaDis, Dis p q) ⇒ [[p, q]]
   | (BetaDis, Neg (Dis p q)) ⇒ [[Neg p], [Neg q]]
   | (DeltaExi, Neg (Exi p)) ⇒ [[Neg (sub 0 (Fun (generateNew A) []) p)]]
   | (GammaExi, Exi p) ⇒ [Exi p # map (λt. sub 0 t p) A]
   ⋮
   | - ⇒ [[f]])
```

We have omitted some similar cases here (and will continue to do so in the sequel; see the formalization for the full definitions). The result of applying a rule is a list of lists of formulas with an implicit conjunction between lists and disjunction between inner formulas. For instance, the parts of *Dis p q* under *AlphaDis* state that we must prove either p or q. The definition takes a parameter A, which should be a list of terms present on the proof branch. For δ-rules, a function which does not appear in A is generated (ensuring soundness), and for γ-rules, the quantifier is instantiated with every term in A (ensuring completeness). Note that if the rule and formula do not match, the result simply contains the original formula. This means that rules are always enabled, but that they do nothing to most formulas.

To construct a proof tree, we need a function that computes the result of applying a rule to (all formulas in) a sequent. This is done by the following function (@ appends two lists):

```
primrec children :: tm list ⇒ rule ⇒ sequent ⇒ sequent list where
  children - - [] = [[]]
| children A r (p # z) =
  (let hs = parts A r p; A' = remdups (A @ subtermFms (concat hs))
   in list-prod hs (children A' r z))
```

It first computes the effect of applying the rule to the first formula in the sequent (using the definition *parts*) and gives a name to the updated list of terms in the sequent (since δ- and γ-rules may introduce new terms). The function then goes through the rest of the sequent recursively, combining the generated child branches with the function *list-prod*:

```
primrec list-prod :: 'a list list ⇒ 'a list list ⇒ 'a list list where
  list-prod - [] = []
| list-prod hs (t # ts) = map (λh. h @ t) hs @ list-prod hs ts
```

The type variable $'a$ in the type signature means that the function works on lists of lists containing any type of elements.

It behaves in the following way (similar to the Cartesian product):

$$set\ (list\text{-}prod\ hs\ ts) = \{h\ @\ t\ |h\ t.\ h \in set\ hs \wedge t \in set\ ts\}$$

For β-rules, the end result is a list of 2^n child branches, where n is the number of β-formulas in the sequent. These branches are ordered such that they correspond to the branches one would have obtained by applying the corresponding SeCaV β-rule n times. For all other rules, the end result is a single child branch. The parameter A to *children* should again be a list of terms present on the proof branch. We should be clear that *children* does not apply rules recursively to sub-formulas, but only to the "top layer." If the application of a rule reveals a formula that this rule applies to again, this formula is left as is and only considered the next time *children* is applied to the sequent with that rule. For example, the result of calling *children* with the rule AlphaDis and the sequent containing only the formula Dis (Dis p q) r is Dis p q, r and not p, q, r.

The prover needs to ensure that bound variables are instantiated with all terms on the current branch when a γ-rule is applied. For this reason, we define the *state* in a proof tree node to be a pair consisting of a list of terms appearing on the branch and a sequent. The list of terms will be used to instantiate the parameter A in the definitions above.

We are now ready to define the effect of applying a proof rule to a proof state:

```
primrec effect :: rule ⇒ state ⇒ state fset where
  effect r (A, z) =
  (if branchDone z then {||} else
    fimage (λz'. (remdups (A @ subterms z @ subterms z'), z'))
    (fset-of-list (children (remdups (A @ subtermFms z)) r z)))
```

To fit the types of the framework, the function returns a finite set (*fset*) instead of a list. If the sequent is an axiom, the branch is proven, and the function returns an empty set of child nodes, closing the branch. Otherwise, the function converts the result of the *children* function to a finite set, and adds any new terms to the list of terms in each child node.

Having defined what rules do, we now need a *stream* of them (*rules* in Table 1). We, somewhat arbitrarily, define a list of rules in the order α, δ, β, γ and cycle it to obtain a stream. For efficiency, we could run, say, all α- and δ-rules to completion before branching with the β-rules, but this cannot be encoded in the simple stream of rules without further machinery: one could imagine having larger "meta-rules" corresponding to groups of SeCaV rules. This would give a notion of "phases" where we would first run all the rules in one group, then all the rules in the next group in the stream etc. For simplicity (see Section 6.4) we apply single rules in a fixed order. This also trivially ensures fairness.

3.1 Applying the framework

We are now ready to apply the abstract completeness framework to obtain the actual proof search procedure (cf. Section 2.2). First, we define a relational version of the *effect* of a rule, called *eff*. To use the framework, we need to prove three properties: that the set of well formed proof states is closed under *eff* (*eff-S*), that it is always possible to apply some rule (*enabled-R*), and that the rules that can be applied are still possible to apply after applying other rules (*per*). We do not need to restrict the set of well formed proof states, so the first property is trivial. Since all of our rules can always be applied (they simply do nothing if they do not match the sequent), the other two properties are also trivial. We can thus instantiate the framework with our effect relation and stream of rules. This allows us to define the prover using the *mkTree* function from the framework:

```
definition secavProver ≡ mkTree rules
```

This function takes a list of terms and a sequent, and applies the rules in the stream in order to build a proof tree with the given sequent at the root, using our *eff* relation to determine the children of each node. The list of terms is used to collect the terms that occur in the sequents on each branch and should initially be empty (in the exported prover, the function is wrapped in another function to ensure that the list of terms is empty).

We call the sequent at the root of this proof tree the *root sequent*:

```
abbreviation rootSequent t ≡ snd (fst (root t))
```

3.2 Making the prover executable

To actually make the prover executable, we need to specify that the stream of rules should be lazily evaluated, or the prover will never terminate. Additionally, we need to define the prover using the code interpretation of the framework to enable computation of some parts of the framework (cf. Table 4). After telling Isabelle how to translate operations on the *option* type to the *Maybe* type, this also allows us to export the prover to Haskell code.

We have implemented a few Haskell modules to drive the exported prover, and translate found proofs into the proof system of SeCaV. These modules are not formally verified, but the proofs generated in this manner can be verified by Isabelle. We have written an automated test suite that tests the unverified code for soundness and completeness by applying the prover to a number of valid formulas, then calling Isabelle to verify the generated proofs, and by applying the prover to a number of invalid formulas and confirming that it does not generate a proof (within 10 seconds). While these tests do not give us absolute certainty that the exported code and the hand-written Haskell modules are correct, they provide a reasonable amount of certainty when combined with the formal proofs of correctness of the proof search procedure within Isabelle.

4 Soundness

We use the abstract soundness framework (cf. Section 2.2) to prove that any sequent with a well formed and finite proof tree can be proved in SeCaV. It follows from the soundness of SeCaV that such sequents for which the prover terminates are semantically valid. The following lemma comprises the core of the result:

▶ **Lemma 1.** *If for all sequents z' in children A r z, we can derive $\vDash$ pre @ z', and the term list A contains all parameters of pre and z, then we can derive $\vDash$ pre @ z itself:*

assumes $\forall z' \in set\ (children\ A\ r\ z).\ (\vDash pre\ @\ z')$
and $paramss\ (pre\ @\ z) \subseteq paramsts\ A$
shows $\vDash pre\ @\ z$

Proof. By induction on z for arbitrary *pre* and A.

For the empty sequent, the thesis holds immediately as we get by assumption and the definition of *children* that we can derive $\vDash$ *pre*.

For the non-empty sequent with formula p as head and z as tail we have the following induction hypothesis (for any *pre* and A):

then have *ih*: $\forall z' \in set\ (children\ A\ r\ z).\ (\vDash pre\ @\ z') \Longrightarrow (\vDash pre\ @\ z)$
if $paramss\ (pre\ @\ z) \subseteq paramsts\ A$ **for** $pre\ A$

We abbreviate the term list that the prover actually recurses on as *?A*. From the first assumption and the definition of *list-prod* we then have (*):

$$\forall hs \in set\ (parts\ A\ r\ p).\ \forall ts \in set\ (children\ ?A\ r\ z).\ (\vDash pre\ @\ hs\ @\ ts)$$

The proof continues by examining the possible cases for *parts*.

Take first the case where $r = AlphaDis$ and $p = Dis\ q\ r$. Then (*) states that we can derive $\vDash pre\ @\ q\ \#\ r\ \#\ z'$ for all z' in *children ?A r z*. We apply the induction hypothesis at *pre* extended with q and r, which is allowed since they are subformulas of p. We then get the derivation $\vDash pre\ @\ q\ \#\ r\ \#\ z$. By the Ext and AlphaDis rules from SeCaV we obtain the desired derivation $\vDash pre\ @\ Dis\ q\ r\ \#\ z$.

The remaining α- and β-cases are similar. In the δ-cases we prove that the constant used by the prover is new to the sequent, as required by the SeCaV δ-rules.

In the γ-cases we get a derivation that includes both the γ-formula and all instances of it using terms from the list A. Here we induct on A to generalize each instance into the corresponding γ-formula and use Ext to contract this γ-formula with the existing occurrence.

When *parts A r p* returns *p*, the thesis holds from (*) and the induction hypothesis. ◀

We only need *pre* in the above lemma to make the induction hypothesis strong enough for the proof, so we can instantiate it afterwards.

▶ **Corollary 2** (Proof tree to SeCaV). *We derive a sequent from derivations of its children:*

assumes $\forall z' \in set\ (children\ A\ r\ z).\ (\vdash z')$ **and** $paramss\ z \subseteq paramsts\ A$
shows $\vdash z$

We obtain the following soundness theorem from the abstract soundness framework.

▶ **Theorem 3** (Prover soundness wrt. SeCaV). *The root sequent of any finite, well formed proof tree has a derivation in SeCaV:*

assumes *tfinite t* **and** *wf t*
shows $\vdash$ *rootSequent t*

5 Completeness

The completeness proof is heavily based on the abstract completeness framework. As noted in Section 2.2, however, the framework only takes us so far. First, we duplicate the output of Table 2, since the *mkTree* function is unhelpfully abstracted away by an existential quantifier. This could easily be changed in the framework and should be considered for the next release.

▶ **Lemma 4** (Prover cases). *The proof tree* generated by the prover *is either finite and well formed or there exists a saturated escape path with our initial state as root:*

defines $t \equiv secavProver\ (A, z)$
shows $(fst\ (root\ t) = (A, z) \wedge wf\ t \wedge tfinite\ t) \vee$
$(\exists\ steps.\ fst\ (shd\ steps) = (A, z) \wedge epath\ steps \wedge Saturated\ steps)$

In the first case, the sequent has a proof (cf. Section 4). In the second case, we need to build a countermodel from the saturated escape path to contradict validity of the sequent. The rest of this section does exactly that. Inspired by Ben-Ari [2] and Ridge [38], we start off by giving a definition of Hintikka sets over a restricted set of terms (Section 5.1). We show that the set of formulas on saturated escape paths fulfill all Hintikka requirements when we take the set of terms to be the terms on the path (Section 5.2). We then define a countermodel for any formula in such a set using a new semantics that bounds quantifiers by an explicit set rather than by types alone (Section 5.3). Finally we tie these results together to show that the prover terminates for all sequents that are valid under our new semantics (Section 5.4). In Section 6.1 we use existing results to prove completeness of the prover wrt. the SeCaV semantics.

locale *Hintikka* =
fixes H :: *fm set*
assumes
Basic: $Pre\ n\ ts \in H \Longrightarrow Neg\ (Pre\ n\ ts) \notin H$ **and**
AlphaDis: $Dis\ p\ q \in H \Longrightarrow p \in H \wedge q \in H$ **and**
BetaDis: $Neg\ (Dis\ p\ q) \in H \Longrightarrow Neg\ p \in H \vee Neg\ q \in H$ **and**
GammaExi: $Exi\ p \in H \Longrightarrow \forall t \in terms\ H.\ sub\ 0\ t\ p \in H$ **and**
DeltaExi: $Neg\ (Exi\ p) \in H \Longrightarrow \exists t \in terms\ H.\ Neg\ (sub\ 0\ t\ p) \in H$ **and**
⋮
Neg: $Neg\ (Neg\ p) \in H \Longrightarrow p \in H$

Figure 4 Abridged list of requirements for a set of formulas H to be a Hintikka set.

5.1 Hintikka

First, by the *terms* of a set of formulas H we mean the following:

definition $terms\ H \equiv if\ (\bigcup p \in H.\ set\ (subtermFm\ p)) = \{\}\ then\ \{Fun\ 0\ []\}$
$else\ (\bigcup p \in H.\ set\ (subtermFm\ p))$

This set contains an arbitrary (but fixed) constant, *Fun 0 []*, when H itself contains no terms. Otherwise it contains all subterms of all formulas in H.

Figure 4 contains an abridged definition of a Hintikka set H. Here, we use a locale slightly differently to the previous ones, in that we have specify no conclusions, only premises: the formula set H and the requirements *Basic*, *AlphaDis*, etc. The omitted requirements are similar to the ones shown. This use simply allows us to assume *Hintikka H* in a theorem and know that the set H then fulfills the stated requirements. Similarly, we can prove that a set H is *Hintikka* by proving that it fulfills the requirements. It is important to note that in the γ- and δ-cases, the quantifiers only range over the *terms* of H.

5.2 Saturated escape paths are Hintikka

The following definition forgets all structure of a path and reduces it to a set of formulas:

definition $tree\text{-}fms\ steps \equiv \bigcup ss \in sset\ steps.\ set\ (pseq\ ss)$

The function *sset* returns the set of steps and *pseq* extracts the sequent from each.

Given a saturated escape path *steps*, we want to prove that *tree-fms steps* is a Hintikka set. For instance, if $Dis\ p\ q$ appears on the path, then both p and q should too. The prover is designed to make this property of its proof trees as evident as possible: formulas unaffected by a given rule are easily shown to be preserved by the application of that rule and any rule immediately applies to all its affected formulas, regardless of their position in the sequent.

We will need a number of intermediate results.

5.2.1 Unaffected formulas

We define the predicate *affects* to hold for a rule and a formula, when that rule does not preserve the formula (thus no rule *affects* a γ-formula, since the γ-rules of the prover, unlike those of SeCaV, preserve the original formula). For instance, $affects\ AlphaDis\ (Dis\ p\ q)$ holds while $affects\ BetaCon\ (Dis\ p\ q)$ does not.

We then prove the following key preservation lemma:

▶ **Lemma 5** (*effect* preserves unaffected formulas). *Assume formula p occurs in sequent z and the rule r does not affect p. Then p also occurs in all children of z as given by effect ($|\in|$ denotes membership of a finite set):*

assumes $p \in$ *set* z **and** $\neg$ *affects* $r\ p$ **and** $(B, z') \ |\in|$ *effect* $r\ (A, z)$
shows $p \in$ *set* z'

Proof. The function *parts* preserves unaffected formulas (proof by cases) so *children* does as well (proof by induction on the sequent) and thus *effect* does too. ◀

We lift this to escape paths:

▶ **Lemma 6** (Escape paths preserve unaffected formulas). *Assume formula p occurs in some sequent at the head of an escape path which consists of a prefix pre, where none of the rules affect p, and a suffix suf. Then p occurs at the head of suf:*

assumes $p \in$ *set* (*pseq* (*shd steps*)) **and** *epath steps* **and** *steps* $=$ *pre* @− *suf* **and**
list-all (*not* (λ*step*. *affects* (*snd step*) *p*)) *pre*
shows $p \in$ *set* (*pseq* (*shd suf*))

Next, notice the following property of streams:

▶ **Lemma 7** (Eventual prefix). *When a property P eventually holds of a stream, then the stream is comprised of a prefix of n (possibly zero) elements for which P does not hold and then a suffix that starts with an element for which P does hold:*

assumes *ev* (*holds P*) *xs*
shows ∃ *n*. *list-all* (*not P*) (*stake n xs*) ∧ *holds P* (*sdrop n xs*)

Saturation states that a rule is eventually applied and Lemmas 6 and 7 combine to state that any affected formulas are preserved until then.

5.2.2 Affected formulas

Knowing that formulas are preserved as desired, we need to know that they are broken down as desired. The following lemma (proof omitted here) states this in general via *parts*:

▶ **Lemma 8** (Parts in effect). *For any formula p in a sequent z, the effect of rule r on z includes some part of r's effect on p:*

assumes $p \in$ *set* z **and** $(B, z') \ |\in|$ *effect* $r\ (A, z)$
shows ∃ *C xs*. *set A* ⊆ *set C* ∧ *xs* ∈ *set* (*parts C r p*) ∧ *set xs* ⊆ *set* z'

This is easier to understand when we specialize the rule and the formula:

▶ **Corollary 9.** *Example effect of the NegNeg rule on a double-negated formula p:*

corollary *Neg* (*Neg p*) ∈ *set z* ⟹ $(B, z') \ |\in|$ *effect NegNeg* (A, z) ⟹ $p \in$ *set* z'

5.2.3 Hintikka requirements

We then need to prove the following:

▶ **Theorem 10** (Hintikka escape paths). *Saturated escape paths fulfill all Hintikka requirements:*

assumes *epath steps* **and** *Saturated steps*
shows *Hintikka* (*tree-fms steps*)

Proof. This boils down to proving each requirement of Figure 4 (and those omitted there). We give a couple of examples and refer to the formalization for the full details.

For *Basic*, assume towards a contradiction that both a predicate and its negation appear on the branch. By preservation of formulas (Lemma 6), both appear in the same sequent at some point. But then *branchDone* holds for that sequent, so it has no children and the branch would terminate. This contradicts that escape paths are infinite, so *Basic* must hold.

For *AlphaDis*, assume that *Dis p q* appears on the branch. Then it appears at some step n. By saturation of the escape path, *AlphaDis* is eventually applied at some (earliest) step $n + k$. By Lemma 6, *Dis p q* is preserved until then. So by the effect of rule *AlphaDis*, both p and q appear at step $n + k + 1$. The cases for the β- and δ-requirements are very similar.

For *GammaExi* assume that *Exi p* occurs at step n. We need to show that it is instantiated with all terms that (eventually) appear on the branch. Fix an arbitrary such term t. There must be some point m where t appears in a sequent. Thus at every point greater than m, term t appears in the term list which is part of the proof state. By saturation, at some step greater than $n + m + 1$, rule *GammaExi* is applied. The formula *Exi p* is preserved until this stage (Lemma 6) and the term list only grows, so t is too. Thus, at the next step, *sub 0 t p* occurs on the branch as desired. ◀

5.3 Countermodel

We need to build a countermodel for any formula in a Hintikka set to contradict the validity of any formula on a saturated escape path. We do this in the usual term model with a (bounded) Herbrand interpretation. Unfortunately, we cannot build a countermodel in the original semantics where the universe is specified as a type, since we cannot form the type of terms in a given Hintikka set (the *typedef* command does not support free variables). Instead, we introduce a custom bounded semantics.

5.3.1 Bounded semantics

The bounded semantics is exactly like the usual semantics (cf. Figure 2) except for an extra argument u, standing for the universe, which bounds the range of the quantifiers in the following cases:

```
| usemantics u e f g (Exi p) = (∃ x ∈ u. usemantics u (SeCaV.shift e 0 x) f g p)
| usemantics u e f g (Uni p) = (∀ x ∈ u. usemantics u (SeCaV.shift e 0 x) f g p)
```

This leads to the following natural requirements on environments e and function denotations f, namely that they must stay inside u:

```
definition is-env u e ≡ ∀ n. e n ∈ u
definition is-fdenot u f ≡ ∀ i l. list-all (λx. x ∈ u) l ⟶ f i l ∈ u
```

In general, we only consider environments and function denotations that satisfy these requirements and call them (and any model based on them) *well formed*. When $u = UNIV$, we do not actually bound the quantifiers and the two semantics coincide.

The SeCaV proof system (cf. Figure 3) is sound for the bounded semantics too.

▶ **Theorem 11** (SeCaV is sound for the bounded semantics)**.** *Given a SeCaV derivation of sequent z and a well formed model, some formula p in z is satisfied in that model:*

```
assumes ⊩ z and is-env u e and is-fdenot u f
shows ∃ p ∈ set z. usemantics u e f g p
```

Proof. The proof closely resembles the original soundness proof (cf. [19]). ◀

We abbreviate validity of a sequent in the bounded semantics as *uvalid*:

abbreviation *uvalid* $z \equiv \forall\, u\ (e :: nat \Rightarrow tm)\ f\ g.$ *is-env* $u\ e \longrightarrow$ *is-fdenot* $u\ f \longrightarrow$
$(\exists\, p \in set\ z.$ *usemantics* $u\ e\ f\ g\ p)$

Namely, for all universes and well formed models, some formula in the sequent is satisfied in the bounded semantics at that universe by that model.

5.3.2 Model construction

Our countermodel is given by a bounded Herbrand interpretation where terms are interpreted as themselves when they appear in the universe *terms H* and as an arbitrary term otherwise.

▶ **Definition 12** (Countermodel induced by Hintikka set S). *We abbreviate the model as M S:*

abbreviation $E\ S\ n \equiv$ *if Var* $n \in$ *terms* S *then Var* n *else SOME* $t.\ t \in$ *terms* S
abbreviation $F\ S\ i\ l \equiv$ *if Fun* $i\ l \in$ *terms* S *then Fun* $i\ l$ *else SOME* $t.\ t \in$ *terms* S
abbreviation $G\ S\ n\ ts \equiv Neg\ (Pre\ n\ ts) \in S$
abbreviation $M\ S \equiv$ *usemantics* $(terms\ S)\ (E\ S)\ (F\ S)\ (G\ S)$

The definition of G is what makes this a countermodel rather than a model: a predicate is satisfied exactly when its negation is present in the Hintikka set.

Importantly, these definitions are *well formed*:

▶ **Lemma 13** (Well formed countermodel). *Definition 12 is well formed:*

shows *is-env* $(terms\ S)\ (E\ S)$
shows *is-fdenot* $(terms\ S)\ (F\ S)$

Proof. By the construction of E and F and the nonemptiness of *terms S*. ◀

▶ **Theorem 14** (Model existence). *The given model falsifies any formula p in Hintikka set S:*

assumes *Hintikka S*
shows $(p \in S \longrightarrow \neg\ M\ S\ p) \wedge (Neg\ p \in S \longrightarrow M\ S\ p)$

Proof. By induction on the size of the formula p (substitution instances are smaller than the quantified formulas they arise from). The second part of the thesis is needed when the Hintikka requirements concern negated formulas. We show a few cases here and refer to the formalization for the full details. The cases omitted here are similar to those shown.

Assume $p = Pre\ n\ ts$ occurs in S. We need to show that the given model falsifies p. Since *terms S* is downwards closed by construction, *ts* is interpreted as itself by the bounded Herbrand interpretation. Moreover, by the *Basic* requirement, we know that $Neg\ p$ is not in S and is therefore satisfied. Thus, p is falsified.

Assume $p = Dis\ q\ r$ occurs negated in S. Then by the *BetaDis* requirement, either $Neg\ q$ or $Neg\ r$ occurs in S. The induction hypothesis applies to these, so p is satisfied as desired.

Assume $p = Uni\ q$ occurs in S. By the *DeltaUni* requirement, so does some instance $sub\ 0\ t\ q$ for a term t in *terms S*. By the induction hypothesis, this is falsified by $M\ S$, and by its origin, t is interpreted as itself. Thus, we have a counterexample that falsifies p.

Assume $p = Exi\ q$ occurs in S. By the *GammaExi* requirement, so do all instances using terms from S. Thus, these are all falsified by the model. These terms from S are interpreted as themselves by definition so we have no witness for p in *terms S* and $M\ S$ falsifies it. ◀

We note that the above proof works for open and closed formulas alike because we consider both bound and free variables to be subterms of a formula.

5.4 Result

We start off by proving completeness for *uvalid* sequents. We need to relate these to saturated escape paths.

▶ **Lemma 15** (Saturated escape paths contradict uvalidity). *A sequent z with a saturated escape path, steps, cannot be uvalid:*

> **assumes** *fst* (*shd steps*) = (A, z) **and** *epath steps* **and** *Saturated steps*
> **shows** ¬ *uvalid z*

Proof. Assume towards a contradiction that z is *uvalid*. By Theorem 10 the formulas on *steps* form a Hintikka set S. Every formula p in z also occurs in S, so by Theorem 14, the well formed model $M\ S$ (Lemma 13) falsifies all of them. This contradicts the uvalidity of z. ◀

This leads to completeness for *uvalid* sequents:

▶ **Theorem 16** (Completeness wrt. *uvalid*). *The prover terminates for uvalid sequents:*

> **assumes** *uvalid z*
> **defines** $t \equiv$ *secavProver* (A, z)
> **shows** *fst* (*root t*) = $(A, z) \wedge$ *wf t* $\wedge$ *tfinite t*

Proof. From the abstract framework (Lemma 4), either the thesis holds or a saturated escape path exists for our sequent, but assumed uvalidity and Lemma 15 contradict the latter. ◀

▶ **Corollary 17** (Completeness wrt. SeCaV). *Termination for sequents derivable in SeCaV:*

> **assumes** $\Vdash z$
> **defines** $t \equiv$ *secavProver* (A, z)
> **shows** *fst* (*root t*) = $(A, z) \wedge$ *wf t* $\wedge$ *tfinite t*

Proof. By the soundness of SeCaV (Theorem 11) and Theorem 16 for *uvalid* sequents. ◀

6 Results and discussion

We have presented an automated theorem prover for the Sequent Calculus Verifier system. The prover is capable of proving a number of selected exercise formulas very quickly, including formulas which are quite difficult for humans to prove. The prover does have some limitations, mostly related to performance and length of the generated proofs, since our proof search procedure is not very optimized for either of these metrics. In particular, our prover always instantiates quantified formulas with all terms in the sequent and breaks down all formulas as much as possible, even when some formulas are "obviously" irrelevant to the proof.

6.1 Summary of theorems

We have proven soundness and completeness of the proof search procedure with regards to the proof system of SeCaV (see Figure 3). For soundness, this was done directly (in Theorem 3), while we took a detour through our notion of a bounded semantics to prove completeness (in Theorems 11 and 16, which lead to Corollary 17). To justify the introduction of our bounded semantics, we can use the existing soundness and completeness theorems of the SeCaV proof system [19] and our results to prove that validity in the two semantics coincide. Additionally, a number of easy corollaries further linking the prover, the proof system and the two semantics follow from our results, and have been collected in Figure 5. In the figure, the interpretations are implicitly universally quantified and for the bounded semantics we only consider well formed interpretations.

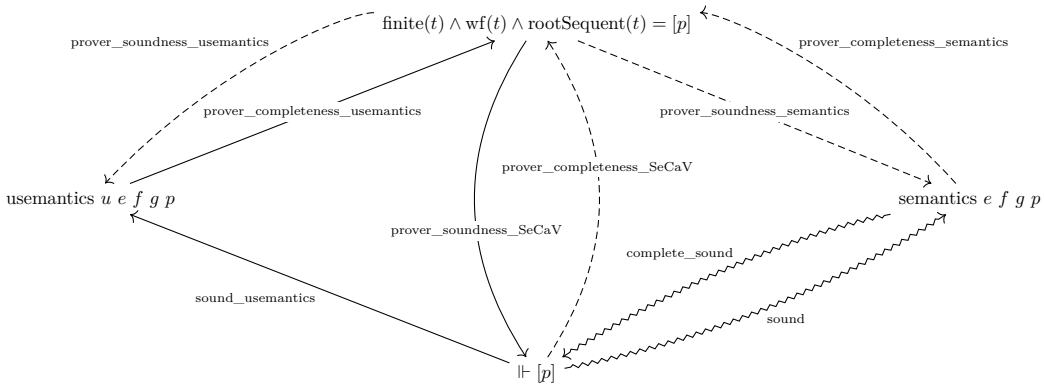


Figure 5 Overview of our results. Solid arrows represent our main contributions, squiggly arrows represent theorems of the existing SeCaV system, and dashed arrows represent easy corollaries.

6.2 Example proofs

Famously, we must beware of a program that has only been proven correct, but not tested. To demonstrate that the automated theorem prover works, we examine some simple generated proofs. The prover generates proofs in the SeCaV Unshortener format: first comes the formula to be proven, then the names of proof rules to apply and the resulting sequent after each application, with each formula in a sequent on its own line. Arguments to predicates and functions are given in square brackets and parentheses are used to disambiguate formulas.

We start with perhaps the simplest possible classical example, that $\neg p \vee p$. Figure 6a shows the proof generated by the prover. This is the shortest possible proof of the formula in the SeCaV system, and the prover is thus on par with a human in this very simple case.

The next example is $\neg p(a) \vee \exists x.p(x)$. Figure 6b contains the generated proof. It can be shortened since the quantified formula only needs to be instantiated once, by a. However, the prover always duplicates a γ-formula before instantiating it with *all* terms on the branch.

6.3 Verification challenges

While verifying the prover, we discovered that our initial version was unsound due to a missing update of the term list when applying (multiple) δ-rules to a sequent. The attempted soundness proof failed in exactly this case, pointing us directly to the issue. Thus, the formal verification caught a critical flaw that we had missed in our testing and helped us fix it.

We have designed the prover to be easily verified and it mostly was. Especially the abstract framework worked well for our novel case with a deterministic prover for first-order logic. One obstacle, however, was in using a type to represent the domain in the SeCaV semantics (cf. Figure 2). To build the countermodel, we need the domain to contain only the terms on the saturated escape path, but we cannot form this type, which depends on a local variable, in Isabelle/HOL. Here we would benefit from Isabelle integration of the work by Kunčar and Popescu [25] which adds exactly this capability to higher-order logic. Instead we introduced the bounded semantics ("the set-based relativization" in their terminology [25]) and proved a new soundness result for it (cf. Section 5.3.1). Otherwise the largest issue was dealing with substitutions using de Bruijn indices. We are excited to see how recent work by Blanchette et al. [7] for reasoning about syntax with bindings improves matters in this area.

```
Dis (Neg p) p

AlphaDis
  Neg p
  p
Ext
  p
  Neg p
Basic
```

(a) $\neg p \vee p$.

```
Dis (Neg (p [a]))
    (Exi (p [0]))

AlphaDis
  Neg (p [a])
  Exi (p [0])
Ext
  Exi (p [0])
  Exi (p [0])
  Neg (p [a])
GammaExi[a]
  p [a]
  Exi (p [0])
  Neg (p [a])
```

(b) $\neg p(a) \vee \exists x.p(x)$.

```
Ext
  Exi (p [0])
  Exi (p [0])
  Neg (p [a])
  p [a]
GammaExi[0]
  p [0]
  Exi (p [0])
  Neg (p [a])
  p [a]
Ext
  p [a]
  Neg (p [a])
  Exi (p [0])
  p [0]
Basic
```

Figure 6b continued.

Figure 6 Proofs generated by the prover in SeCaV Unshortener format.

6.4 Limitations and future work

There are a number of limitations and possibilities for optimization in the proof search itself. Most importantly, the focus of the procedure is on completeness, not performance. Our prover is much slower than state-of-the-art provers such as Vampire [24], but our goal was not to compete on speed, but simply to show that formal verification of provers with advanced features such as generation of proof certificates and support for functions is possible. The prover also cannot output counterexamples, even though these can be detected in some cases: our prover simply never terminates on invalid formulas.

We believe that the approach used for our prover is extendable to more sophisticated and optimized proof search procedures, albeit with considerably more work needed to formally verify them. The most obvious opportunity for optimization is controlling the order of proof rules. In systems with unordered sequents, it is generally better to apply as many α-rules as possible before applying β-rules to avoid duplicating work, but the prover simply applies rules in a fixed order. As mentioned in Section 3, this optimization can be done by working with "meta-rules" corresponding to groups of SeCaV rules such that a meta-rule e.g. applies as many α-rules as possible before continuing to the next "phase" of the proof. We have attempted to implement this, but found that it complicates the proofs considerably since this idea makes it much harder to determine when a proof rule is actually applied. In the proof of fairness and the proof that the formulas on saturated escape paths form Hintikka sets, we need to know that certain formulas are preserved until proof rules are eventually applied to them. By introducing phases in the proof, proving this becomes much more difficult, since we then need to prove that each phase actually ends (requiring some measure which depends on the specific sequents in question), and to locate each rule within the meta-rule it is part of. We thus leave optimizations in this vein as future work. We note that, since the SeCaV system requires application of the Ext rule to permute sequents, and proof rules only apply to the first formula in a sequent, the optimization described above may not always reduce the number of SeCaV proof steps needed to prove a formula, and some heuristics would probably be needed to produce reasonably short proofs in all cases.

Another optimization could be to only support closed formulas and thus reduce the number of subterms of a given formula. For our current Herbrand interpretation, we need variables to be subterms, but if we only considered closed terms, we could do away with this.

The length of proofs could also be optimized by performing more post-processing of the found proofs, for example by removing unnecessary instantiations or rule applications that do not contribute to proving a branch. This would not improve the performance in the sense that the prover would still spend the same amount of time finding the proof, but it could reduce the length of some proofs significantly. The proof trees generated by the prover already require some (unverified) post-processing to obtain proofs in the SeCaV system. It would be interesting to move these steps from Haskell into Isabelle/HOL and extend the proofs to cover them.

7 Conclusion

We have designed, implemented and verified an automated theorem prover for first-order logic with functions in Isabelle/HOL. We have used an existing framework in a novel way to get us part of the way towards completeness and extended existing techniques on countermodels over restricted domains to reach our destination. We build on the existing SeCaV system and contribute an automatic way of finding derivations to the project. Thus, we have demonstrated the utility of Isabelle/HOL for implementing and verifying executable software and the strength of its libraries in doing so. Our prover handles the full syntax of first-order logic with functions and constructs human-readable proof certificates in a sequent calculus. We hope our work inspires others to verify more sophisticated provers in the same vein.

References

1 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.

2 Mordechai Ben-Ari. *Mathematical Logic for Computer Science*. Springer London, 2012. doi:10.1007/978-1-4471-4129-7.

3 A. Bentkamp, J. Blanchette, S. Tourret, and P. Vukmirović. Superposition for Full Higher-order Logic. In A. Platzer and G. Sutcliffe, editors, *Proceedings of the 28th International Conference on Automated Deduction*, number 12699 in Lecture Notes in Computer Science, pages 396–412. Springer-Verlag, 2021. doi:10.1007/978-3-030-79876-5_23.

4 Stefan Berghofer. First-order logic according to Fitting. *Archive of Formal Proofs*, August 2007. Formal proof development. URL: https://isa-afp.org/entries/FOL-Fitting.html.

5 Jasmin Christian Blanchette. Formalizing the metatheory of logical calculi and automatic provers in Isabelle/HOL (invited talk). In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019*, pages 1–13. ACM, 2019. doi:10.1145/3293880.3294087.

6 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(1-4):333–365, 2018. doi:10.1007/s10817-018-9455-7.

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

8 Jasmin Christian Blanchette and Andrei Popescu. Mechanizing the metatheory of Sledgehammer. In Pascal Fontaine, Christophe Ringeissen, and Renate A. Schmidt, editors, *Frontiers of Combining Systems*, pages 245–260, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-40885-4_17.

9 Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. Abstract completeness. *Archive of Formal Proofs*, April 2014. Formal proof development. URL: https://isa-afp.org/entries/Abstract_Completeness.html.

10 Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. Unified classical logic completeness. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning*, pages 46–60, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-08587-6_4.

11 Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. Soundness and completeness proofs by coinductive methods. *Journal of Automated Reasoning*, 58:149–179, 2017. doi:10.1007/s10817-016-9391-3.

12 Joachim Breitner. Visual theorem proving with the Incredible Proof Machine. In J. Blanchette and S. Merz, editors, *Interactive Theorem Proving*, volume ITP 2016. Springer, Cham, 2016. doi:10.1007/978-3-319-43144-4_8.

13 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-78800-3_24.

14 Mathias Fleury. Optimizing a verified SAT solver. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods – 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*, volume 11460 of *Lecture Notes in Computer Science*, pages 148–165. Springer, 2019. doi:10.1007/978-3-030-20652-9_10.

15 Asta Halkjær From. Formalized soundness and completeness of epistemic logic. In Alexandra Silva, Renata Wassermann, and Ruy J. G. B. de Queiroz, editors, *Logic, Language, Information, and Computation – 27th International Workshop, WoLLIC 2021, Virtual Event, October 5-8, 2021, Proceedings*, volume 13038 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2021. doi:10.1007/978-3-030-88853-4_1.

16 Asta Halkjær From, Frederik Krogsdal Jacobsen, and Jørgen Villadsen. SeCaV: A sequent calculus verifier in Isabelle/HOL. In Mauricio Ayala-Rincon and Eduardo Bonelli, editors, Proceedings 16th *Logical and Semantic Frameworks with Applications,* Buenos Aires, Argentina (Online), 23rd – 24th July, 2021, volume 357 of *Electronic Proceedings in Theoretical Computer Science*, pages 38–55. Open Publishing Association, 2022. doi:10.4204/EPTCS.357.4.

17 Asta Halkjær From. Epistemic logic: Completeness of modal logics. *Archive of Formal Proofs*, October 2018. Formal proof development. URL: https://isa-afp.org/entries/Epistemic_Logic.html.

18 Asta Halkjær From and Frederik Krogsdal Jacobsen. A sequent calculus prover for first-order logic with functions. *Archive of Formal Proofs*, January 2022. Formal proof development. URL: https://isa-afp.org/entries/FOL_Seq_Calc2.html.

19 Asta Halkjær From, Alexander Birch Jensen, Anders Schlichtkrull, and Jørgen Villadsen. Teaching a formalized logical calculus. *Electronic Proceedings in Theoretical Computer Science*, 313:73–92, 2020. doi:10.4204/EPTCS.313.5.

20 Asta Halkjær From, Jørgen Villadsen, and Patrick Blackburn. Isabelle/HOL as a metalanguage for teaching logic. *Electronic Proceedings in Theoretical Computer Science*, 328:18–34, October 2020. doi:10.4204/eptcs.328.2.

21 Frederik Krogsdal Jacobsen. Formalization of logical systems in Isabelle: An automated theorem prover for the Sequent Calculus Verifier. Master's thesis, Technical University of Denmark, June 2021. URL: https://findit.dtu.dk/en/catalog/2691928304.

22 Alexander Birch Jensen, John Bruntse Larsen, Anders Schlichtkrull, and Jørgen Villadsen. Programming and verifying a declarative first-order prover in Isabelle/HOL. *AI Communications*, 31(3):281–299, 2018. doi:10.3233/AIC-180764.

23 Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. Locales – A sectioning concept for Isabelle. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin-Mohring, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*, volume 1690 of *Lecture Notes in Computer Science*, pages 149–166. Springer, 1999. doi:10.1007/3-540-48256-3_11.

24 Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. *Lecture Notes in Computer Science*, 8044, 2013. doi:0.1007/978-3-642-39799-8_1.

25 Ondrej Kunčar and Andrei Popescu. From types to sets by local type definition in higher-order logic. *Journal of Automated Reasoning*, 62(2):237–260, 2019. doi:10.1007/s10817-018-9464-6.

26 Peter Lammich. The GRAT tool chain. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing – SAT 2017*, pages 457–463, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-66263-3_29.

27 Peter Lammich. Efficient verified (UN)SAT certificate checking. *Journal of Automated Reasoning*, 64(3):513–532, 2020. doi:10.1007/s10817-019-09525-z.

28 Stephane Lescuyer. *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq*. Phd thesis, Université Paris Sud – Paris XI, January 2011. URL: https://tel.archives-ouvertes.fr/tel-00713668.

29 Filip Marić. Formal verification of modern SAT solvers. *Archive of Formal Proofs*, July 2008. Formal proof development. URL: https://isa-afp.org/entries/SATSolverVerification.html.

30 Filip Marić. Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. *Theoretical Computer Science*, 411(50):4333–4356, 2010. doi:10.1016/j.tcs.2010.09.014.

31 Filip Marić, Mirko Spasić, and René Thiemann. An incremental simplex algorithm with unsatisfiable core generation. *Archive of Formal Proofs*, August 2018. Formal proof development. URL: https://isa-afp.org/entries/Simplex.html.

32 Julius Michaelis and Tobias Nipkow. Propositional proof systems. *Archive of Formal Proofs*, June 2017. Formal proof development. URL: https://isa-afp.org/entries/Propositional_Proof_Systems.html.

33 Julius Michaelis and Tobias Nipkow. Formalized Proof Systems for Propositional Logic. In Andreas Abel, Fredrik Nordvall Forsberg, and Ambrus Kaposi, editors, *23rd International Conference on Types for Proofs and Programs (TYPES 2017)*, volume 104 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.TYPES.2017.5.

34 Dominique Pastre. MUSCADET 2.3: A knowledge-based theorem prover based on natural deduction. *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2083:685–689, 2001. doi:10.1007/3-540-45744-5_56.

35 Francis Jeffry Pelletier. Automated natural deduction in THINKER. *Studia Logica*, 60(1):3–43, 1998. doi:10.1023/A:1005035316026.

36 Nicolas Peltier. Propositional resolution and prime implicates generation. *Archive of Formal Proofs*, March 2016. Formal proof development. URL: https://isa-afp.org/entries/PropResPI.html.

37 Nicolas Peltier. A variant of the superposition calculus. *Archive of Formal Proofs*, September 2016. Formal proof development. URL: https://isa-afp.org/entries/SuperCalc.html.

38 Tom Ridge and James Margetson. A mechanically verified, sound and complete theorem prover for first order logic. *Lecture Notes in Computer Science*, 3603:294–309, 2005. doi:10.1007/11541868_19.

39 Anders Schlichtkrull. Formalization of the resolution calculus for first-order logic. *Journal of Automated Reasoning*, 61(1–4):455–484, 2018. doi:10.1007/s10817-017-9447-z.

40 Anders Schlichtkrull, Jasmin Christian Blanchette, and Dmitriy Traytel. A verified prover based on ordered resolution. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, pages 152–165, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293880.3294100.

41 Anders Schlichtkrull, Jasmin Christian Blanchette, Dmitriy Traytel, and Uwe Waldmann. Formalizing Bachmair and Ganzinger's ordered resolution prover. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *Automated Reasoning*, pages 89–107, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-94205-6_7.

42 Anders Schlichtkrull and Jørgen Villadsen. Paraconsistency. *Archive of Formal Proofs*, December 2016. Formal proof development. URL: https://isa-afp.org/entries/Paraconsistency.html.

43 Natarajan Shankar and Marc Vaucher. The mechanical verification of a DPLL-based satisfiability solver. *Electronic Notes in Theoretical Computer Science*, 269:3–17, 2011. Proceedings of the Fifth Logical and Semantic Frameworks, with Applications Workshop (LSFA 2010). doi:10.1016/j.entcs.2011.03.002.

44 Raymond M. Smullyan. *First-order logic*. Dover Publications, 1995.

45 Mirko Spasić and Filip Marić. Formalization of incremental simplex algorithm by stepwise refinement. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods*, pages 434–449, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-32759-9_35.

46 Jørgen Villadsen, Asta Halkjær From, Alexander Birch Jensen, and Anders Schlichtkrull. Interactive theorem proving for logic and information. In Roussanka Loukanova, editor, *Natural Language Processing in Artificial Intelligence — NLPinAI 2021*, pages 25–48, Cham, 2022. Springer International Publishing. doi:10.1007/978-3-030-90138-7_2.

47 Jørgen Villadsen and Frederik Krogsdal Jacobsen. Using Isabelle in two courses on logic and automated reasoning. In João F. Ferreira, Alexandra Mendes, and Claudio Menghi, editors, *Formal Methods Teaching*, pages 117–132, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-91550-6_9.

48 Jørgen Villadsen and Anders Schlichtkrull. Formalizing a paraconsistent logic in the Isabelle proof assistant. In Abdelkader Hameurlain, Josef Küng, Roland Wagner, and Hendrik Decker, editors, *Transactions on Large-Scale Data- and Knowledge-Centered Systems XXXIV: Special Issue on Consistency and Inconsistency in Data-Centric Applications*, pages 92–122, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg. doi:10.1007/978-3-662-55947-5_5.

49 Jørgen Villadsen, Andreas Halkjær From, and Anders Schlichtkrull. Natural deduction assistant (NaDeA). *Electronic Proceedings in Theoretical Computer Science*, 290(290):14–29, 2019. doi:10.4204/EPTCS.290.2.

50 Jørgen Villadsen, Anders Schlichtkrull, and Andreas Halkjær From. A verified simple prover for first-order logic. *CEUR Workshop Proceedings*, 2162:88–104, 2018. URL: http://ceur-ws.org/Vol-2162/#paper-08.

Formalizing the Ring of Adèles of a Global Field

María Inés de Frutos-Fernández
Imperial College London, UK

Abstract

The ring of adèles of a global field and its group of units, the group of idèles, are fundamental objects in modern number theory. We discuss a formalization of their definitions in the Lean 3 theorem prover. As a prerequisite, we formalize adic valuations on Dedekind domains. We present some applications, including the statement of the main theorem of global class field theory and a proof that the ideal class group of a number field is isomorphic to an explicit quotient of its idèle class group.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Type theory

Keywords and phrases formal math, algebraic number theory, class field theory, Lean, mathlib

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.14

Supplementary Material
Software (Source Code): `https://github.com/mariainesdff/ideles-journal`
InteractiveResource (Website): `https://mariainesdff.github.io/ideles-journal/`

Funding EPSRC Grant EP/V048724/1: Digitising the Langlands Program (UK).

Acknowledgements I would like to thank Kevin Buzzard for his constant support and for many helpful conversations during the completion of this project, and Ashvni Narayanan for pointing out that the finite adèle ring can be defined for any Dedekind domain. I am also grateful to Patrick Massot for making some of the topological prerequisites available in `mathlib`, and to Sebastian Monnet for formalizing the topology on the infinite Galois group. Finally, I thank the `mathlib` community for their helpful advice, and the `mathlib` maintainers for the insightful reviews of the parts of this project already submitted to the library.

1 Introduction

Number theory is the branch of mathematics that studies the ring of integer numbers $\mathbb{Z}$ and its field of fractions $\mathbb{Q}$, the rational numbers. While this description may seem deceptively simple, it is a very rich area, involving myriads of abstractions and techniques.

Consider for example the problem of finding all integer solutions to a polynomial equation in several variables (a "Diophantine equation"). Perhaps the most famous of these equations is $x^n + y^n = z^n$, where n is an integer greater than 2. Fermat's Last Theorem tells us that this equation has no integer solutions for which the product xyz is nonzero. While Fermat was able to state this conjecture around 1637, its proof was not concluded until 1995, although some particular cases were established sooner.

The general proof, due to Wiles and Taylor, is built upon the combined work of hundreds of mathematicians who over the last couple of centuries developed a rich arithmetic theory of elliptic curves, modular forms and Galois representations. The key result is a special case of the Taniyama–Shimura–Weil conjecture. If we want to be able to formalize a complete proof of Fermat's Last Theorem in a theorem prover, we first need to formalize all the necessary ingredients.

In this paper we formalize the ring of adèles and the group of idèles of a *global field* (a generalization of the field $\mathbb{Q}$). As a consequence of our work we are able to state the main theorem of global class field theory. Class field theory is needed for the proof of the

Taniyama–Shimura–Weil conjecture, which implies Fermat's Last Theorem. Adèles and idèles are used in many areas of current research, including the theory of automorphic forms and the Langlands program, an ambitious group of conjectures that seek to establish deep connections between geometry and number theory.

Our formalization was carried out using the Lean 3 theorem prover [9]. At the time of writing this paper, the source code is in the process of being integrated in Lean's mathematics library `mathlib`. We provide a public repository[1] containing the version of the code referred to in this article and the associated documentation[2] in HTML format. This is the first time that adèles and idèles have been formalized in any theorem prover.

Before describing our formalization, we give a quick overview of the ring of adèles of $\mathbb{Q}$. When studying the rational numbers, both algebraic and analytic methods can be employed. A natural way to do analysis over $\mathbb{Q}$ is by regarding it as a subspace of the real numbers $\mathbb{R}$, which are by definition the completion of $\mathbb{Q}$ with respect to the usual absolute value. However, this is not the only absolute value that can be defined on $\mathbb{Q}$: in fact, for every prime number p, there is a p-adic absolute value $|\cdot|_p$ and we can consider the corresponding completion $\mathbb{Q}_p$ of $\mathbb{Q}$. Ostrowki's theorem tells us that, up to equivalence, there are no more nontrivial absolute values on the rational numbers.

We remark that while the field $\mathbb{Q}_p$ of p-adic numbers is a basic object in number theory, it was not formalized in any proof assistant until 2015, when Pelayo, Voevodsky, and Warren formalized it in the Coq UniMath library [15]. The p-adic numbers were added to Lean's mathematical library `mathlib` in 2018, by R. Y. Lewis [12].

Since the various absolute values on $\mathbb{Q}$ provide us with different insights about the rationals, a natural question is whether it is possible to study all of them simultaneously. A first approximation would be to consider the product of the completions with respect to each absolute value. However, for technical reasons it is better to work with the following subset of the product:

$$\mathbb{A}_{\mathbb{Q}} := \prod_p{}' \mathbb{Q}_p \times \mathbb{R} := \left\{ (x_p)_p \in \prod_p \mathbb{Q}_p \;\middle|\; |x_p|_p \leq 1 \text{ for all but finitely many } p \right\} \times \mathbb{R}.$$

$\mathbb{A}_{\mathbb{Q}}$ is a ring under component-wise addition and multiplication, it contains $\mathbb{Q}$ as a subring via the diagonal map $r \mapsto ((r)_p, r)$, and it can be endowed with a topology that makes it into a locally compact topological ring. We call $\mathbb{A}_{\mathbb{Q}}$ the ring of adèles or adèle ring of $\mathbb{Q}$ and $\mathbb{A}_{\mathbb{Q},f} := \prod_p{}' \mathbb{Q}_p$ its finite adèle ring. The groups of units of these rings are respectively called the idèle group $\mathbb{I}_{\mathbb{Q}}$ and finite idèle group $\mathbb{I}_{\mathbb{Q},f}$ of $\mathbb{Q}$.

The definitions of adèle ring and idèle group can be generalized to any global field K [2]; see sections 3 and 4 for the details. Global fields are one of the main subjects of study in algebraic number theory and they can be of two kinds: number fields, which are finite extensions of the field $\mathbb{Q}$, and function fields, which are finite extensions of the field $\mathbb{F}_q(t)$ of rational functions over a finite field $\mathbb{F}_q$.

Every global field is the field of fractions of a Dedekind domain, but the converse is not true. However, the definition of finite adèle ring makes sense for any Dedekind domain, so we have formalized it in that degree of generality.

[1] `https://github.com/mariainesdff/ideles-journal`
[2] `https://mariainesdff.github.io/ideles-journal/`

1.1 Lean and mathlib

Lean 3 is a functional programming language and interactive theorem prover [9] based on dependent type theory, with proof irrelevance and non-cumulative universes [7]. For an introduction to Lean, see for instance [3].

This project is based on Lean's mathematical library `mathlib`, which is characterized by its decentralized nature with over 200 contributors. Due to the distributed organization of `mathlib`, it is impossible to cite every author who contributed a piece of code that we used. However, we remark that our formalization makes extensive use of the theory of Dedekind domains [4] and of the theory of uniform spaces and completions, originally developed in the perfectoid space formalization project [6].

In Lean's core library and `mathlib`, type classes are used to handle mathematical structures on types. For example, the type class `ring` packages two operations, addition and multiplication, as well as a list of properties they must satisfy. Then, given a type `R`, we can declare an instance `[ring R]`, and Lean's instance resolution procedure will infer that `R` has a ring structure. Besides `instance`, whose behaviour we have just described, we use in this paper the keywords `variables`, `def`, `lemma` and `theorem`, which have the evident meaning.

1.2 Structure of the paper

We start Section 2 with some background on Dedekind domains and their nonarchimedean absolute values, which we then use to define the finite adèle ring and the finite idèle group and explore how the latter is related to the group of invertible fractional ideals. In Section 3, we build on this work to define the adèle ring, the idèle group and the idèle class group of a number field, while in Section 4 we treat the function field case. In Section 5 we discuss two applications of the idèle group to class field theory. Finally, we conclude Section 6.1 with some implementation remarks and a discussion of future work connected to this project.

2 The finite adèle ring of a Dedekind domain

2.1 Dedekind domains and adic valuations

There are several equivalent definitions of Dedekind domain, three of which have been formalized in `mathlib` [4]. We work with the one formalized in `is_dedekind_domain` : a Dedekind domain R is an integrally closed Noetherian integral domain with Krull dimension 0 or 1 [14].

A Dedekind domain of Krull dimension 0 is a field. In this project we will only consider Dedekind domains of Krull dimension 1, for which the maximal ideals are exactly the nonzero prime ideals. Some examples are the integers $\mathbb{Z}$, the Gaussian integers $\mathbb{Z}[i] := \{a + bi \mid a, b \in \mathbb{Z}\}$, or the ring of univariate polynomials $k[X]$ over a field k. All of these examples are unique factorization domains; however, not every Dedekind domain is. For instance, $\mathbb{Z}[\sqrt{-5}] := \{a + b\sqrt{-5} \mid a, b \in \mathbb{Z}\}$ is a Dedekind domain but not a unique factorization domain, since elements like $6 = 2 \cdot 3 = (1 + \sqrt{-5}) \cdot (1 - \sqrt{-5})$ admit two genuinely distinct factorizations.

The maximal spectrum of R is the set of its maximal ideals (implemented as a type in Lean). The fraction field K of R is the smallest field containing R; its elements can be represented by fractions r/s, where r and s are in R and s is nonzero. For example, the fraction fields of $\mathbb{Z}$, $\mathbb{Z}[i]$, and $k[X]$ are respectively $\mathbb{Q}$, $\mathbb{Q}(i) := \{a + bi \mid a, b \in \mathbb{Q}\}$, and the field $k(X)$ of rational functions over k.

```
variables (R : Type*) [comm_ring R] [is_domain R] [is_dedekind_domain R]
  {K : Type*} [field K] [algebra R K] [is_fraction_ring R K]
-- Note : not the maximal spectrum if R is a field
structure maximal_spectrum :=
(as_ideal : ideal R)
(is_prime : as_ideal.is_prime)
(ne_bot   : as_ideal ≠ ⊥)
variable (v : maximal_spectrum R)
```

Let R be a Dedekind domain (of Krull dimension 1). Then every nonzero ideal of R can be written as a product of maximal ideals, and this factorization is unique up to reordering. In particular, given an element $r \in R$ and a maximal ideal v of R, we can count how many times v appears in the factorization of the principal ideal (r), and this defines a nonarchimedean additive valuation on R [10, Chapter II], that is, a function $\mathrm{val}_v : R \to \mathbb{Z} \cup \{\infty\}$ such that

1. $\mathrm{val}_v(r) = \infty$ if and only if $r = 0$,
2. $\mathrm{val}_v(rs) = \mathrm{val}_v(r) + \mathrm{val}_v(s)$ for all r, s in R, and
3. $\mathrm{val}_v(r + s) \geq \min\{\mathrm{val}_v(r), \mathrm{val}_v(s)\}$ for all r, s in R.

The function val_v is called the v-adic valuation on R. It can be extended to a valuation on the fraction field K of R by defining $\mathrm{val}_v(r/s) := \mathrm{val}_v(r) - \mathrm{val}_v(s)$. For example, when $R = \mathbb{Z}$ and $v = (p)$ is the ideal generated by a prime number, val_v is the p-adic valuation on $\mathbb{Z}$ and $\mathbb{Q}$.

For both theoretical and implementation reasons, it is more convenient to work with the multiplicative version of the valuation: given any real number $n_v > 1$, we define a function $|\cdot|_v : R \to n_v^{\mathbb{Z}\cup\{-\infty\}} = n_v^{\mathbb{Z}} \cup \{0\}$ sending r to $n_v^{-\mathrm{val}_v(r)}$. From the definition of val_v, we immediately deduce that $|\cdot|_v$ has the following properties:

(i) $|r|_v = 0$ if and only if $r = 0$,
(ii) $|rs|_v = |r|_v|s|_v$ for all r, s in R, and
(iii) $|r + s|_v \leq \max\{|r|_v, |s|_v\}$ for all r, s in R.

A function $|\cdot|_v$ satisfying conditions (i) – (iii) is called a nonarchimedean absolute value (note that the third condition is stronger than $|r + s|_v \leq |r|_v + |s|_v$). The choice of n_v used in the definition is not relevant, in the sense that any two choices of n_v will yield equivalent absolute values. If, instead of property (iii), the function $|\cdot|_v$ satisfies only the weaker condition $|r + s|_v \leq |r|_v + |s|_v$, we say that it is an archimedean absolute value.

We formalized the v-adic absolute value on R in `mathlib` using the structure `valuation`, which consists on a function $|\cdot|$ from a ring R to a `linear_ordered_comm_monoid_with_zero` Γ_0 satisfying conditions (ii) and (iii), plus $|0| = 0$ and $|1| = 1$. We chose Γ_0 equal to `with_zero (multiplicative ℤ)`, which is a way to represent $n_v^{\mathbb{Z}} \cup \{0\}$ in Lean : if `T` is a type that carries some additive structure, then `multiplicative T` carries the corresponding multiplicative structure. The definition `with_zero` is used to add a new element 0 to a given type.

In the code below, if r is a nonzero element of R, we use `(associates.mk (ideal.span r : ideal R)).factors` to obtain the multiset of factors of the ideal (r) and we count how many times the maximal ideal v appears in this factorization with `(associates.mk v.as_ideal).count`. This count is returned as a natural number; after coercing to $\mathbb{Z}$ and taking its negative, we use `multiplicative.of_add` to get the corresponding element of `multiplicative ℤ`.

Let us briefly explain why we use `associates.mk` in this definition. Two elements of a monoid are associated if they differ by multiplication by a unit, and this defines an equivalence relation. In `mathlib`, given a monoid `M`, `associates M` is the quotient of `M` by this equivalence relation, and `associates.mk` is the canonical map sending an element to its equivalence class.

Two ideals of a commutative ring are associated if and only if they are equal, so from a mathematical point of view, the associated relation is trivial in this case. However, from an implementation point of view, it is more convenient to work with associates than to work directly with ideals, since the corresponding factorization API (that is, the collection of available definitions and lemmas) is more extensive.

```
def int_valuation_def (r : R) : with_zero (multiplicative ℤ) :=
if r = 0 then 0 else multiplicative.of_add (-(associates.mk v.as_ideal).count
  (associates.mk (ideal.span {r} : ideal R)).factors : ℤ)
def int_valuation : valuation R (with_zero (multiplicative ℤ)) :=
{ to_fun           := v.int_valuation_def,
  map_zero'        := int_valuation.map_zero' v,
  map_one'         := int_valuation.map_one' v,
  map_mul'         := int_valuation.map_mul' v,
  map_add_le_max'  := int_valuation.map_add_le_max' v }
```

We extended `int_valuation` to a `valuation` on the fraction field K, by setting the valuation of a fraction to be the valuation of the numerator divided by the valuation of the denominator. We checked in `lemma valuation_well_defined` that this definition does not depend on the choice of fraction used to represent an element of K.

```
def valuation_def (x : K) : (with_zero (multiplicative ℤ)) :=
let s := classical.some (classical.some_spec (is_localization.mk'_surjective
  (non_zero_divisors R) x)) in
  (v.int_valuation_def (classical.some (is_localization.mk'_surjective
  (non_zero_divisors R) x)))/(v.int_valuation_def s)
```

```
lemma valuation_well_defined {r r' : R} {s s' : non_zero_divisors R}
  (h_mk : is_localization.mk' K r s = is_localization.mk' K r' s') :
  (v.int_valuation_def r)/(v.int_valuation_def s) =
  (v.int_valuation_def r')/(v.int_valuation_def s')
```

We proved several properties of the valuation[3], of which we remark the fact that for every maximal ideal v of R, there exists a uniformizer $\pi_v \in K$ for the v-adic valuation, that is, an element having absolute value $|\pi_v|_v = n_v^{-1}$, or equivalently additive v-adic valuation 1.

```
lemma valuation_exists_uniformizer :
  ∃ (π : K), v.valuation_def π = multiplicative.of_add (-1 : ℤ)
```

We also provide some examples[4] of explicit computations of 2-adic valuations of elements of $\mathbb{Z}$ and $\mathbb{Q}$.

Since $|\cdot|_v$ is an absolute value on the Dedekind domain R and its field of fractions K, we can complete R and K with respect to $|\cdot|_v$. We denote the respective completions by R_v and K_v, and recall that R_v is an integral domain with field of fractions K_v.

We first formalize the definition of K_v using the theory of completions of valued fields available in `mathlib`, which was originally developed as part of the formalization of perfectoid spaces [6]. Among the possible ways to define K_v, this one was chosen because of its powerful API : we can use the `field_completion` instance to recover the fact that K_v is a field, and `valued.extension_valuation` to extend the v-adic valuation on K to a valuation on the completion K_v. We denoted by `adic_completion K v` the completion of K with respect to the v-adic valuation.

[3] https://github.com/mariainesdff/ideles-journal/blob/master/src/valuation.lean
[4] https://github.com/mariainesdff/ideles-journal/blob/master/src/examples.lean

```
def adic_valued : valued K (with_zero (multiplicative ℤ)) := ⟨v.valuation⟩
```

```
def adic_completion := @uniform_space.completion K (us' v)
instance : field (v.adic_completion K) :=
@field_completion K _ (us' v) (tdr' v) _ (ug' v)
instance valued_adic_completion :
  valued (v.adic_completion K) (with_zero (multiplicative ℤ)):=
⟨@valued.extension_valuation K _ _ _ v.adic_valued)⟩
```

It can be shown that R_v is equal to the ring of integers of K_v, that is, the subring of K_v consisting of elements of absolute value less than or equal to one. In our formalization, we actually use this characterization to define R_v (which we called `adic_completion_integers`), so that we automatically have an inclusion of R_v in K_v.

```
def adic_completion_integers : subring (v.adic_completion K) :=
@valuation.integer (v.adic_completion K) (with_zero (multiplicative ℤ)) _ _
  v.valued_adic_completion.v
```

2.2 The finite adèle ring

Now that we have defined nonarchimedean absolute values on a Dedekind domain R and their extension to K, we can attempt to simultaneously study all of them. In order to do so, we define the finite adèle ring $\mathbb{A}_{R,f}$ of R as the restricted product of the completions K_v with respect to their ring of integers R_v, i. e.,

$$\mathbb{A}_{R,f} := \prod_v{}' K_v := \left\{ (x_v)_v \in \prod_v K_v \;\middle|\; x_v \in R_v \text{ for all but finitely many } v \right\},$$

where v runs over the set of maximal ideals of R. Recall that $x_v \in R_v$ is equivalent to $|x_v|_v \leq 1$, so $\mathbb{A}_{R,f}$ is an immediate generalization of $\mathbb{A}_{\mathbb{Q},f}$.

Since $\mathbb{A}_{R,f}$ is a subset of the product $\prod_v K_v$, it is easy to prove that it is a commutative ring with component-wise addition and multiplication (one just needs to check that it is closed under addition, negation and multiplication).

```
def K_hat := (Π (v : maximal_spectrum R), v.adic_completion K )
def finite_adele_ring' := { x : (K_hat R K) //  ∀ᶠ (v : maximal_spectrum R) in
  filter.cofinite, (x v ∈ v.adic_completion_integers K) }
instance : comm_ring (finite_adele_ring' R K) := ...
```

In Lean, the notation `{t :  T // p t}` is used to define the type of pairs `⟨t, p t⟩` where `t` is a term of type `T` that satisties some predicate `p : T → Prop`. Since we use this construction to define `finite_adele_ring'`, given a finite adèle `x :  finite_adele_ring' R K`, we can use `x.val` to get the corresponding term of the product `K_hat R K`, and `x.property` to access a proof that the component `x.val v` belongs to R_v for all but finitely many maximal ideals v. Lean's syntax to indicate that a certain property `p` holds for all but finitely many terms of a type is `∀ᶠ (t :  T) in filter.cofinite, p t`.

We endow $\mathbb{A}_{R,f}$ with the topology generated by the collection of sets $\{\Pi_v U_v \mid U_v \text{ is open and } U_v = R_v \text{ for all but finitely many } v\}$ and prove that addition and multiplication on $\mathbb{A}_{R,f}$ are continuous for this topology, which makes $\mathbb{A}_{R,f}$ into a topological ring. While these proofs are not conceptually hard, their formalization turned out to be quite long. The main reason is that, while on paper we can express that a subset U of $\mathbb{A}_{R,f}$ is equal to a product of subsets V_v of K_v by writing $U = \prod_v V_v$, this cannot be an equality in the formalization

since the two sides of the equation have different types. Instead, we are forced to say that there exists a collection of subsets V_v of K_v such that a finite idèle $x = (x_v)_v$ belongs to U if and only if x_v belongs to V_v for all v, which adds some extra bookkeeping to the proof. A second reason is that, when checking that addition (or multiplication) is continuous at the pair of adèles (x, y), the argument has to be split in several cases depending on whether the v components of x and y are integers, and whether the open sets V_v equal R_v.

```
def finite_adele_ring'.generating_set : set (set (finite_adele_ring' R K)) :=
{ U : set (finite_adele_ring' R K)
  ∃ (V : Π (v : maximal_spectrum R), set (v.adic_completion K)),
    (∀ x : finite_adele_ring' R K, x ∈ U ↔ ∀ v, x.val v ∈ V v) ∧
    (∀ v, is_open (V v)) ∧
    ∀ᶠ v in filter.cofinite, V v = v.adic_completion_integers K }
instance : topological_space (finite_adele_ring' R K) :=
topological_space.generate_from (finite_adele_ring'.generating_set R K)
```

For every element $k \in K$, there are finitely many maximal ideals v of R such that the v-adic absolute value of k is greater than 1; hence $(k)_v$ is a finite adèle of R. The map $\mathrm{inj}_K : K \to \mathbb{A}_{R,f}$ sending k to $(k)_v$ is an injective ring homomorphism, which allows us to regard K as a subring of $\mathbb{A}_{R,f}$. Note that we are using the fact that R has Krull dimension 1 to conclude the injectivity of this map, since if R had Krull dimension 0, then $\mathbb{A}_{R,f}$ would be the trivial ring, and injectivity would fail.

```
def inj_K : K → finite_adele_ring' R K :=
λ x, ⟨(λ v : maximal_spectrum R, (coe : K → (v.adic_completion K)) x),
  inj_K_image R K x⟩
```

One might wonder why we defined $\mathbb{A}_{R,f}$, instead of just working with the full product $\prod_v K_v$. The main reason for this is that, while both $\mathbb{A}_{R,f}$ and $\prod_v K_v$ are topological rings containing K as a subring, only the former is locally compact and contains K as a discrete and co-compact subring. Since $\mathbb{A}_{R,f}$ is in particular a locally compact topological group, it is possible to define a (unique up to scalars) Haar measure on $\mathbb{A}_{R,f}$, which allows us to integrate functions over $\mathbb{A}_{R,f}$. Tate famously used this integration theory in his thesis to study the properties of Hecke L-functions of number fields. Note that Haar measures have recently been formalized in `mathlib` [17].

2.2.1 Alternative definition of the finite adèle ring

There is a second characterization of the ring of finite adèles of R which is also widely used in number theory. We start with the product $\hat{R} := \prod_v R_v$ over all maximal ideals of R and observe that it contains R via the diagonal inclusion $r \mapsto (r)_v$. Hence, we can consider the localization $(\prod_v R_v)[\frac{1}{R\setminus\{0\}}]$ of $\hat{R}$ at $R \setminus \{0\}$, consisting of tuples of the form $(\frac{r_v}{s})_v$ where $r_v \in R_v$ for all v and $s \in R \setminus \{0\} \subseteq R_v \setminus \{0\}$.

To define the topological ring structure on $\hat{R}[\frac{1}{R\setminus\{0\}}]$, we use the fact that for any ring S, ring topologies on S form a complete lattice. In particular, given any map $f : T \to S$ from a topological space T to a ring S, one can define the coinduced ring topology on S to be the finest topology such that S is a topological ring and f is continuous. The complete lattice structure was formalized as part of this project and is already a part of `mathlib`. We give $\hat{R}[\frac{1}{R\setminus\{0\}}]$ the ring topology coinduced by the localization map $(r_v)_v \mapsto (\frac{r_v}{1})_v$ from $\hat{R}$ with the product topology to $\hat{R}[\frac{1}{R\setminus\{0\}}]$.

It is well known that $\mathbb{A}_{R,f}$ is isomorphic to $(\prod_v R_v)[\frac{1}{R\setminus\{0\}}]$ as topological rings. Given an element $(\frac{r_v}{s})_v \in (\prod_v R_v)[\frac{1}{R\setminus\{0\}}]$, the absolute value $|\frac{r_v}{s}|_v$ will be less than or equal to one, except possibly at the finitely many v dividing the denominator s; hence $(\frac{r_v}{s})_v$ is a finite adèle and one easily sees that this map is an isomorphism of rings. Checking that it is also a homeomorphism requires more work.

We formalized this second definition of the adèle ring in `finite_adele_ring`, but we have not yet formalized the proof that the two definitions yield isomorphic topological rings. A strategy to verify that the map described above is a homeomorphism boils down to checking that the localization maps $\hat{R} \to \hat{R}[\frac{1}{R\setminus\{0\}}]$ and $\hat{R} \to \mathbb{A}_{R,f}$ sending $(r_v)_v$ to $(\frac{r_v}{1})_v$ are both continuous and open; however, formalizing this would take some time and, since we do not have immediate plans to use this second definition of the finite adèle ring, we leave it as future work.

The `finite_adele_ring` definition has the advantage that, being defined as a localization, `finite_adele_ring R` automatically inherits a commutative topological ring structure, while for `finite_adele_ring' R` this has to be proven by hand. However, we found that for proving results such as the one described in Section 5.1, our first definition was easier to work with.

```
def finite_adele_ring := localization (diag_R R K)
instance : comm_ring (finite_adele_ring R K) := localization.comm_ring
instance : algebra (R_hat R K) (finite_adele_ring R K) := localization.algebra
instance : is_localization (diag_R R K) (finite_adele_ring R K):=
localization.is_localization
instance : topological_space (finite_adele_ring R K) :=
localization.topological_space
instance : topological_ring (finite_adele_ring R K) :=
localization.topological_ring
```

2.3 The finite idèle group

The finite idèle group $\mathbb{I}_{R,f}$ of R is the unit group of the finite adèle ring $\mathbb{A}_{R,f}$. It is a topological group with the topology induced by the map $\mathbb{I}_{R,f} \to \mathbb{A}_{R,f} \times \mathbb{A}_{R,f}$ sending x to (x, x^{-1}). This topology is finer than the subspace topology induced by the inclusion of $\mathbb{I}_{R,f}$ in $\mathbb{A}_{R,f}$, which is not a group topology since inversion fails to be continuous.

```
def finite_idele_group' := units (finite_adele_ring' R K)
instance : topological_space (finite_idele_group' R K) := units.topological_space
instance : comm_group (finite_idele_group' R K) := units.comm_group
instance : topological_group (finite_idele_group' R K) := units.topological_group
```

Note that for every nonzero $k \in K$, the finite adèle $(k)_v$ is invertible, with inverse $(k^{-1})_v$. It follows that $\mathbb{I}_{R,f}$ contains $K^* = K \setminus \{0\}$ as a subgroup. We formalize this fact by defining a function `inj_units_K` from K^* to $\mathbb{I}_{R,f}$ and proving that it is an injective group homomorphism. As in Section 2.2, the injectivity of this map requires the fact that R has Krull dimension 1.

```
def inj_units_K : units K → finite_idele_group' R K :=
λ x, ⟨inj_K R K x.val, inj_K R K x.inv, right_inv R K x, left_inv R K x⟩
```

2.4 Relation to fractional ideals

The finite idèle group of R is closely related to its group of invertible fractional ideals. A fractional ideal of R is an R-submodule I of K for which there exists an $a \in R$ such that aI is an ideal J of R. We say that I is invertible if there exists another fractional ideal I' such that $II' = R$.

For a Dedekind domain R, every nonzero fractional ideal is invertible and can be factored as a product $v_1^{n_1} \cdots v_m^{n_m}$ of maximal ideals of R where the n_i are integers, uniquely up to reordering of the factors. We formalize this definition in `fractional_ideal.factorization`, where we express I as a `finprod` over all maximal ideals of R, as follows.

The `finprod` of a function `f : T → M` from a type `T` to a commutative monoid `M` is defined to be the product of all values `f t` as `t` ranges over `T`, if `f t = 1` for all but finitely many `t`; otherwise `finprod f` is defined to be one. The notation Π^f `t, f t` can be used in place of `finprod f`. Given a fractional ideal I, denote by n_v the exponent of the maximal ideal v in the factorization of I and let $f :$ maximal spectrum$(R) \to$ fractional ideals(R) be the function sending v to v^{n_v}. Since all but finitely many of the n_v are zero, I is equal to `finprod f`. Besides proving this, we provide some API to work with the exponents appearing in the factorization.

```
lemma fractional_ideal.factorization (I : fractional_ideal (non_zero_divisors R) K)
  (hI : I ≠ 0) {a : R} {J : ideal R}
  (haJ : I = fractional_ideal.span_singleton (non_zero_divisors R)
    ((algebra_map R K) a)⁻¹ * ↑J) :
  Π^f (v : maximal_spectrum R),
    (v.as_ideal : fractional_ideal (non_zero_divisors R) K)^ ((associates.mk
      v.as_ideal).count (associates.mk J).factors - (associates.mk
      v.as_ideal).count (associates.mk (ideal.span{a})).factors : ℤ) = I
```

We can define a group homomorphism from $\mathbb{I}_{R,f}$ to the group of invertible fractional ideals by sending $(x_v)_v \in \mathbb{I}_{R,f}$ to the product $\Pi_v v^{\mathrm{val}_v(x_v)}$. Since for every $(x_v)_v \in \mathbb{I}_{R,f}$ there are finitely many maximal ideals v such that $\mathrm{val}_v(x_v)$ is nonzero, this product is actually finite, so it indeed defines a nonzero fractional ideal of R.

```
def finite_idele.to_add_valuations (x : finite_idele_group' R K) :
  Π (v : maximal_spectrum R), ℤ :=
λ v, -(with_zero.to_integer ((valuation.ne_zero_iff valued.v).mpr
  (v_comp.ne_zero R K v x)))
lemma finite_add_support (x : finite_idele_group' R K ) :
  ∀^f (v : maximal_spectrum R) in filter.cofinite,
    finite_idele.to_add_valuations R K x v = 0 := ...
def map_to_fractional_ideals.val :
  (finite_idele_group' R K) → (fractional_ideal (non_zero_divisors R) K) :=
λ x, Π^f (v : maximal_spectrum R), (v.as_ideal : fractional_ideal
  (non_zero_divisors R) K)^(finite_idele.to_add_valuations R K x v)
```

We show that this homomorphism is surjective and its kernel is the set $\mathbb{I}_{R,\infty}$ of elements $(x_v)_v$ in $\mathbb{I}_{R,f}$ having additive valuation zero at all v. Moreover, this map is continuous when the group of invertible fractional ideals is given the discrete topology.

3 Adèles and idèles of number fields

3.1 Number fields and their rings of integers

A number field K is a finite extension of the field $\mathbb{Q}$ of rational numbers [10]. Every finite extension is algebraic, so every element $k \in K$ is the root of a polynomial with coefficients in $\mathbb{Q}$. If moreover k is the root of a monic polynomial with integer coefficients, we say that k is an algebraic integer. The algebraic integers of K form a subring $\mathcal{O}_K$, called the ring of integers of K, which is a Dedekind domain of Krull dimension 1 in which every nonzero ideal is of finite index.

Remember from the introduction that one motivation for defining the adèles of K was to simultaneously study all the (equivalence classes of) nontrivial absolute values on K. These absolute values can be split into two kinds: nonarchimedean and archimedean. The nonarchimedean ones are exactly the v-adic absolute values associated to maximal ideals of the ring of integers $\mathcal{O}_K$, discussed in section 2.1.

To obtain the archimedean absolute values, we first recall that we can find a $\mathbb{Q}$-vector space basis of K of the form $\{1, \alpha, \dots, \alpha^{n-1}\}$, where n is the dimension of K over $\mathbb{Q}$ and α is an element of K. This α is a root of a degree n polynomial f_α with coefficients in $\mathbb{Q}$. For each real root r of f_α, we get an embedding of K into the real numbers $\mathbb{R}$ (the map sending α to r), and restricting the usual absolute value on $\mathbb{R}$ to the image of K, we get an archimedean absolute value on K. Similarly, for every pair of complex conjugate roots (s_1, s_2) of f_α, we get a pair of embeddings of K into the complex numbers $\mathbb{C}$, and we can restrict the complex absolute value to the image of K under one of them to get an absolute value on K. Note that the two embeddings coming from a conjugate pair yield equivalent absolute values.

3.2 The ring of adèles

Let K be a number field. We define the ring of adèles of K as the restricted product of the completions K_v of K with respect to each absolute value $|\cdot|_v$ on it: $\mathbb{A}_K := \prod'_{|\cdot|_v} K_v$. That is, $\mathbb{A}_K$ is the subring of the product $\prod_{|\cdot|_v} K_v$ consisting on tuples $(a_v)_v$ such that $|a_v|_v \leq 1$ for all but finitely many v. Since each nonarchimedean absolute value $|\cdot|_v$ corresponds to a maximal ideal v of O_K, and there are finitely many archimedean absolute values, we can rewrite this definition as

$$\mathbb{A}_K := \prod_{v \text{ max.}}' K_v \times \prod_{|\cdot|_v \text{ arch.}} K_v = \prod_{v \text{ max.}}' K_v \times (\mathbb{R} \otimes_{\mathbb{Q}} K),$$

where we have used a theorem from algebraic number theory to get the second equality. Note that $\prod'_v K_v$ is the finite adèle ring associated to the Dedekind domain $\mathcal{O}_K$; we will denote it by $\mathbb{A}_{K,f}$ and call it the finite adèle ring of K. We formalize these definitions as follows:

```
variables (K : Type) [field K] [number_field K]
def A_K_f := finite_adele_ring' (ring_of_integers K) K
def A_K := (A_K_f K) × (ℝ ⊗[ℚ] K)
```

We proved in Section 2.2 that `A_K_f` is a topological commutative ring. The product and tensor product of commutative rings are commutative rings, so `A_K` is a commutative ring. To prove that it is a topological commutative ring, it therefore suffices to show that $\mathbb{R} \otimes_{\mathbb{Q}} K$ is a topological ring. We do this by using the fact that there are isomorphisms $\mathbb{R}^n \simeq \mathbb{R} \otimes_{\mathbb{Q}} \mathbb{Q}^n \simeq \mathbb{R} \otimes_{\mathbb{Q}} K$, where n is the dimension of K over $\mathbb{Q}$.

Note that $\mathbb{R}^n$ is represented in Lean by the type `fin n → ℝ` of functions from $\{1,\dots,n\}$ to $\mathbb{R}$, and it is a topological commutative ring when endowed with the product topology `Pi.topological_space`:

```
variables (n : ℕ)
instance : ring (fin n → ℝ) := pi.ring
instance : topological_space (fin n → ℝ) := Pi.topological_space
instance : has_continuous_add (fin n → ℝ) := pi.has_continuous_add'
instance : has_continuous_mul (fin n → ℝ) := pi.has_continuous_mul'
instance : topological_semiring (fin n → ℝ) := topological_semiring.mk
instance : topological_ring (fin n → ℝ) := topological_ring.mk
```

We then define the topology on $\mathbb{R} \otimes_{\mathbb{Q}} K$ as the ring topology coinduced by the map $\mathbb{R}^n \to \mathbb{R} \otimes_{\mathbb{Q}} K$. Finally, `A_K` becomes a topological ring with the product topology.

```
def linear_map.Rn_to_R_tensor_K :
  (fin (finite_dimensional.finrank ℚ K) → ℝ) →ₗ[ℝ] (ℝ ⊗[ℚ] K) :=
linear_map.comp (linear_map.base_change K) (linear_map.Rn_to_R_tensor_Qn _)
def infinite_adeles.ring_topology : ring_topology (ℝ ⊗[ℚ] K) :=
ring_topology.coinduced (linear_map.Rn_to_R_tensor_K K)
instance : topological_space (ℝ ⊗[ℚ] K) :=
(infinite_adeles.ring_topology K).to_topological_space
instance : topological_ring (ℝ ⊗[ℚ] K) :=
(infinite_adeles.ring_topology K).to_topological_ring
instance : topological_space (A_K K) := prod.topological_space
instance : topological_ring (A_K K) := prod.topological_ring
```

We end this section by recalling that $\mathbb{A}_{K,f}$ contains the field K as a subring via the diagonal map sending $k \in K$ to the finite adèle $(k)_v$, which is injective due to the fact that the ring of integers of a number field is not a field[5]. Combining this with the natural inclusion $k \mapsto 1 \otimes k$ of K in $\mathbb{R} \otimes_{\mathbb{Q}} K$, we can also view K as a subring of $\mathbb{A}_K$.

```
def inj_K_f : K → A_K_f K := inj_K (ring_of_integers K) K
def inj_K : K → A_K K :=
λ x, ⟨inj_K_f K x, algebra.tensor_product.include_right x⟩
```

3.3 The group of idèles and the idèle class group

We define the group $\mathbb{I}_K$ of idèles of K as the unit group of the ring of adèles $\mathbb{A}_K$, and the group $\mathbb{I}_{K,f}$ of finite idèles as the unit group of $\mathbb{A}_{K,f}$.

```
def I_K_f := units (A_K_f K)
def I_K := units (A_K K)
```

For every nonzero $k \in K$, the finite adèle $(k)_v$ is a unit (with inverse $(k^{-1})_v$), and so is the adèle $((k)_v, 1 \otimes k)$. Therefore, we can regard K^* as a subgroup of the (finite) idèle group, which allows us to define the idèle class group C_K of K as the quotient of $\mathbb{I}_K$ by K^*. C_K is a topological group with the quotient topology.

```
def C_K := (I_K K) / (inj_units_K.group_hom K).range
```

The name idèle class group is justified by the close relation between C_K and the ideal class group of K, which we discuss in section 5.1.

[5] `https://mariainesdff.github.io/ideles-journal/adeles_number_field.html#number_field.ring_of_integers.not_field`

4 Adèles and idèles of function fields

Let k be a field, $k[t]$ be the ring of polynomials in one variable over k and $k(t)$ be the field of rational functions (quotients of polynomials) over k. A function field F is a finite field extension of $k(t)$ [16].

```
variables (k F : Type) [field k] [field F] [algebra (polynomial k) F]
  [algebra (ratfunc k) F] [function_field k F]
  [is_scalar_tower (polynomial k) (ratfunc k) F] [is_separable (ratfunc k) F]
```

All of the absolute values that can be defined over $k(t)$ are nonarchimedean: there is one v-adic absolute value for each maximal ideal v of $k[t]$, plus one extra absolute value, called the place at infinity $|\cdot|_\infty$, defined by setting $\left|\frac{f}{g}\right|_\infty = q^{\deg(f)-\deg(g)}$, where $q > 1$ is a fixed real number. The completion of $k(t)$ with respect to this absolute value is the field $k((t^{-1}))$ of formal Laurent series in t^{-1}.

Following the strategy from Section 2.1, we formalize $|\cdot|_\infty$ in Lean under the name `infty_valuation` and we let `kt_infty` denote the completion of $k(t)$ with respect to $|\cdot|_\infty$.

```
def infty_valuation_def (r : ratfunc k) : with_zero (multiplicative ℤ) :=
if (r = 0) then 0 else
  (multiplicative.of_add ((r.num.nat_degree : ℤ) - r.denom.nat_degree))
def kt_infty := @uniform_space.completion (ratfunc k) (usq' k)
```

More generally, all of the absolute values on a function field F over k are nonarchimedean. Most of them correspond to maximal ideals of the integral closure of $k[t]$ in F. The finite adèle ring of F is the restricted product

$$\mathbb{A}_{F,f} := \prod_v{}' F_v := \left\{ (x_v)_v \in \prod_v F_v \;\middle|\; |x_v|_v \leq 1 \text{ for all but finitely many } v \right\},$$

where v runs over these maximal ideals. However, F also contains a finite collection of nonarchimedean absolute values coming from the absolute value $|\cdot|_\infty$ on $k(t)$. In order to include these absolute values as well, we define the adèle ring of F as the product

$$\mathbb{A}_F := \mathbb{A}_{F,f} \times (k((t^{-1})) \otimes_{k(t)} F).$$

```
def A_F_f := finite_adele_ring' (ring_of_integers k F) F
def A_F := (A_F_f k F) × ((kt_infty k) ⊗[ratfunc k] F)
```

The (finite) adèle ring of F is a topological commutative ring. We define the (finite) idèle group of F to be its group of units, respectively denoted $\mathbb{I}_{F,f}$ and $\mathbb{I}_F$, with the topology induced by the map $x \mapsto (x, x^{-1})$ as in Section 2.3.

The idèle class group C_F of F is the quotient of $\mathbb{I}_F$ by F^*. Since, as in the number field case, the ring of integers of F is not a field[6] and hence the diagonal inclusion of F^* in $\mathbb{I}_F$ is injective, C_F is a topological group with the quotient topology.

```
def I_F_f := units (A_F_f k F)
def I_F := units (A_F k F)
def C_F := (I_F k F) / (inj_units_F.group_hom k F).range
```

[6] `https://mariainesdff.github.io/ideles-journal/adeles_function_field.html#function_field.not_is_field`

Note that in number theory one is usually interested in the adèle ring of a function field over a finite field $k = \mathbb{F}_q$. However, $\mathbb{A}_F$ can be defined for any choice of field k, so we do not require k to be finite in our formalization; instead, this finiteness assumption will have to be included in the lemmas that need it.

5 Class Field Theory

Class field theory is a branch of number theory whose goal is to describe the Galois abelian extensions of a local or global field K, as well as their corresponding Galois groups, in terms of the arithmetic of the field K [1, 8, 13]. Recall from the introduction that a global field is either a number field or a function field over a finite field $\mathbb{F}_q$. A local field is the completion of a global field with respect to an absolute value. Examples of local fields include the real numbers $\mathbb{R}$, the complex numbers $\mathbb{C}$, the p-adic numbers $\mathbb{Q}_p$, or the field $\mathbb{F}_q((X))$ of formal Laurent series over a finite field.

In this section we discuss two class field theory results involving the definition of the idèle class group. The first one is a proof that the ideal class group of a global field is isomorphic to a quotient of its idèle class group, which we describe explicitly. The second one is a formalization of the statement of the main theorem of global class field theory.

5.1 The ideal class group is a quotient of the idèle class group

We have seen in Section 2.4 that, for any Dedekind domain R, there is a continuous surjective group homomorphism from the finite idèle group $\mathbb{I}_{R,f}$ to the group $\mathrm{Fr}(R)$ of invertible fractional ideals of R, sending $(x_v)_v$ to $\prod_v v^{\mathrm{val}_v(x_v)}$.

If K is a number field with ring of integers R, we can extend this map to a group homomorphism $\mathbb{I}_K \to \mathrm{Fr}(R)$ by pre-composing with the natural projection $\mathbb{I}_K \to \mathbb{I}_{K,f}$, obtaining again a continuous surjection. It is easy to see that an idèle $((x_v)_v, r \otimes_{\mathbb{Q}} k) \in \mathbb{I}_K$ belongs to the kernel of this map, which we denote $\mathbb{I}_{K,\infty}$, if and only if $\mathrm{val}_v(x_v)$ is equal to zero for every maximal ideal v of R. We wrote this map in Lean and formalized proofs of each of the listed properties.

```
-- For a Dedekind domain R with fraction field K :
def map_to_fractional_ideals.val :
  (finite_idele_group' R K) → (fractional_ideal (non_zero_divisors R) K) :=
λ x, Π^f (v : maximal_spectrum R), (v.as_ideal : fractional_ideal
  (non_zero_divisors R) K)^(finite_idele.to_add_valuations R K x v)
```

```
lemma I_K.map_to_fractional_ideals.surjective :
   function.surjective (I_K.map_to_fractional_ideals K) := ...
lemma I_K.map_to_fractional_ideals.continuous :
   continuous (I_K.map_to_fractional_ideals K) := ...
lemma I_K.map_to_fractional_ideals.mem_kernel_iff (x : I_K K) :
   I_K.map_to_fractional_ideals K x = 1 ↔ ∀ v : maximal_spectrum
    (ring_of_integers K), finite_idele.to_add_valuations (ring_of_integers K) K
    (I_K.fst K x) v = 0 := ...
```

Now, we want to show that this map induces a homomorphism at the level of class groups. The ideal class group $\mathrm{Cl}(K)$ of K is defined as the quotient of the group of invertible fractional ideals of K by the subgroup of principal fractional ideals. It is an important object in algebraic number theory, since it can be interpreted as a measure of how far the ring of integers of K is from being a unique factorization domain.

Note that the idèle $((k)_v, 1 \otimes_{\mathbb{Q}} k)$ corresponding to a nonzero $k \in K$ gets mapped to $\prod_v v^{\mathrm{val}_v(k)}$, which is the principal fractional ideal generated by k. Hence, we get an induced map from the idèle class group C_K to the ideal class group $\mathrm{Cl}(K)$. Using the universal property of the quotient topology, we conclude that this map $C_K \to \mathrm{Cl}(K)$ is a continuous surjective homomorphism, with kernel $\mathbb{I}_{K,\infty}K^*/K^*$. Therefore, by the first isomorphism theorem for topological groups, $\mathrm{Cl}(K)$ is isomorphic to the quotient of C_K by $\mathbb{I}_{K,\infty}K^*/K^*$.

The complete formalization of this proof can be found in the file `ideles_number_field`[7]. The theorem also holds in the function field case, with a completely analogous proof available in the file `ideles_function_field`[8]. By providing this proof, we show that our formalization of the adèles and idèles of a global field can be effectively used in practice to prove graduate-level number theoretic results.

5.2 The main theorem of global class field theory

Let K be a number field, $\overline{K}$ an algebraic closure of K and $G_K := \mathrm{Gal}_{\overline{K}/K}$ the Galois group of the extension $\overline{K}/K$. The topological group G_K is isomorphic to the inverse limit $\varprojlim_L \mathrm{Gal}(L/K)$ over all finite extensions L/K, with the inverse limit topology. We consider the topological abelianization $G_K^{ab} := G_K/\overline{[G_K, G_K]}$ of G_K, defined as the quotient of G_K by the topological closure of the commutator subgroup of G_K. The group G_K^{ab} is a topological group with the quotient topology, because $\overline{[G_K, G_K]}$ is a normal subgroup of G_K.

An exercise in infinite Galois theory shows that G_K^{ab} is the Galois group of the maximal abelian extension K^{ab} of K. The main theorem of global class field theory allows us to describe this Galois group in terms of the idèle class group of K :

▶ **Theorem 1** (Main Theorem of Global Class Field Theory). *Let K be a number field. Denote by $\pi_0(C_K)$ the quotient of C_K by the connected component of the identity. There is an isomorphism of topological groups $\pi_0(C_K) \simeq G_K^{ab}$.*

We formalized the statement of this theorem in two parts: we first claimed the existence of a group isomorphism `main_theorem_of_global_CFT.group_isomorphism` between $\pi_0(C_K)$ and G_K^{ab} and then in `main_theorem_of_global_CFT.homeomorph` we stated that this map is also a homeomorphism. Note that a complete pen-and-paper proof of this theorem spans hundreds of pages, so we have not attempted to formalize it.

```
variables (K : Type) [field K] [number_field K]
theorem main_theorem_of_global_CFT.group_isomorphism : (number_field.C_K K) /
    (subgroup.connected_component_of_one (number_field.C_K K)) ≃* (G_K_ab K) :=
sorry
theorem main_theorem_of_global_CFT.homeomorph :
homeomorph ((number_field.C_K K) / (subgroup.connected_component_of_one
    (number_field.C_K K))) (G_K_ab K) :=
{ continuous_to_fun  := sorry,
  continuous_inv_fun := sorry,
  ..(main_theorem_of_global_CFT.group_isomorphism K) }
```

[7] https://github.com/mariainesdff/ideles-journal/blob/master/src/ideles_number_field.lean
[8] https://github.com/mariainesdff/ideles-journal/blob/master/src/ideles_function_field.lean

6 Discussion

6.1 Design choices

Now that we have described all of the number theoretical content of the paper, we can say a few words about the choices we had to make in the formalization process. When a number theorist defines the ring of adèles, they will typically let K be a *global* field and define its ring of adèles as the restricted product $\mathbb{A}_K := \prod_v' K_v$, where v runs over the set of *places* of K, that is, over the equivalence classes of nontrivial absolute values on K.

The first thing that we observe is that, in Lean, we need to treat number fields and function fields separately. Suppose first that K is a number field. The next observation is that we cannot currently construct the type of all places of K, and we need to use different tools to work with the archimedean and nonarchimedean places.

Similarly, while in the function field case all places are nonarchimedean, we do not yet have a convenient way to obtain the set of all places of a function field (this would require an algebraic geometric interpretation not yet formalized); instead, we have to distinguish between the places coming from the ring of integers of the field place, and the places at infinity (those coming from the absolute value $|\cdot|_\infty$ on $k(t)$).

However, these descriptions show that, regardless of whether K is a number field or a function field, its finite adèle ring can be described as the restricted product $\mathbb{A}_{K,f} := \prod_v' K_v$, where v runs over the maximal ideals of the ring of integers of K. In both cases, this ring of integers is a Dedekind domain. Moreover, the definition $\mathbb{A}_{R,f} := \prod_v' \mathrm{Frac}(R)_v$ makes sense for any Dedekind domain R with field of fractions $\mathrm{Frac}(R)$, regardless of whether $\mathrm{Frac}(R)$ is a global field.

We therefore chose to define `finite_adele_ring'` for any Dedekind domain R. This allowed us to unify the number and function field cases in a big part of the theory, and to show that some properties of $\mathbb{A}_{R,f}$ hold in greater generality than the one typically considered in informal mathematics.

6.2 Implementation comments

In this section we discuss some technical details of our formalization. The first one has to do with the universe in which the Dedekind domain R and its function field K are defined. Lean is based on a version of dependent type theory with a countable hierarchy of non-cumulative universes: `Type` (short for `Type 0`) is the universe of small or ordinary types, `Type 1` is a larger universe of types which contains `Type` as an element, and, in general, for any natural number $n > 0$, there is a `Type n` which contains `Type n - 1` as an element. There is an extra type, called `Prop`, which has some special properties. We can declare universe variables explicitly, or use `Type*` to avoid naming the arbitrary universe.

```
universe u
variables {T : Type u} {S : Type*}
```

Dedekind domains and their fields of fractions can be defined over any universe, as we did at the beginning of Section 2.1.

```
variables (R : Type*) [comm_ring R] [is_domain R] [is_dedekind_domain R]
  {K : Type*} [field K] [algebra R K] [is_fraction_ring R K]
```

However, the ring $\mathbb{Z}$ has type `Type`, and hence so does `with_zero (multiplicative ℤ)`. When we started to formalize adic valuations on Dedekind domains, `mathlib`'s definition of the class `valued` required the ring `R` and the `linear_ordered_comm_group_with_zero` Γ_0 to live in the same universe:

```
universe u
class valued (R : Type u) [ring R] :=
(Γ₀ : Type u)
[grp : linear_ordered_comm_group_with_zero Γ₀]
(v : valuation R Γ₀)
```

This forced us to require our Dedekind domain R and its field of fractions K to live in `Type`. However, we observed that the definition of `valued` could be generalized to allow Γ_0 to have a different type than the ring `R`, without any negative consequences to the library. After this observation and some input from the `mathlib` community, in March 2022 the definition of `valued` was changed to the following:

```
class valued (R : Type u) [ring R] (Γ₀ : out_param (Type v))
  [linear_ordered_comm_group_with_zero Γ₀] :=
(v : valuation R Γ₀)
```

which in particular allows `R` and Γ_0 to live in different universes. With this design, we can use the variable declaration below to indicate that the ring `R` has a canonical valuation with values on Γ_0. The `out_param` in the definition of the class `valued` has the effect that, when proving lemmas about the valued structure on `R`, Lean will pick Γ_0 based on the valued instance it found.

```
universes u v
variables {R : Type u} [ring R] {Γ₀ : Type v }
  [linear_ordered_comm_group_with_zero Γ₀] [valued R Γ₀]
```

Secondly, we ran into a computability issue in Lean 3. A function is computable if there is an algorithm that can produce the output corresponding to every possible input. Every computable definition in Lean 3 is compiled to bytecode at definition time. However, functions that rely on the axiom of choice and therefore do not admit a computational interpretation are also allowed in Lean. These functions have to be declared using the `noncomputable` modifier.

When a definition is stated in Lean 3, a computability check is deployed, even if the definition has been marked as `noncomputable`. If a computable definition has been labeled as `noncomputable`, or a noncomputable definition is missing the label, an error will be raised.

We found that in some definitions, the computability check was causing unexpected timeouts. We would like to thank Gabriel Ebner for finding the cause of these errors and providing a first solution to it, the `force_noncomputable` definition, with a corresponding `simp` lemma.

```
noncomputable def force_noncomputable {α : Sort*} (a : α) : α :=
function.const _ a (classical.choice ⟨a⟩)
@[simp] lemma force_noncomputable_def {α} (a : α) : force_noncomputable a = a :=
rfl
```

The trick is that, given a value `a`, `force_noncomputable` uses the axiom of choice to return an element of the singleton `{a}`. That is, it returns the original value; however, since the axiom of choice is explicitly invoked, the definition is noncomputable. When `force_noncomputable`

is pre-composed with any definition in Lean, the new definition is noncomputable (regardless of whether the original definition was), and the computability check is able to identify this without timing out.

In March 2022, the Lean mantainers added a new `noncomputable!` modifier. Definitions with this label do not have their computability checked, get marked as noncomputable when added to the environment, and do not get compiled at definition time. Hence this modifier can be used to solve the issue we found above (in which the computability check was timing out), and it also helps in the case where a definition is correctly identified as computable but the compiler times out when producing the corresponding bytecode.

As an example, the definition of the coercion map from $\mathbb{A}_{R,f}$ to $\prod_v K_v$ was causing an "deterministic timeout" error, which was solved by using the `noncomputable!` modifier.

```
noncomputable! def coe' : (finite_adele_ring' R K) → K_hat R K := λ x, x.val
```

6.3 Future work

There are several natural directions for future formalization work stemming from this project. We list some of them, starting with the most immediate goals.

- Show that the two definitions of the finite adèle ring formalized in Section 2.2 give isomorphic topological rings. Constructing an isomorphism of rings between them will be easy, but checking that it is a homeomorphism will require some work.
- Formalize topological results about the adèle ring and the idèle group, such as the proof that $\mathbb{A}_K$ is locally compact and contains K as a discrete co-compact subring.
- Given a finite extension L/K of global fields, formalize the isomorphism $\mathbb{A}_L \simeq L \otimes \mathbb{A}_K$ and its consequences.
- Keep stating, and eventually proving, results from class field theory.
- Formalize Tate's thesis.

More generally, having the definitions of $\mathbb{A}_K$ and $\mathbb{I}_K$ opens the door to formalizing concepts and results used in state-of-the-art number theory, including the definition of automorphic forms [5] and the statement of the Langlands correspondence [11]. Note that only some cases of the Langlands correspondence have been proven, and the Langlands program is currently one of the main research areas in number theory.

References

1 Emil Artin and John Tate. *Class Field Theory*. W. A. Benjamin, New York, 1967.

2 Emil Artin and George Whaples. Axiomatic Characterization of Fields by the Product Formula for Valuations. *Bulletin of the American Mathematical Society*, 51(7):469–492, 1945. URL: `https://mathscinet.ams.org/mathscinet-getitem?mr=MR0013145`.

3 Jeremy Avigad, Leonardo de Moura, and Soonho Kong. *Theorem Proving in Lean*. Carnegie Mellon University, 2021. Release 3.23.0. URL: `https://leanprover.github.io/theorem_proving_in_lean/`.

4 Anne Baanen, Sander R. Dahmen, Ashvni Narayanan, and Filippo A. E. Nuccio Mortarino Majno di Capriglio. A Formalization of Dedekind Domains and Class Groups of Global Fields. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ITP.2021.5`.

5 Daniel Bump. *Automorphic Forms and Representations*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, Cambridge, 1997. `doi:10.1017/CBO9780511609572`.

6 Kevin Buzzard, Johan Commelin, and Patrick Massot. Formalising Perfectoid Spaces. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 299–312. ACM, 2020. doi:10.1145/3372885.3373830.

7 Mario Carneiro. *The Type Theory of Lean*. Springer, Berlin, Heidelberg, 2019. Master thesis. URL: https://github.com/digama0/lean-type-theory/releases.

8 J. W. S. Cassels and A. Fröhlich (eds.). *Algebraic Number Theory*. Academic Press, London; Thompson Book Co., Inc., Washington, D.C., 1967.

9 L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean Theorem Prover (System Description). In Felty A. and Middeldorp A., editors, *Automated Deduction - CADE-25*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, Cham, 2015. doi:10.1007/978-3-319-21401-6_26.

10 Gerald J. Janusz. *Algebraic Number Fields*, volume 55 of *Pure and Applied Mathematics*. Academic Press, London, 2nd edition, 1996.

11 R. P. Langlands. Problems in the Theory of Automorphic Forms. In *Lectures in Modern Analysis and Applications III*, volume 170 of *Lecture Notes in Mathematics*, pages 18–61. Springer, Berlin, Heidelberg, 1970. doi:10.1007/BFb0079065.

12 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. Association for Computing Machinery. doi:10.1145/3293880.3294089.

13 J. S. Milne. Class Field Theory (v4.03), 2020. URL: https://www.jmilne.org/math/CourseNotes/CFT.pdf.

14 Jürgen Neukirch. *Algebraic Number Theory*. Springer, Berlin, Heidelberg, 1999. doi:10.1007/978-3-662-03983-0.

15 Álvaro Pelayo, Vladimir Voevodsky, and Michael A. Warren. A univalent formalization of the p-adic numbers. *Mathematical Structures in Computer Science*, 25(5):1147–1171, 2015. doi:10.1017/S0960129514000541.

16 Henning Stichtenoth. *Algebraic Function Fields and Codes*. Universitext. Springer, 1993. URL: https://dblp.org/rec/books/daglib/0084861.

17 Floris van Doorn. Formalized Haar Measure. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:17, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ITP.2021.18.

A Verified Cyclicity Checker

For Theories with Overloaded Constants

Arve Gengelbach
KTH Royal Institute of Technology, Stockholm, Sweden

Johannes Åman Pohjola
University of New South Wales, Sydney, Australia

Abstract

Non-terminating (dependencies of) definitions can lead to logical contradictions, for example when defining a boolean constant as its own negation. Some proof assistants thus detect and disallow non-terminating definitions. Termination is generally undecidable when constants may have different definitions at different type instances, which is called *(ad-hoc) overloading*. The Isabelle/HOL proof assistant supports overloading of constant definitions, but relies on an unclear foundation for this critical termination check. With this paper we aim to close this gap: we present a mechanised proof that, for restricted overloading, non-terminating definitions are of a detectable cyclic shape, and we describe a mechanised algorithm with its correctness proof. In addition we demonstrate this cyclicity checker on parts of the Isabelle/HOL main library. Furthermore, we introduce the first-ever formally verified kernel of a proof assistant for higher-order logic with overloaded definitions. All our results are formalised in the HOL4 theorem prover.

2012 ACM Subject Classification Software and its engineering → Correctness; Theory of computation → Program verification; Theory of computation → Higher order logic; Theory of computation → Graph algorithms analysis

Keywords and phrases cyclicity, non-termination, ad-hoc overloading, definitions, Isabelle/HOL

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.15

Related Version *Previous Version*: `http://www.it.uu.se/research/publications/reports/2021-001/` [7]

Supplementary Material *Software (Source Code)*: `https://code.cakeml.org/tree/master/candle/overloading`

Acknowledgements We are grateful to Andrei Popescu and Tjark Weber for technical discussion. We thank the anonymous reviewers for their constructive feedback. This work was begun while the first author was affiliated with Uppsala University, Sweden.

1 Introduction

For a consistent logical foundation, a theorem prover should only accept contradiction-free definitions. Although the logical foundations of many theorem provers are well studied, e. g. [21, 10, 13, 20, 5], still unverified implementations may allow proof of contradiction, e. g. by contradictory definitions [18, 14, 15].

Contradictory definitions can be avoided if each defined symbols and its dependants span a graph with only finite chains, i. e. if the so-called *dependency* graph is *terminating*. We showed this in earlier work for a variant of higher-order logic (HOL) [1]. For more expressive definitions, termination of the dependency graph is generally undecidable, which Obua [18] showed if a symbol (like a constant or a type) may be defined at different type instances, so-called *(ad-hoc) overloading*. Overloaded definitions enable recursion through types and their dependency graphs are generally infinite.

In-logic overloading of constant definitions distinguishes the Isabelle/HOL theorem prover from others, and permits Haskell-style type classes [26], which enable types to carry structure with operations, e. g. a monoid type class with composition and a neutral element [24].

We illustrate non-terminating definitions by an example,[1] that enabled proof of contradiction in an earlier version of Isabelle/HOL. Assume a theory with the three polymorphic constants $c_{\alpha\ \mathsf{list}\to\mathsf{Bool}}$, $d_{(\alpha\times\beta)\to\mathsf{Bool}}$ and $\mathsf{undefined}_\alpha$, and the following two definitions.

$$c\,(x_{\alpha\ \mathsf{list}}) \equiv d(\mathsf{undefined}_{\alpha\times\alpha}) \qquad\qquad d\,(x_{\alpha\times\mathsf{nat}}) \equiv \neg c([\mathsf{undefined}_\alpha])$$

Here, c with the argument $x_{\alpha\ \mathsf{list}}$ is defined in terms of d, and d with the argument $x_{\alpha\times\mathsf{nat}}$ is defined in terms of c. For the type instances of c at type $\mathsf{nat\ list} \to \mathsf{Bool}$ and of d at type $(\mathsf{nat} \times \mathsf{nat}) \to \mathsf{Bool}$, we obtain a non-terminating sequence and also a contradiction.

$$c\,([\mathsf{undefined}_{\mathsf{nat}}]) = d\,(\mathsf{undefined}_{\mathsf{nat}\times\mathsf{nat}}) = \neg c\,([\mathsf{undefined}_{\mathsf{nat}}])$$

Non-termination and the contradiction only surface after type variable instantiation. (To make this an overloaded definition, we could instead declare c of type $\alpha \to \mathsf{Bool}$ and in the definitions replace each name d by c.)

In order to facilitate the check of termination of overloaded definitions in Isabelle/HOL, Kunčar's work suggests that so-called *composable* non-terminating dependency graphs have a structure [14]. Detecting this structure in dependency relations that additionally are *orthogonal* is decidable. Kunčar defines a dependency relation orthogonal if it is functional, i. e. any symbol only depends on one other symbol. Despite these findings, we are unaware of any formalisation of the theorems.

In this paper, we present a complete formalisation of the theory that any non-terminating dependency relation contains cycles. We innovate to resolve a problem in Kunčar's argument for the main theorem that stems from an incorrect size comparison of a type prior to and after type instantiation. Further, we discover that Kunčar's restriction to orthogonal dependency relations is not satisfiable by dependencies stemming from definitions, and invent a cyclicity checker algorithm of our own. By formal proof our cyclicity checker can correctly calculate non-termination for composable dependency relations. We profit from the rich infrastructure around the CakeML language [23, 11] to synthesise a correct binary cyclicity checker. This checker shows that extracts of dependencies from Isabelle/HOL theories are composable and acyclic. If the checker detects acyclic dependencies of a theory of definitions, then by its correctness guarantees, we can discharge the assumption of non-terminating definitions, and obtain a consistent theory of definitions [1] with model-theoretic conservativity guaranties [8]. Altogether, we compose our verified cyclicity checker with the infrastructure from prior work [1, 8], to obtain a formally verified theorem prover kernel for HOL with overloading, that has the mentioned mechanised foundational properties.

All our definitions and theorems are formalised in the HOL4 theorem prover [19], and available online.[2]

The remainder of this paper is structured as follows. In Section 2 we discuss the syntax, to give an insight into the theory of non-terminating dependencies in Section 3. The full technical account of the proof are documented in a technical report [7]. Our algorithm is presented in Section 4 and illustrated with examples. We discuss potential future optimisations in Section 6 and conclude with Section 7.

---

[1] A variant of this example is attributed to Popescu by Kunčar [14].

[2] Our mechanisation of Sections 3 and 4 are part of the CakeML development repository at `https://code.cakeml.org/tree/master/candle/overloading`.

2 Syntax

In this section we define the syntax that we use throughout the paper. Type substitutions are one essential component that we define in Section 2.2. We define symbols in Section 2.3 to be types and typed constants (i. e. tuples of names and types). We extend all the notions to symbols (Section 2.3) and introduce dependency relations on symbols (Section 2.4). In Section 2.5 we define composability of dependency relations.

2.1 Notation

The definitions and theorems are mostly generated from our HOL4 formalisation, with theorems prefixed by $\vdash$. Constants in HOL4 are printed in sans-serif font and variables in *italic face*, i. e. SUC n for the successor of a natural number n. We freely move between lists and sets, and equate $X\ x$ with $x \in X$. The list functions last, front, $(++)$ and null denote the last element, all elements but the last, append and emptiness, respectively. A colon denotes the type of a term, as in $(\mathsf{last} : \alpha\ \mathsf{list} \Rightarrow \alpha)$.

The sum type is written $\alpha + \beta$, with disjoint branches $(\mathsf{INL} : \alpha \Rightarrow \alpha + \beta)$ and $(\mathsf{INR} : \beta \Rightarrow \alpha + \beta)$.

2.2 Types and Type Substitutions

Types are rank 1 polymorphic, and follow the grammar:

$$\mathsf{type} = \mathsf{Tyvar\ string} \mid \mathsf{Tyapp\ string\ (type\ list)}$$

The set of all type variables of a type ty is $\mathsf{FV}(ty)$. The size of a type is defined as

$$\mathsf{size}\ (\mathsf{Tyvar}\ m) \stackrel{\mathsf{def}}{=} 1 \qquad \mathsf{size}\ (\mathsf{Tyapp}\ m\ tys) \stackrel{\mathsf{def}}{=} 1 + |tys| + \mathsf{sum}\ (\mathsf{map\ size}\ tys).$$

We identify Tyvar «a», Tyvar «b», Tyvar «c» with α, β, γ respectively, and abbreviate common types like Tyapp «list» $[\alpha]$ with α list, and Tyapp «bool» [] with Bool, and function types Tyapp «fun» $[\alpha, \beta]$ with $\alpha \to \beta$. By the definition of size nullary types and type variables have the same size, e. g. size Bool = size $\alpha = 1$.

2.2.1 Type Substitutions

A *type substitution* ρ is a list of pairs of types such that $(y, \mathsf{Tyvar}\ x) \in \rho$ whenever $\rho\ (\mathsf{Tyvar}\ x) = y$. Duplicates w. r. t. the second component within the list ρ are ignored.

Type substitutions extend to types homomorphically, that is type substitutions instantiate type variables in a type. A type ty' is an *instance* of a more general type ty, written $ty \geq ty'$, if there exists a type substitution ρ such that $ty' = \rho\ ty$. We have implemented and verified an algorithm that computes whether or not a type is an instance of another type.

In addition, we implement and verify a first-order unification algorithm (from [3, § 2.3.2]) that produces an idempotent, most general unifier of two types (if one exists). A type unification algorithm can be used to calculate if two types have no common type instance, i. e. are *orthogonal* (written with infix #):

$$ty_1 \# ty_2 \stackrel{\mathsf{def}}{=} \neg\exists\, ty.\ ty_1 \geq ty \wedge ty_2 \geq ty$$

The types $\alpha \times \alpha$ and $\alpha \times \mathsf{nat}$ have the common instance $\mathsf{nat} \times \mathsf{nat}$, and are not orthogonal.

2.2.2 Variable Renamings

A special kind of type substitution η is a *renaming* of type variables, written $\mathsf{var\_renaming}\ \eta$. It is bijective and acts as the identity everywhere except on the subset of its domain $\mathsf{dom}\ \eta$, and only renames type variables.

$$\mathsf{var\_renaming}\ \eta \stackrel{\text{def}}{=} (\mathsf{img}\ \eta) = (\mathsf{dom}\ \eta) \wedge (\forall x.\ x \in \mathsf{img}\ \eta \Rightarrow \exists a.\ x = \mathsf{Tyvar}\ a) \wedge \mathsf{all\_distinct}\ (\mathsf{dom}\ \eta)$$

For instance, the type substitution $\eta = \{\alpha \mapsto \beta, \beta \mapsto \gamma, \gamma \mapsto \alpha\}$ is a renaming.

Two types x and y are *equivalent* if they differ by a renaming.

$$x \approx y \stackrel{\text{def}}{=} \exists \eta.\ \mathsf{var\_renaming}\ \eta \wedge x = \eta\ y$$

For instance, the types $\alpha\ \mathsf{list}$ and $\beta\ \mathsf{list}$ are equivalent by the renaming $\eta = \{\alpha \mapsto \beta, \beta \mapsto \alpha\}$. The relation $\approx$ is an equivalence (reflexive, symmetric and transitive).

2.3 Typed Constants and Symbols

Constants consist of a name and a type. *Symbols* are the sum type whose left leaves are types, $\mathsf{INL}\ (ty : \mathsf{type})$, and whose right leaves are typed constants, $\mathsf{INR}\ (\mathsf{Const}\ (c : \mathsf{string})\ (ty : \mathsf{type}))$. For a constant we sometimes write the type as an index, like c_{Bool} for a constant c of type Bool.

We lift all notions (like size, FV, type substitutions, $\leq$, $\#$, $\approx$) from types to constants in the obvious manner, like $\mathsf{size}(c_\tau) = \mathsf{size}(\tau)$ for a constant c_τ of type τ.

2.4 Dependency Relations

A dependency relation $\rightsquigarrow$ is a binary relation on symbols, i. e. types and typed constants. For any relation $\mathcal{R}$ we interchangeably use the infix notation $x\,\mathcal{R}\,y$, and $(x, y) \in \mathcal{R}$, even when $\mathcal{R}$ is internally represented as a list. Definitions imply dependencies, as is described elsewhere [15, 1]. For example, the two definitions

$$c\,(x_{\alpha\ \mathsf{list}}) \equiv d(\mathsf{undefined}_{\alpha\times\alpha}) \qquad \text{and} \qquad d\,(x_{\alpha\times\mathsf{nat}}) \equiv \neg c([\mathsf{undefined}_\alpha])$$

from Section 1 entail a dependency relation $\curvearrowright$ that contains the two elements

$$c_{\alpha\ \mathsf{list}\to\mathsf{Bool}} \curvearrowright d_{(\alpha\times\alpha)\to\mathsf{Bool}} \qquad \text{and} \qquad d_{(\alpha\times\mathsf{nat})\to\mathsf{Bool}} \curvearrowright c_{\alpha\ \mathsf{list}\to\mathsf{Bool}}\,.$$

We refer to this particular dependency relation as the bold-face green-coloured arrow $\curvearrowright$ throughout this Section 2.

We write $\mathcal{R}^+$ for transitive closure, and $\mathcal{R}^*$ for reflexive-transitive closure. With $\mathcal{R}^n$ we denote the n-times iterated composition relation $\underbrace{\mathcal{R} \cdots \mathcal{R}}_{n\text{-times}}$.

2.4.1 Monotone Relations

A relation is *monotone* if each type variable of the second argument is contained in the type variables of the first argument.

$$\mathsf{monotone}\ \mathcal{R} \stackrel{\text{def}}{=} \forall x\ y.\ (x,y) \in \mathcal{R} \Rightarrow \mathsf{FV}\ y \subseteq \mathsf{FV}\ x$$

For dependencies arising from definitions, monotonicity is a natural assumption as it means that each type variable of the right-hand side occurs in the defined symbol's type. For example, attempting to define a constant e_{nat} as the cardinality of the universe of type α, i. e. $\mathsf{CARD}\,\mathcal{U}(:\alpha)$, entails a non-monotone dependency $e_{\mathsf{nat}} \rightsquigarrow \alpha$. This attempted definition is unsound as $\mathsf{CARD}\,\mathcal{U}(:\mathsf{bool}) = 2$ and there certainly are types of different cardinality. From this point onwards all dependency relations under consideration are monotone, unless otherwise stated.

2.4.2 Type-substitutive Closure

With overloading, non-termination may stem from recursion through the types of constants. Thus the analysis needs to consider type instances of dependencies. For a binary relation $\mathcal{R}$ on symbols, two symbols x and y are in the type-substitutive closure relation $x\,\mathcal{R}^{\downarrow}\,y$ if there exists a type substitution ρ such that $(\rho\ x)\,\mathcal{R}\,(\rho\ y)$. For example, $c_{\alpha\ \mathsf{list}\to\mathsf{Bool}} \curvearrowright d_{(\alpha\times\alpha)\to\mathsf{Bool}}$ implies $c_{\mathsf{nat\ list}\to\mathsf{Bool}} \curvearrowright^{\downarrow} d_{(\mathsf{nat}\times\mathsf{nat})\to\mathsf{Bool}}$. The type-substitutive closure of a dependency relation is infinite if the dependency relation contains a type variable.

2.4.3 Paths of Dependencies

In this section we define *solutions*, that we later use to witness elements in $\rightsquigarrow^{\downarrow *}$, *paths* as witnesses for elements in $\rightsquigarrow\rightsquigarrow^{\downarrow *}$ modulo renaming, and introduce paths of fixed length.

A *solution* to a list of pairs of symbols pq is a list of type substitutions ρ, with the constraint that applying the respective type substitution the i-th component $(\rho_i)\ (\mathsf{snd}\ (pq_i))$ equals the next $(\rho_{i+1})\ (\mathsf{fst}\ (pq_{i+1}))$.

$$\begin{aligned}&\mathsf{sol\_seq}\ \rho\ pq \stackrel{\mathsf{def}}{=}\\ &\quad \mathsf{wellformed}\ pq \wedge |\rho| = |pq| \wedge \forall\, i.\ i+1 < |\rho| \Rightarrow (\rho_i)\ (\mathsf{snd}\ (pq_i)) = (\rho_{i+1})\ (\mathsf{fst}\ (pq_{i+1}))\end{aligned}$$

The wellformed predicate restricts the sequences to symbols (and could instead have been realised by the type system). For example, the sequence of length two of the pairs $(c_{\alpha\ \mathsf{list}\to\mathsf{Bool}}, d_{(\alpha\times\alpha)\to\mathsf{Bool}})$ and $(d_{(\alpha\times\mathsf{nat})\to\mathsf{Bool}}, c_{\alpha\ \mathsf{list}\to\mathsf{Bool}})$ has a solution ρ with the compontents $\rho_0 = \rho_1 = \alpha \mapsto \mathsf{nat}$. This solution witnesses $c_{\mathsf{nat\ list}\to\mathsf{Bool}} \curvearrowright^{\downarrow} d_{(\mathsf{nat}\times\mathsf{nat})\to\mathsf{Bool}} \curvearrowright^{\downarrow} c_{\mathsf{nat\ list}\to\mathsf{Bool}}$, cf. Section 2.4.

We are mainly interested in solutions to lists of dependencies, i. e. where $pq \subseteq \rightsquigarrow$ holds.

A *path* through a dependency relation $\rightsquigarrow$ is a list of pairs of symbols pq in $\rightsquigarrow$, that have a solution ρ, such that ρ_0 is invertible on the type variables of $\mathsf{fst}\ (pq_0)$.

$$\mathsf{path} \rightsquigarrow \rho\ pq \stackrel{\mathsf{def}}{=} 0 < |\rho| \wedge pq \subseteq \rightsquigarrow \wedge \mathsf{invertible\_on}\ (\rho_0)\ (\mathsf{FV}\ (\mathsf{fst}\ (pq_0))) \wedge \mathsf{sol\_seq}\ \rho\ pq$$

Hence, modulo renaming, any path $\mathsf{path} \rightsquigarrow \rho\ pq$, corresponds to an element of $\rightsquigarrow\rightsquigarrow^{\downarrow *}$, namely

$$(\rho_0)\ (\mathsf{fst}\ (pq_0))\ \rightsquigarrow\rightsquigarrow^{\downarrow *}\ (\mathsf{last}\ \rho)\ (\mathsf{snd}\ (\mathsf{last}\ pq)).$$

A cyclicity checker that calculates paths will fix the first element of a path and try to extend it. For the verification, we write $\mathsf{has\_path\_to} \rightsquigarrow n\ x\ y$ when there exists a path of length $n > 0$ from x to y, namely $x \rightsquigarrow (\rightsquigarrow^{\downarrow})^{n-1}\ y$.

$$\begin{aligned}&\mathsf{has\_path\_to} \rightsquigarrow n\ x\ y \stackrel{\mathsf{def}}{=}\\ &\quad \exists\, \rho\ pq.\ \mathsf{path} \rightsquigarrow \rho\ pq \wedge n = |pq| \wedge x = \mathsf{fst}\ (pq_0) \wedge y \approx (\mathsf{last}\ \rho)\ (\mathsf{snd}\ (\mathsf{last}\ pq))\end{aligned}$$

We define $x = \mathsf{fst}\ (pq_0)$ instead of $x = (\rho_0)\ (\mathsf{fst}\ (pq_0))$, because due to monotonicity the sequence $({\rho_0}^{-1} \circ \rho_i)$ is a solution with $({\rho_0}^{-1} \circ \rho_0) = \mathsf{id}$ on $\mathsf{FV}\ (\mathsf{fst}\ (pq_0))$ and ${\rho_0}^{-1}$ only renames type variables of $(\mathsf{last}\ \rho)\ (\mathsf{snd}\ (\mathsf{last}\ pq))$.

For a fixed x and length n we can freely rename variables of y:

$$\vdash \mathsf{var\_renaming}\ \eta \Rightarrow (\mathsf{has\_path\_to} \rightsquigarrow n\ x\ y \iff \mathsf{has\_path\_to} \rightsquigarrow n\ x\ (\eta\ y))$$

2.4.4 Terminating and Cyclic Relations

A relation $\mathcal{R}$ is *terminating* if its converse relation has no infinite chains, i. e. is well-founded.

$$\mathsf{terminating}\ \mathcal{R} \stackrel{\mathsf{def}}{=} \mathsf{WF}\ (\lambda\, x\ y.\ (y,x) \in \mathcal{R})$$

For a dependency relation $\rightsquigarrow$, termination of its type-substitutive transitive closure $\rightsquigarrow^{\downarrow+}$ is difficult to characterise, but for certain relations we prove "not cyclic implies terminating" in Section 3.6.

A path of length n is *cyclic*, written $\mathsf{cyclic\_len}\ \rightsquigarrow\ n$, if its last element is an instance of the first, namely $x\ \rightsquigarrow\ (\rightsquigarrow^{\downarrow})^{n-1}\ y$ with $x \geq y$, modulo renaming. A dependency relation is *cyclic*, written $\mathsf{cyclic\_dep}\ \rightsquigarrow$, if it has a cyclic path.

$$\mathsf{cyclic\_len}\ \rightsquigarrow\ n \stackrel{\text{def}}{=} \exists\, x\ y.\ \mathsf{has\_path\_to}\ \rightsquigarrow\ n\ x\ y \wedge x \geq y$$
$$\mathsf{cyclic\_dep}\ \rightsquigarrow \stackrel{\text{def}}{=} \exists\, n.\ \mathsf{cyclic\_len}\ \rightsquigarrow\ n$$

A cyclic relation is not terminating, because a cycle, i. e. $x\ \rightsquigarrow\rightsquigarrow^{\downarrow*}\ y$ with $y = \rho\ x$, entails non-terminating dependencies $x\ \rightsquigarrow\rightsquigarrow^{\downarrow*}\ (\rho^i\ x)$ for any $i > 0$.

The converse is not true. As an example, the closure $\curvearrowright^{\downarrow+}$ (cf. Section 2.4) is non-terminating, as witnessed by

$$c_{\mathsf{nat\ list}\to\mathsf{Bool}} \curvearrowright^{\downarrow} d_{(\mathsf{nat}\times\mathsf{nat})\to\mathsf{Bool}} \curvearrowright^{\downarrow} c_{\mathsf{nat\ list}\to\mathsf{Bool}} \curvearrowright^{\downarrow} \cdots,$$

however all paths in $\curvearrowright\curvearrowright^{\downarrow*}$ are of length one and not cyclic.

2.5 Composability

Composability is a central concept that makes checking for termination of dependency relations more feasible. We first give its definition and then exemplify the intuition.

A path $x\ \rightsquigarrow\ (\rightsquigarrow^{\downarrow})^n\ y$ from x to y is *composable* if for all $p \rightsquigarrow q$, either $y \leq p$ or $y \geq p$, or otherwise y and p are orthogonal $y\#p$. We quantify over x and y, and formally write that all paths of length n within $\rightsquigarrow$ are composable as $\mathsf{composable\_len}\ \rightsquigarrow\ n$:

$$\mathsf{composable\_len}\ \rightsquigarrow\ n \stackrel{\text{def}}{=}$$
$$\forall\, x\ y\ p\ q.\ \mathsf{has\_path\_to}\ \rightsquigarrow\ n\ x\ y \wedge (p,q) \in \rightsquigarrow \Rightarrow y \geq p \vee p \geq y \vee y\ \#\ p$$

A dependency relation $\rightsquigarrow$ is *composable*, denoted by $\mathsf{composable\_dep}\ \rightsquigarrow$, if all paths are. The relation $\curvearrowright$ (cf. Section 2.4) is not composable, because attempting to extend the dependency $c_{\alpha\ \mathsf{list}\to\mathsf{Bool}} \curvearrowright d_{(\alpha\times\alpha)\to\mathsf{Bool}}$ by $d_{(\alpha\times\mathsf{nat})\to\mathsf{Bool}} \curvearrowright c_{\alpha\ \mathsf{list}\to\mathsf{Bool}}$ contradicts composability. The constants $d_{(\alpha\times\alpha)\to\mathsf{Bool}}$ and $d_{(\alpha\times\mathsf{nat})\to\mathsf{Bool}}$ are not orthogonal, because their types are not. If d is instead defined at a more general type, like $d_{\alpha\times\alpha\to\mathsf{Bool}} \equiv \ldots$ then the dependency relation becomes composable, but remains non-terminating.

The implications of composability for a user are discussed in the Isabelle/Isar reference manual [25, § 5.9]. Composability requires all instances of overloaded constants to occur either at their most general type, or with all type variables instantiated.

3 Theory

Our main theoretic contribution is a formal proof that any monotone, composable, and finite dependency relation $\rightsquigarrow$ has a not terminating type-substitutive closure if, and only if the relation is cyclic.

$$\vdash \mathsf{monotone}\ \rightsquigarrow \wedge\ \mathsf{composable\_dep}\ \rightsquigarrow \wedge\ \mathsf{wellformed}\ \rightsquigarrow \wedge\ \mathsf{finite}\ (\rightsquigarrow) \Rightarrow$$
$$(\neg\mathsf{terminating}\ \rightsquigarrow^{\downarrow+} \iff \mathsf{cyclic\_dep}\ \rightsquigarrow)$$

The direction "cyclic implies not terminating" follows without composable_dep $\rightsquigarrow$, as sketched in Section 2.4.4. In the following we discuss the interesting converse direction, which in other words says, that for a finite, composable dependency relation $\rightsquigarrow$ an infinite sequence in $\rightsquigarrow^{\downarrow+}$ gives rise to a cycle in $\rightsquigarrow\rightsquigarrow^{\downarrow*}$ of particular shape. (Generally, the type-substitutive closure $\rightsquigarrow^{\downarrow}$ is already infinite if any type variables occur in $\rightsquigarrow$.)

Our proof arguments deviate from Kunčar's due to a false lemma [14, Lemma 3.1b], whose proof argues, that a type substitution that is not the identity should instantiate at least one type variable by a type. This argument misses the effect of renamings, and requires overall changes to the proof, although following Kunčar's general proof idea. We discuss how we circumvent this problem in Section 3.1.

The remaining section is structured as follows. We introduce some background concepts, like the effect of type substitution on the type variables FV and type size size in Section 3.1, and solutions that are *most general* in Section 3.2. Thereafter we continue the two main proof arguments. First, in Section 3.3, with composability every sequence in $\rightsquigarrow^{\downarrow+}$ implies that there is some corresponding sequence in $\rightsquigarrow\rightsquigarrow^{\downarrow*}$ [14, Lemma 5.11]. As a consequence (in Section 3.4) a solution can only be prolonged in two ways, either through a $\leq$-*extension* or a *strict* $\geq$-*extension*. Second, in Section 3.5, by composability the shape of any infinite sequence in $\rightsquigarrow\rightsquigarrow^{\downarrow*}$ can be narrowed down further. We combine these two main arguments to a proof sketch in Section 3.6.

For full technical details, this work is complemented by a technical report [7].

3.1 Type Instantiation, Type Size and Type Variables

In this section we describe why we deviate from Kunčar's original proof. One argument claims that non-identical type instantiation increases the size of a type [14, Lemma 3.1b]. A counterexample to this claim[3] is the instantiation of a function type $\alpha \to \beta$ with the type substitution $\rho = \alpha \mapsto \beta$, that unifies the two type variables $\rho(\alpha \to \beta) = \beta \to \beta$ and does not change the size: $\mathsf{size}(\alpha \to \beta) = \mathsf{size}(\beta \to \beta)$.

We circumnavigate this problem by observing change of size and number of type variables [7, Lemma 4]. A type substitution ρ may increase the size:

$$\vdash \mathsf{size}\ p \leq \mathsf{size}\ (\rho\ p)$$

However, if ρ is not invertible on $\mathsf{FV}(p)$ and the size is invariant $\mathsf{size}(p) = \mathsf{size}(\rho(p))$ then the number of type variables decreases strictly $|\mathsf{FV}(p)| > |\mathsf{FV}(\rho(p))|$.

That ρ has no inverse although the size is unchanged means that ρ only unifies type variables $\alpha, \beta \in \mathsf{FV}(p)$ (such that $\rho(\alpha) = \rho(\beta)$) and instantiates type variables to nullary types. Both nullary types and type variables have the same size, e. g. $\mathsf{size}\ \mathsf{Bool} = \mathsf{size}\ \alpha = 1$.

Assuming $q = \rho(p)$, i. e. q is an instance of p witnessed by ρ, then we can rephrase "ρ is not invertible on $\mathsf{FV}(p)$" as $\neg(q \leq p)$ which means that there is no type substitution that witnesses that $\rho(p)$ is an instance of p.

$$\vdash q \geq p \wedge \neg(p \geq q) \wedge \mathsf{size}\ q = \mathsf{size}\ p \Rightarrow |\mathsf{FV}\ p| < |\mathsf{FV}\ q|$$

Later in Section 3.4 we will call this a *strict* $\geq$-*extension*, as $q \geq p$ and $\neg(q \leq p)$.

[3] Note, that the stated problem also surfaces for Kunčar's slightly different definition of size.

3.2 Most General Solutions

Recall that solutions of dependencies witness elements in $\rightsquigarrow^{\downarrow *}$, as defined in Section 2.4.3. In this section we define *most general* solutions. For example, the sequence $(\alpha\ \mathsf{list}, \alpha), (\beta\ \mathsf{list}, \beta)$ has the solution $\rho_0 = (\alpha \mapsto \mathsf{Bool\ list}), \rho_1 = (\beta \mapsto \mathsf{Bool})$, which gives rise to the dependencies $\mathsf{Bool\ list\ list} \rightsquigarrow^{\downarrow} \mathsf{Bool\ list} \rightsquigarrow^{\downarrow} \mathsf{Bool}$. However, another more general solution would be $(\alpha \mapsto \beta\ \mathsf{list}), \mathsf{id}$.

A solution $(\rho'_i)_{i\leq n}$ for $(p_i, q_i)_{i\leq n}$ is a *most general solution*, written $\mathsf{mg\_sol\_seq}\ \rho'\ pq$, if any other solution $(\rho_i)_{i\leq n}$ is an instance of $(\rho'_i)_{i\leq n}$, i. e. there exist type substitutions $(\eta_i)_{i\leq n}$ such that $\rho'_i(p_i) = (\eta_i \circ \rho_i)(p_i)$ holds for $i \leq n$. (Note, that $\rho_i(p_i) = (\eta_i \circ \rho_i)(p_i)$ entails $\rho_i(q_i) = (\eta_i \circ \rho_i)(q_i)$ due to monotonicity.)

A solution $(\rho_i)_{i\leq n}$ is most general if ρ_0 is invertible, e. g. when $\rho_0 = \mathsf{id}$.

$$\vdash \mathsf{monotone}\ {\rightsquigarrow} \wedge \mathsf{sol\_seq}\ \rho\ pq \wedge 0 < |pq| \wedge pq \subseteq {\rightsquigarrow} \wedge \mathsf{invertible\_on}\ (\rho_0)\ (\mathsf{FV}\ (\mathsf{fst}\ (pq_0))) \Rightarrow \mathsf{mg\_sol\_seq}\ \rho\ pq$$

Each path is a most general solution, and hence every path in $\rightsquigarrow\rightsquigarrow^{\downarrow *}$ is most general.

For a monotone dependency relation any two most general solutions $(\rho_i)_{i\leq n}$ and $(\rho'_i)_{i\leq n}$ are equivalent up to renaming, i. e. there exists a renaming η with $\rho'_i(p_i) = (\eta \circ \rho_i)(p_i)$ for $i \leq n$. Thus variable names in most general solutions can be freely renamed:

$$\vdash \mathsf{mg\_sol\_seq}\ \rho\ pq \wedge \mathsf{var\_renaming}\ \eta \Rightarrow \mathsf{mg\_sol\_seq}\ (\mathsf{map}\ (\lambda x.\ (\ \eta\ \circ\ x\ ))\ \rho)\ pq.$$

3.3 Restricting to Suffixes of Solutions

The first main implication of composability is, that any sequence in $\rightsquigarrow^{\downarrow +}$ has a corresponding suffix in $\rightsquigarrow\rightsquigarrow^{\downarrow *}$. More precisely, any solution has a most general solution with an invertible type substitution at some index k. After normalising the most general solution ρ', we could even assume that ρ'_k is the identity type substitution.

$$\vdash 0 < |pq| \wedge \mathsf{sol\_seq}\ \rho\ pq \wedge pq \subseteq {\rightsquigarrow} \wedge \mathsf{monotone}\ {\rightsquigarrow} \wedge \mathsf{composable\_dep}\ {\rightsquigarrow} \Rightarrow \exists \rho'\ k.\ \mathsf{mg\_sol\_seq}\ \rho'\ pq \wedge \mathsf{invertible\_on}\ (\rho'_k)\ (\mathsf{FV}\ (\mathsf{fst}\ (pq_k))) \wedge k < |pq|$$

This theorem has an important implication: for $pq = (p_i, q_i)_{i\leq n}$, it entails that the suffix $(\rho'_i)_{k\leq i\leq n}$ is a most general solution for $(p_i, q_i)_{k\leq i\leq n}$, because ρ'_k is invertible and w. l. o. g. a renaming, and hence composability applies to any possible extension of this suffix solution.

3.4 Prolonging a Solution by One Element

Searching for infinite sequences in $\rightsquigarrow^{\downarrow +}$ means at each breadth level n finding all extensions of a sequence of length n by one element in the $\rightsquigarrow^{\downarrow}$ relation. There are four possibilities for extending a solution $(\rho_i)_{i\leq n}$ of a sequence $(p_i, q_i)_{i\leq n}$ in the dependency relation, by one more element $p \rightsquigarrow q$.

Either the sequence cannot be prolonged by an instance of $p \rightsquigarrow q$ because $\rho_n(q_n)$ and p are orthogonal, i. e. $\rho_n(q_n)\#p$, or there are unifying type substitutions σ, σ' such that $\sigma(\rho_n(q_n)) = \sigma'(p)$. The following three cases arise.

1. σ is invertible and $\rho_n(q_n) = \sigma^{-1}(\sigma'(p))$. As we showed [7, Lemma 5], then $(\rho_i)_{i\leq n}, (\sigma^{-1} \circ \sigma')$ is a (most general) solution of $(p_i, q_i)_{i\leq n}, (p, q)$. This also includes the case that additionally σ' is invertible. We say $\leq$-*extension* for this case, as $\rho_n(q_n) \leq p$.
2. The substitution σ' is invertible, and σ is not invertible and $\sigma'^{-1}(\sigma(\rho_n(q_n))) = p$. Then by [7, Lemma 7] the type substitutions $(\sigma'^{-1} \circ \sigma \circ \rho_i)_{i\leq n}, \mathsf{id}$ form a (most general) solution of $(p_i, q_i)_{i\leq n}, (p, q)$. We say *strict* $\geq$-*extension* for this case, as $\rho_n(q_n) \geq p$ and the type substitution $\sigma'^{-1} \circ \sigma$ that witnesses this instantiation is not invertible.

3. Both σ and σ' are not invertible. Then the type substitutions $(\sigma \circ \rho_i)_{i \leq n}, \sigma'$ form a solution of $(p_i, q_i)_{i \leq n}, (p, q)$, which is most general if σ and σ' are most general unifiers.

The following theorem summarises that, assuming composability, the case of Item 3 does not occur. If a solution for the sequence of dependency pairs pq can be extended, then a most general solution of pq, that is invertible at some index k, can be extended in only two manners, that together comprise Item 1 and Item 2.

$$\begin{aligned}\vdash\ &\mathsf{sol\_seq}\ \rho\ (pq +\!\!+ [(p,q)]) \wedge pq \subseteq \rightsquigarrow \wedge (p,q) \in \rightsquigarrow \wedge \mathsf{composable\_dep}\ \rightsquigarrow \wedge \\ &\mathsf{monotone}\ \rightsquigarrow \wedge\ \mathsf{mg\_sol\_seq}\ \rho'\ pq \wedge \mathsf{invertible\_on}\ (\rho'_k)\ (\mathsf{FV}\ (\mathsf{fst}\ (pq_k))) \wedge k < |\rho'| \Rightarrow \\ &(\mathsf{last}\ \rho')\ (\mathsf{snd}\ (\mathsf{last}\ pq)) \geq p \vee p \geq (\mathsf{last}\ \rho')\ (\mathsf{snd}\ (\mathsf{last}\ pq))\end{aligned}$$

By applying the theorem from Section 3.3 to $\mathsf{sol\_seq}\ (\mathsf{front}\ \rho)\ pq$ (any prefix of a solution is a solution), the assumptions $\mathsf{mg\_sol\_seq}\ \rho'\ pq$ and $\mathsf{invertible\_on}\ (\rho'_k)\ (\mathsf{FV}\ (\mathsf{fst}\ (pq_k)))$ can be discharged, as long as $|pq| > 0$.

3.5 Restriction to Only $\leq$-Extensions

Non-termination of $\rightsquigarrow^{\downarrow +}$ means that there is an infinite sequence $(p_i, q_i)_{i \in \mathbb{N}} \subseteq \rightsquigarrow$ with an infinite solution. Kunčar observes [14, Lemma 5.16] that the following holds. We correct the argument in [7, Theorem 11].

> For a composable and monotone dependency relation $\rightsquigarrow$, if the sequence $(p_i, q_i)_{i \in \mathbb{N}} \subseteq \rightsquigarrow$ has an infinite solution, then the following holds. There exists an index k, such that for each $k' > k$ the sequence $(p_i, q_i)_{i < k'}$ has a most general solution whose extension with $(p_{k'}, q_{k'})$ is a $\leq$-extension.

It suffices to show this claim for the sequence $(p_i, q_i)_{k \leq i < k'}$ instead (by an argument involving Section 3.3). Using composability, by contradiction there exists for each k a smallest index k' such that for any most general solution $(\rho_i)_{i < k'}$ of $(p_i, q_i)_{i < k'}$ the extension step by $(p_{k'}, q_{k'})$ is a strict $\geq$-extension. We briefly illustrate why these infinitely many strict $\geq$-extension steps lead to a contradiction, by observing the change of each solution step at index 0 using Section 2.2.

Assume $(\rho'_i)_{i \leq k'}$ is the most general solution of the longer sequence $(p_i, q_i)_{i \leq k'}$. For a $\leq$-extension, type sizes and number of type variables do not change: $\mathsf{size}(\rho_0(p_0)) = \mathsf{size}(\rho'_0(p_0))$ and $|\,\mathsf{FV}(\rho_0(p_0))| = |\,\mathsf{FV}(\rho'_0(p_0))|$. For a strict $\geq$-extension at index k', by monotonicity we can transfer the reasoning about type size and type variables from index k' to index 0: the type size may increase $\mathsf{size}(\rho_0(p_0)) \leq \mathsf{size}(\rho'_0(p_0))$, and if instead holds $\mathsf{size}(\rho_0(p_0)) = \mathsf{size}(\rho'_0(p_0))$ then the number of free variables decreases $|\,\mathsf{FV}(\rho_0(p_0))| > |\,\mathsf{FV}(\rho'_0(p_0))|$. During each of the infinitely many strict $\geq$-extensions the number of type variables at p_0 after instantiation can only decrease to zero. The contradiction is, that also the size of the type substitution applied to p_0 may not strictly increase infinitely. After finitely many strict $\geq$-extensions the type size of the respective most general solution at index 0 will be larger than the type size of the original infinite solution, that witnessed non-termination.

3.6 Non-termination Implies Cyclicity

By the arguments in Section 3.5, any infinite sequence in $\rightsquigarrow^{\downarrow +}$ entails a corresponding sequence in $\rightsquigarrow\rightsquigarrow^{\downarrow +}$ that contains $\leq$-extensions steps only.

As $\rightsquigarrow$ is finite, in an infinite sequence $(p_i, q_i)_{i \in \mathbb{N}} \subseteq \rightsquigarrow$ after some index k, a repetition must exist, i. e. for every i with $i > k$, (p_i, q_i) occurs again in $(p_j, q_j)_{j > i}$. We apply the theorem from Section 3.5 to the sequence $(p_i, q_i)_{i \geq k} \subseteq \rightsquigarrow$ with solution $(\rho_i)_{i \geq k}$. Hence at

some index $k' > k$ we have a most general solution $(\rho'_i)_{i \geq k'}$ such that $\rho'_{k'}$ only renames the variables of $p_{k'}$ and only $\leq$-extension steps occur. We assume that $\rho'_{k'} = \mathsf{id}$, otherwise variables in the solution can be renamed (cf. Section 2.4.3). Let $m > k'$ be an index at which $(p_{k'}, q_{k'})$ occurs again, i. e. $(p_{k'}, q_{k'}) = (p_m, q_m)$, then $\rho'_{m-1}(q_{m-1}) \leq p_m = p_{k'}$ gives a cycle:

$$p_{k'} \rightsquigarrow q_{k'} = \rho'_{k'+1}(p_{k'+1}) \rightsquigarrow^{\downarrow} \ldots \rightsquigarrow^{\downarrow} \rho'_{m-1}(q_{m-1}) \leq p_m = p_{k'}$$

3.7 Comment on the Formalisation

From a proof-engineering perspective the formalisation of the arguments in our correction to the key lemma (cf. [7, Theorem 11] and [14, Lemma 5.16]) and also the implication "non-termination implies cyclicity" ([14, Lemma 5.17]) involves reasoning about infinite sequences and their infinite subsequences that satisfy certain properties, for which we used the Hilbert choice operator. One overall technical obstacle is the correct handling of type variables, for example some substitutions must be considered up to variable renaming and we need to verify a unification algorithm.

4 Algorithmically Checking for Cycles

In this section we describe the main idea and the correctness properties of a clocked breadth-first search that checks for composability and cycles in dependency relations. We invent a clocked algorithm of our own, because the restriction of *orthogonality* of dependency relations, that tremendously decreases the search space (as suggested by Kunčar [14]) does not apply to dependencies induced by theories of overloading definitions. We discuss the most important function, and conclude this section with an evaluation. We extract provably correct code, and check Isabelle/HOL theories for cycles in definitions.

4.1 Main Idea

As argued in Section 3.6, it suffices to search $\rightsquigarrow\rightsquigarrow^{\downarrow+}$ for cycles, in order to check non-termination of $\rightsquigarrow^{\downarrow+}$, given composability.

The basis for the cyclicity check is the following corollary. For a finite monotone dependency relation $\rightsquigarrow$, if at each depth composability and acyclicity hold, the type-substitutive closure of the dependency relation is terminating.

$$\vdash \mathsf{wellformed} \rightsquigarrow \wedge\ \mathsf{monotone} \rightsquigarrow \wedge\ \mathsf{finite}\ (\rightsquigarrow) \wedge (\forall n.\ \mathsf{composable\_len} \rightsquigarrow n) \wedge (\forall n.\ \neg\mathsf{cyclic\_len} \rightsquigarrow n) \Rightarrow \mathsf{terminating} \rightsquigarrow^{\downarrow+}$$

We recall from the definitions of $\mathsf{composable\_len} \rightsquigarrow n$ and $\mathsf{cyclic\_len} \rightsquigarrow n$, that these consider the paths $\rightsquigarrow (\rightsquigarrow^{\downarrow})^n$, which consist of only $\leq$-extensions, cf. Section 3.4.

The algorithm checks $\mathsf{composable\_len} \rightsquigarrow n$ and $\neg\mathsf{cyclic\_len} \rightsquigarrow n$ in a breadth-first manner for all $n \geq 1$ up to a given depth limit. (The case $n = 0$ is trivial.) The depth limit is required, because the search might not terminate if the dependencies $\rightsquigarrow\rightsquigarrow^{\downarrow+}$ are infinite. We discuss the depth limit for some practical examples in Section 4.5.

4.2 Central Components of the Algorithm

In this section we describe the algorithm dep_steps and discuss its core function composable_one in further detail.

To check a dependency relation $\rightsquigarrow$, the cyclicity checker algorithm dep_steps is called as dep_steps $\rightsquigarrow$ k $\rightsquigarrow$ with a maximal depth k. The function call dep_steps $\rightsquigarrow$ k $\mathcal{R}$ recursively extends each path that is stored in $\mathcal{R}$ by a dependency step $\rightsquigarrow$, and terminates either when $\mathcal{R}$ is empty, thus no extension is possible and all earlier steps were composable and acyclic. Or otherwise the recursion terminates if the depth counter k decreases to 0, and thus there are paths longer than k that the algorithm did not check for composability nor for cyclicity.

A call to dep_steps $\rightsquigarrow$ k $\rightsquigarrow$ results in one of the following outcomes, whose correctness we discuss in Section 4.3.

- Maybe_cyclic if the recursion depth k was too small, cf. Section 4.3.2.
- Acyclic _ if the relation is acyclic, cf. Section 4.3.3.
- Cyclic_step $(p,\_,p')$ if a cycle $p \rightsquigarrow\rightsquigarrow^{\downarrow+} p'$ with $p \geq p'$ exists, cf. Section 4.3.4,
- Non_comp_step (p,q,pq') if a non-composable path exists, i. e. $p \rightsquigarrow\rightsquigarrow^{\downarrow+} q$ with $pq' \in \rightsquigarrow$ and $\neg(q \geq \mathsf{fst}\ pq')$, $\neg(\mathsf{fst}\ pq' \geq q)$, $\neg(q \# \mathsf{fst}\ pq')$, cf. Section 4.3.4.

The function dep_steps $\rightsquigarrow$ k $\mathcal{R}$ folds another function dep_step $\rightsquigarrow$ at most k times over $\mathcal{R}$. For a relation $\mathcal{R}$ corresponding to $\rightsquigarrow$ $(\rightsquigarrow^{\downarrow})^n$ (for some breadth $n \geq 0$) the call to dep_step $\rightsquigarrow$ $\mathcal{R}$ computes all $\leq$-extensions for the current breadth n and checks the resulting relation $\rightsquigarrow$ $(\rightsquigarrow^{\downarrow})^{n+1}$ for cyclicity. Each of the possible extension steps of each of the paths $x\mathcal{R}y$ (i. e. $x \rightsquigarrow (\rightsquigarrow^{\downarrow})^n y$) by $p \rightsquigarrow q$ is computed according to Section 3.4 by composable_one y p, which may fail whenever $\mathcal{R}$ is not composable.

```
composable_one y p ≝
  case unify y p of
    None ⇒ Ignore
  | Some (s_y,s_p) ⇒
    let sp_inv = invertible_on s_p (FV p) ; sy_inv = invertible_on s_y (FV y) in
      if sp_inv ∧ ¬sy_inv then Ignore
      else if ¬sp_inv ∧ ¬sy_inv then Uncomposable
      else if ¬sp_inv ∧ sy_inv then Continue s_p
      else Continue []
```

Figure 1 Definition of the function composable_one, that is called by dep_steps.

The function composable_one y p, as defined in Figure 1, attempts to unify the two symbols y and p, and calculates if the resulting type substitutions are invertible. The possible return values are either Ignore, Uncomposable or Continue ρ for some type substitution ρ.

An Ignore return value signifies either orthogonality of y and p or a strict $\geq$-extension, i. e. $y \geq p$ and $\neg(y \leq p)$. If the unifying type substitutions are both not invertible, the relation is uncomposable. A return value Continue ρ means that $y \approx \rho\, p$, and the path can be continued as $x \rightsquigarrow (\rightsquigarrow^{\downarrow})^{n+1} (\rho\, q)$. Following the $\leq$-extension steps each calculated path is checked for cyclicity, i. e. if $x \geq (\rho\, q)$ holds.

4.3 Correctness

The major result about our algorithm dep_steps is its correctness. In Section 4.1 we motivated that for a monotone dependency relation that is acyclic and composable at all lengths, the type-substitutive transitive closure of the dependency relation is terminating. This theorem shows that we check composability and acyclicity using our algorithm dep_steps:

$$\vdash \mathsf{wellformed}\ {\rightsquigarrow} \wedge \mathsf{monotone}\ (\rightsquigarrow) \wedge \mathsf{finite}\ (\rightsquigarrow) \wedge \mathsf{dep\_steps}\ {\rightsquigarrow}\ (\mathsf{SUC}\ k)\ {\rightsquigarrow} = \mathsf{Acyclic}\ k' \Rightarrow \mathsf{terminating}\ {\rightsquigarrow}^{\downarrow +}$$

We outline some proof ideas and discuss further soundness and correctness properties.

The correctness proof is unsurprising but technically involved.

4.3.1 The Recursion Invariant of dep_steps

We establish a simple invariant $\mathsf{dep\_steps\_inv}\ {\rightsquigarrow}\ i\ \mathcal{R}\ j\ \mathcal{R}'$ that captures that $\mathcal{R}$ reduces to $\mathcal{R}'$ in $(i - j)$ applications of $\mathsf{dep\_step}\ {\rightsquigarrow}$, that is, $(i - j)$ composable and non-cyclic steps in the dependency relation $\rightsquigarrow$. We further on regard the case of $\mathcal{R} = {\rightsquigarrow}$.

The invariant $\mathsf{dep\_steps\_inv}\ {\rightsquigarrow}\ i\ {\rightsquigarrow}\ j\ \mathcal{R}'$ entails that $\mathcal{R}' = {\rightsquigarrow}\ (\rightsquigarrow^{\downarrow})^{i-j}$ modulo renaming.

$$\vdash \mathsf{wellformed}\ {\rightsquigarrow} \wedge \mathsf{monotone}\ (\rightsquigarrow) \wedge \mathsf{dep\_steps\_inv}\ {\rightsquigarrow}\ i\ {\rightsquigarrow}\ j\ \mathcal{R}' \Rightarrow \forall x.\ \mathsf{wellformed}\ [x] \Rightarrow ((\exists y.\ (\mathsf{fst}\ x, y) \in \mathcal{R}' \wedge y \approx \mathsf{snd}\ x) \iff \mathsf{has\_path\_to}\ (\rightsquigarrow)\ (\mathsf{SUC}\ (i - j))\ (\mathsf{fst}\ x)\ (\mathsf{snd}\ x))$$

Modulo renaming, $\mathcal{R}'$ contains exactly the paths in $\rightsquigarrow$ of length $i - j + 1$.

For a non-trivial, monotone dependency relation we characterise the invariant exactly.

$$\vdash \mathsf{wellformed}\ {\rightsquigarrow} \wedge \mathsf{monotone}\ (\rightsquigarrow) \wedge \neg\mathsf{null}\ {\rightsquigarrow} \wedge j \leq i \Rightarrow ((\exists \mathcal{R}'.\ \mathsf{dep\_steps\_inv}\ {\rightsquigarrow}\ i\ {\rightsquigarrow}\ j\ \mathcal{R}') \iff (1 < i - j \Rightarrow \exists x\ y.\ \mathsf{has\_path\_to}\ (\rightsquigarrow)\ (i - j)\ x\ y) \wedge \forall k'.\ 0 < k' \wedge k' \leq i - j \Rightarrow \mathsf{composable\_len}\ (\rightsquigarrow)\ k' \wedge \neg\mathsf{cyclic\_len}\ (\rightsquigarrow)\ (\mathsf{SUC}\ k'))$$

The first conjunct within the conclusion $\exists x\ y.\ \mathsf{has\_path\_to}\ {\rightsquigarrow}\ (i - j)\ x\ y$ expresses non-emptiness of the previous search depth. If the search depth $i - j$ is strictly larger than one, then there exists a path in $\rightsquigarrow\ (\rightsquigarrow^{\downarrow})^{i-j-1}$. The second conjunct says that for each depth k' such that $0 < k' \leq i - j$, all paths in the relation $\rightsquigarrow\ (\rightsquigarrow^{\downarrow})^{k'-1}$ are composable and all paths in $\rightsquigarrow\ (\rightsquigarrow^{\downarrow})^{k'}$ are acyclic.

The lengths of the paths for composability and acyclicity differ by one, as within $\mathsf{dep\_step}$ a composable relation is first extended, and then the resulting (one step longer) paths are checked for cyclicity.

4.3.2 Example: Deriving Correctness for Non-exhaustive Search

We establish when $\mathsf{dep\_steps}\ {\rightsquigarrow}\ k\ {\rightsquigarrow}$ outputs Maybe_cyclic: it can happen when $\rightsquigarrow\ (\rightsquigarrow^{\downarrow})^{k}$ is composable and acyclic, and contains a path that can be prolonged. In other words k-many iterative calls to $\mathsf{dep\_step}\ {\rightsquigarrow}$ yield a non-empty result, which in-turn shall mean that the supplied search depth was too small to cover all paths, and the calculated relation is non-empty and could contain the initial segment of a cycle. We establish this claim in terms of the invariant that we introduced in Section 4.3.1.

$$\vdash \mathsf{wellformed}\ {\rightsquigarrow} \Rightarrow (\mathsf{dep\_steps}\ {\rightsquigarrow}\ k\ {\rightsquigarrow} = \mathsf{Maybe\_cyclic} \iff \exists \mathcal{R}'.\ \mathsf{dep\_steps\_inv}\ {\rightsquigarrow}\ k\ {\rightsquigarrow}\ 0\ \mathcal{R}' \wedge \neg\mathsf{null}\ \mathcal{R}')$$

Further, we establish that composability and acyclicity hold for the respective intermediate search steps up to the recursion level k, and secondly, that the relation $\mathcal{R}' = {\rightsquigarrow}\ (\rightsquigarrow^{\downarrow})^{k}$ is non-empty. The latter is witnessed by a path of length $k + 1$.

$$\begin{aligned}&\vdash \mathsf{wellformed}\ {\rightsquigarrow} \wedge \mathsf{monotone}\ ({\rightsquigarrow}) \Rightarrow\\ &\quad (\mathsf{dep\_steps}\ {\rightsquigarrow}\ k\ {\rightsquigarrow} = \mathsf{Maybe\_cyclic} \iff\\ &\qquad (\forall\, l.\ 0 < l \wedge l \leq k \Rightarrow \mathsf{composable\_len}\ ({\rightsquigarrow})\ l \wedge \neg\mathsf{cyclic\_len}\ ({\rightsquigarrow})\ (\mathsf{SUC}\ l)) \wedge\\ &\qquad \exists\, x\ y.\ \mathsf{has\_path\_to}\ ({\rightsquigarrow})\ (\mathsf{SUC}\ k)\ x\ y)\end{aligned}$$

4.3.3 Correctness for Acyclicity

For a sufficiently large k the function dep_steps detects that the non-trivial, monotone relation $\rightsquigarrow$ is acyclic, returning Acyclic k' for some integer $k' \leq k$. The value k' is such that all paths in the calculated relation ${\rightsquigarrow}{\rightsquigarrow}^{\downarrow+}$ are at most of length $k - k'$ and no path of length $k - k' + 1$ (or larger) exists. All paths of smaller length are composable and acyclic.

$$\begin{aligned}&\vdash \mathsf{wellformed}\ {\rightsquigarrow} \wedge \mathsf{monotone}\ ({\rightsquigarrow}) \wedge \neg\mathsf{null}\ {\rightsquigarrow} \wedge 0 < k \Rightarrow\\ &\quad (\mathsf{dep\_steps}\ {\rightsquigarrow}\ k\ {\rightsquigarrow} = \mathsf{Acyclic}\ k' \iff\\ &\qquad k' \leq k \wedge (\forall\, x\ y.\ \neg\mathsf{has\_path\_to}\ ({\rightsquigarrow})\ (\mathsf{SUC}\ (k - k'))\ x\ y) \wedge\\ &\qquad (1 < k - k' \Rightarrow \exists\, x\ y.\ \mathsf{has\_path\_to}\ ({\rightsquigarrow})\ (k - k')\ x\ y) \wedge\\ &\qquad \forall\, l.\ 0 < l \wedge l \leq k - k' \Rightarrow \mathsf{composable\_len}\ ({\rightsquigarrow})\ l \wedge \neg\mathsf{cyclic\_len}\ ({\rightsquigarrow})\ (\mathsf{SUC}\ l))\end{aligned}$$

Acyclicity holds for paths of all lengths, including length one (which is not included in the equivalence). Such are cycles in $\rightsquigarrow$, and these entail cycles of length 2, and thus $\neg$cyclic_len $\rightsquigarrow$ 1 follows from $\neg$cyclic_len $\rightsquigarrow$ 2. Overall holds composable_dep $\rightsquigarrow$ and $\neg$cyclic_dep $\rightsquigarrow$.

4.3.4 Soundness for *Non-composable* and *Cyclic*

By the above two correctness results (in Sections 4.3.2 and 4.3.3) the algorithm dep_steps is sound and complete. For illustration, we state the soundness for when the algorithm witnesses a non-composable path or detects a cycle. For better readability we omit acyclicity and composability of previous search lengths from the conclusion.

$$\begin{aligned}&\vdash \mathsf{wellformed}\ {\rightsquigarrow} \wedge \mathsf{monotone}\ ({\rightsquigarrow}) \wedge\\ &\quad \mathsf{dep\_steps}\ {\rightsquigarrow}\ (\mathsf{SUC}\ k)\ {\rightsquigarrow} =\\ &\qquad \mathsf{Non\_comp\_step}\ (p,q,pq') \Rightarrow\\ &\qquad \exists\, n.\ n \leq k \wedge\\ &\qquad\quad \mathsf{has\_path\_to}\ ({\rightsquigarrow})\ (\mathsf{SUC}\ n)\ p\ q \wedge\\ &\qquad\quad pq' \in {\rightsquigarrow} \wedge \neg(q \geq \mathsf{fst}\ pq') \wedge\\ &\qquad\quad \neg(\mathsf{fst}\ pq' \geq q) \wedge \neg(q\ \#\ \mathsf{fst}\ pq')\end{aligned}$$

$$\begin{aligned}&\vdash \mathsf{wellformed}\ {\rightsquigarrow} \wedge \mathsf{monotone}\ ({\rightsquigarrow}) \wedge\\ &\quad \mathsf{dep\_steps}\ {\rightsquigarrow}\ (\mathsf{SUC}\ k)\ {\rightsquigarrow} =\\ &\qquad \mathsf{Cyclic\_step}\ (p,q,p') \Rightarrow\\ &\qquad \exists\, n.\ n \leq k \wedge\\ &\qquad\quad \mathsf{has\_path\_to}\ ({\rightsquigarrow})\ (\mathsf{SUC}\ (\mathsf{SUC}\ n))\ p\ p' \wedge\\ &\qquad\quad p \geq p'\end{aligned}$$

When an uncomposable step is detected, then there exists a path $p \rightsquigarrow (\rightsquigarrow^{\downarrow})^n\ q$ that is not composable (cf. Section 2.5). As stated, when a cyclic step is detected, then there is a path in $\rightsquigarrow (\rightsquigarrow^{\downarrow})^{n+1}$ that ends in an instance of the starting symbol, and by definition cyclic_dep $\rightsquigarrow$ holds (cf. Section 2.4.4).

4.4 A Verified Theorem Prover Kernel

In earlier work [1] we implemented parts of a theorem prover that supports overloading of constant definitions, and is partially based on a verified implementation of HOL Light (nicknamed Candle) [13]. A verified cyclicity checker is the missing puzzle piece of this verified theorem prover. Whenever an overloaded constant is added to a proof development the resulting theory needs to have a terminating dependency relation.

Theory	#Dependencies	Output	Runtime	Longest path
HOL	165	Acyclic	0.01s	7
Orderings	764	Acyclic	0.4s	13
Set	2657	Acyclic	13s	14
Fun	2773	Acyclic	13s	14
Transitive_Closure*	2195	Acyclic	8s	20
Transitive_Closure	7159	Acyclic	14h	35
Main*	12913	Acyclic	12h	37
Main	45738	-	-	-

Figure 2 Results of checking exported dependencies of Isabelle theories. The asterisk * denotes that the dependencies only include dependencies of constants on constants. The runtimes are from single runs.

A theory of definitions *ctxt* has the relational dependencies dependency *ctxt* that are computed by dependency_compute *ctxt*. The following corollary (of the results from Section 3 and Section 4.3.3) allows to discharge termination of dependency *ctxt* by a call to the dep_steps algorithm on the dependencies dependency_compute *ctxt*.

$$\begin{aligned}&\vdash \mathsf{let}\ \rightsquigarrow\ =\ \mathsf{dependency\_compute}\ ctxt\ \mathsf{in}\\&\quad \mathsf{dep\_steps}\ \rightsquigarrow\ (\mathsf{SUC}\ k)\ \rightsquigarrow = \mathsf{Acyclic}\ k' \wedge \mathsf{good\_constspec\_names}\ ctxt \Rightarrow\\&\quad\ \mathsf{terminating\ dependency}\ ctxt^{\downarrow +}\end{aligned}$$

The premise good_constspec_names *ctxt* states that all type variables on the right-hand side of a constant definition must occur on the left-hand side. This ensures that the dependency relation is monotonic. All HOL kernel implementations that we are aware of enforce this restriction, since without it, HOL is inconsistent (see Section 2.4.1).

To obtain a verified kernel, we integrate our cyclicity check into a shallowly embedded monadic HOL kernel derived from [13], and extract a correct-by-construction CakeML implementation using existing tools for proof-producing synthesis [11]. By the previous theorem, we can replace the termination assumption with a call to the (monadic) cyclicity checker and prove kernel soundness, i. e. a successful check entails a valid update of a theory.

To our knowledge, this yields the first verified theorem prover kernel that both supports overloading of constant definitions and has mechanised semantics [1] with a formal proof that any theory is consistent. The proof of consistency for theories relies on the consistency of Zermelo-Fraenkel set theory. From our joint work with Weber follow (formally verified) model-theoretic conservativity guarantees [8].

4.5 Checking Cyclicity of Isabelle/HOL Theories

In addition to verifying the theory and deriving a correct algorithm, we extracted the dependencies of theories from Isabelle/HOL and checked if their dependency graphs are acyclic and composable. We focus on the theory Main, that extends HOL with libraries for e. g. orderings, lattices, transitive closure, sets and natural numbers [16].

We composed an (unverified) dependency parser written in CakeML with the (verified) CakeML implementation of our cyclicity checker, and extracted an executable binary using the CakeML compiler. The translator ensures that the cyclicity checking function of the binary has the same correctness properties as the monadic variant, which in turn is equivalent to the HOL4 implementation from Section 4.2.

The results in Figure 2 show for each Isabelle/HOL theory the number of extracted dependencies, the result output of the checker, the runtime in seconds or hours on an Intel Core i7 processor, and the length of the longest checked path. All of these results state that the extracted dependencies contain no cycles and are composable. The reported maximal length of covered paths of 37 shows that for these realistic scenarios the maximum depth limit can be chosen small. Implementing a cyclicity checker that is optimised for performance was not the objective of this work, which shows in the runtimes. Checking the 45,738 dependencies of the complete Isabelle/HOL main library is not feasible, but we establish that the subset of dependencies from constants to constants is acyclic. This approach is unsound, but gives an idea of the algorithm's performance.

In the checked theories overloading is mainly occurring due to type classes, e. g. the `size_class` exhibits a constant `Nat.size_class.size` of type $\alpha \to \mathsf{nat}$, and the list type is a class instance that defines a `size` constant at the type $\alpha\ \mathsf{list} \to \mathsf{nat}$.

The runtimes motivate that a cyclicity checker should be checking dependencies incrementally, which we discuss further in Section 6, because incremental checking corresponds to the incremental nature of theory extension.

5 Related Work

Like other theorem provers that do not support overloading, the verified implementation of HOL Light into the CakeML framework [13], nicknamed Candle, achieves acyclic definitions by a simple syntactic check: Only already defined constants and types are allowed to occur in the definition of a new symbol.

In the Coq theorem prover a termination check corresponds to finding a type hierarchy acyclic. When type-checking a term, hierarchy (in)equality constraints are collected, whose conjunction needs to be satisfiable. As Sozeau and Tabareau argue [22], checking these conjunctions is decidable. Earlier a proof of contradiction seemed to be originating from a bug in this cyclicity checker [6]. The current Coq implementation [12] relies on an incremental cyclicity checking algorithm by Bender et al. [4], that combines forward- and backward-search and uses a non-decreasing integer level invariant for chains in the graph. Guéneau et al. [9] verified a similar algorithm in addition to some of the complexity properties. We discuss incremental extension to our algorithm in Section 6. In contrast to Sozeau and Tabareau, checking termination of interesting dependency relations in our context is not decidable. To add another difference, their rewriting system allows unfolding of a constant by its definition, whereas our cyclicity checker can also check dependencies from theories with more expressive definitions [2], e. g. implicit definitions.

The current algorithm that checks for cycles in Isabelle theories[4] is authored by Wenzel. It is unclear how the implementation relates to Kunčar's work [14], and our cyclicity checker. A later proof checker for Isabelle/HOL by Nipkow and Roßkopf [17] treats definitions as axioms, i. e. does not check for cycles.

With the intent to obtain a terminating and efficient cycle detection algorithm, Kunčar suggests in [14] that dependency relations should satisfy *orthogonality*. The orthogonality criterion implies that paths do not diverge, hence any two paths from a symbol pass through the same symbols if the paths are of same length [14, Theorem 6.2]. That means that composability only needs to hold for paths that cannot be extend any further, so called *final* paths. Orthogonality is no suiting criterion for dependencies of definitions (cf. [15, 1]): any definition of a symbol may depend on more than one other symbol [7].

[4] `https://isabelle.sketis.net/repos/isabelle/file/Isabelle2021/src/Pure/defs.ML`

6 Future Work

With the suggested algorithm we already implement the interesting and correctness critical optimisation to avoid inversion of bijective renamings. For future work we identify three main areas of improvements.

First, we should investigate further restrictions of dependency relations that avoid checking composability of every path and second, the dependency relation that is generated from theories should be minimised. For example, dependencies introduced by some built-in symbols can be omitted from the graph.

Third, theory extension is incremental, and such should the check of cyclicity (and composability) be. Whenever a theory is extended by a definition, only a few dependencies are added by the new definition, which implies that big parts of the dependency graph remain unchanged. Together with Weber [8] we have identified those parts of the dependency graph that change by the introduction of a new definition. An incremental cyclicity check clearly profits from such an analysis.

As an example, assume a theory with terminating dependencies that contains the constant $\mathsf{length}_{\alpha\ \mathsf{list}\to\mathsf{nat}}$ (that returns the length of a list) with the dependency $\mathsf{length}_{\alpha\ \mathsf{list}} \rightsquigarrow \alpha\ \mathsf{list}$. When this theory is extended by a new polymorphic constant $\mathsf{size}_{\alpha\to\mathsf{nat}}$ and the definition

$$\mathsf{size}_{\alpha\ \mathsf{list}} \equiv \mathsf{length}_{\alpha\ \mathsf{list}}$$

two new dependencies $\mathsf{size}_{\alpha\ \mathsf{list}} \rightsquigarrow \mathsf{length}_{\alpha\ \mathsf{list}}$ and $\mathsf{size}_{\alpha\ \mathsf{list}} \rightsquigarrow \alpha\ \mathsf{list}$ are introduced. A cyclicity check may stop after covering the path from $\mathsf{size} \rightsquigarrow \mathsf{length}$, because it is already known that paths from length are composable and acyclic. Thus any path starting from $\mathsf{size}_{\alpha\ \mathsf{list}} \rightsquigarrow \alpha\ \mathsf{list}$ need not to be covered at all.

7 Conclusion

In this paper we have presented the theory and implementation of a formally verified cyclicity checker and its use in a verified theorem prover kernel that supports (ad-hoc) overloaded constant definitions. We demonstrated a verified binary cyclicity checker on theories of Isabelle/HOL and established that the definitions are acyclic. The verified binary was synthesised from our verified version through the CakeML infrastructure [23].

This work closes a gap in the foundation of Isabelle/HOL, in two ways. First, we establish the formalised theory for checking dependencies. Second, we discharge an assumption from earlier work [1], which strengthens the consistency by mechanised semantics.

Our verified kernel could be used as a verified proof checker for simple Isabelle/HOL theories. After accounting for differences Isabelle/HOL's and our kernel's logic, like axiomatic type classes, and implementing and verifying a performant cyclicity checker, our theorem prover kernel could proof-check Isabelle/HOL theories, and their dependencies.

References

1 Johannes Åman Pohjola and Arve Gengelbach. A Mechanised Semantics for HOL with Ad-hoc Overloading. In Elvira Albert and Laura Kovács, editors, *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020*, volume 73 of *EPiC Series in Computing*, pages 498–515. EasyChair, 2020. doi:10.29007/413d.

2 Rob Arthan. HOL constant definition done right. 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 531–536. Springer, 2014. doi:10.1007/978-3-319-08970-6_34.

3 Franz Baader, Wayne Snyder, Paliath Narendran, Manfred Schmidt-Schauß, and Klaus U. Schulz. Unification Theory. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 445–532. Elsevier and MIT Press, 2001. doi:10.1016/b978-044450813-3/50010-2.

4 Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan. A New Approach to Incremental Cycle Detection and Related Problems. *ACM Trans. Algorithms*, 12(2):14:1–14:22, 2016. doi:10.1145/2756553.

5 Jared Curran Davis. *A self-verifying theorem prover*. The University of Texas at Austin, 2009.

6 Maxime Dénès. [Coq-Club] Propositional extensionality is inconsistent in Coq, December 2013. Coq Club Mailinglist. URL: https://sympa.inria.fr/sympa/arc/coq-club/2013-12/msg00119.html.

7 Arve Gengelbach and Johannes Åman Pohjola. Towards Correctly Checking for Cycles in Overloaded Definitions. Technical Report 2021-001, Department of Information Technology, Uppsala University, March 2021. URL: http://www.it.uu.se/research/publications/reports/2021-001/.

8 Arve Gengelbach, Johannes Åman Pohjola, and Tjark Weber. Mechanisation of Model-theoretic Conservative Extension for HOL with Ad-hoc Overloading. In Claudio Sacerdoti Coen and Alwen Tiu, editors, *Proceedings Fifteenth Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP 2020, Paris, France, 29th June 2020*, volume 332 of *EPTCS*, pages 1–17, 2020. doi:10.4204/EPTCS.332.1.

9 Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. Formal Proof and Analysis of an Incremental Cycle Detection Algorithm. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*, pages 18:1–18:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ITP.2019.18.

10 John Harrison. Towards self-verification of HOL light. In Ulrich Furbach and Natarajan Shankar, editors, *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 177–191. Springer, 2006. doi:10.1007/11814771_17.

11 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 Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*, volume 10900 of *Lecture Notes in Computer Science*, pages 646–662. Springer, 2018. doi:10.1007/978-3-319-94205-6_42.

12 Jacques-Henri Jourdan. Coq pull request #90, July 2015. URL: https://github.com/coq/coq/pull/89.

13 Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. Self-Formalisation of Higher-Order Logic - Semantics, Soundness, and a Verified Implementation. *J. Autom. Reason.*, 56(3):221–259, 2016. doi:10.1007/s10817-015-9357-x.

14 Ondřej Kunčar. Correctness of Isabelle's Cyclicity Checker: Implementability of Overloading in Proof Assistants. In *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*, pages 85–94. ACM, 2015. doi:10.1145/2676724.2693175.

15 Ondřej Kunčar and Andrei Popescu. A Consistent Foundation for Isabelle/HOL. *J. Autom. Reason.*, 62(4):531–555, 2019. doi:10.1007/s10817-018-9454-8.

16 Tobias Nipkow. What's in Main, December 2021. URL: https://isabelle.in.tum.de/dist/Isabelle2021-1/doc/main.pdf.

17 Tobias Nipkow and Simon Roßkopf. Isabelle's Metalogic: Formalization and Proof Checker. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 93–110. Springer, 2021. doi:10.1007/978-3-030-79876-5_6.

18 Steven Obua. Checking Conservativity of Overloaded Definitions in Higher-Order Logic. In *Term Rewriting and Applications, 17th International Conference, RTA 2006, Seattle, WA, USA, August 12-14, 2006, Proceedings*, pages 212–226. Springer, 2006. doi:10.1007/11805618_16.

19 Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008. doi:10.1007/978-3-540-71067-7_6.

20 Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *Proc. ACM Program. Lang.*, 4(POPL):8:1–8:28, 2020. doi:10.1145/3371076.

21 Matthieu Sozeau and Nicolas Oury. First-class type classes. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2008. doi:10.1007/978-3-540-71067-7_23.

22 Matthieu Sozeau and Nicolas Tabareau. Universe polymorphism in coq. 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 499–514. Springer, 2014. doi:10.1007/978-3-319-08970-6_32.

23 Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *J. Funct. Program.*, 29:e2, 2019. doi:10.1017/S0956796818000229.

24 Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 60–76. ACM Press, 1989. doi:10.1145/75277.75283.

25 Makarius Wenzel. The Isabelle/Isar Reference Manual, December 2021. URL: https://isabelle.in.tum.de/dist/Isabelle2021-1/doc/isar-ref.pdf.

26 Markus Wenzel. Type classes and overloading in higher-order logic. In Elsa L. Gunter and Amy P. Felty, editors, *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97, Murray Hill, NJ, USA, August 19-22, 1997, Proceedings*, volume 1275 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 1997. doi:10.1007/BFb0028402.

The Isabelle ENIGMA

Zarathustra A. Goertzel
Czech Technical University in Prague, Czech Republic

Jan Jakubův
Czech Technical University in Prague, Czech Republic
Universität Innsbruck, Austria

Cezary Kaliszyk
Universität Innsbruck, Austria

Miroslav Olšák
Institut des Hautes Études Scientifiques, Bures-sur-Yvette, France

Jelle Piepenbrock
Czech Technical University in Prague, Czech Republic
Radboud University, Nijmegen, The Netherlands

Josef Urban
Czech Technical University in Prague, Czech Republic

Abstract

We significantly improve the performance of the E automated theorem prover on the Isabelle Sledgehammer problems by combining learning and theorem proving in several ways. In particular, we develop targeted versions of the ENIGMA guidance for the Isabelle problems, targeted versions of neural premise selection, and targeted strategies for E. The methods are trained in several iterations over hundreds of thousands untyped and typed first-order problems extracted from Isabelle. Our final best single-strategy ENIGMA and premise selection system improves the best previous version of E by 25.3% in 15 seconds, outperforming also all other previous ATP and SMT systems.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases E Prover, ENIGMA, Premise Selection, Isabelle/Sledgehammer

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.16

Supplementary Material *Dataset*: `https://github.com/ai4reason/isa_enigma_paper`

Funding This work was partially supported by the European Regional Development Fund under the Czech project AI&Reasoning no. CZ.02.1.01/0.0/0.0/15_003/0000466 (ZG, JP, JU), the ERC Starting Grant *SMART* no. 714034 (JJ, CK), Amazon Research Awards (JP, JU) and by the Czech MEYS under the ERC CZ project *POSTMAN* no. LL1902 (JJ, JP).

1 Introduction

Formal verification in interactive theorem provers (ITPs) increasingly benefits from general proof automation in the form of *hammers* [7] and guided tactical provers [4, 13, 36]. In particular, the Sledgehammer system [8] for Isabelle is today perhaps the most widely used strong general proof automation system in ITP. In the recent years, machine learning and related AI methods for proof automation have also been significantly developed [48]. Such methods are relevant for hammers in at least three ways: (i) learning-based *premise selection* [2, 3, 12, 37, 39, 40] usually improves the heuristic filters used by the hammers, (ii) learning-based *internal guidance* of the automated theorem provers (ATPs) used for the heavy lifting in the hammers usually improves on heuristic guidance of ATPs [16, 22, 24, 26, 27, 29, 41, 49], and (iii) targeted theorem proving strategies developed by automated strategy invention systems often improve on manually designed ATP strategies [20, 23, 42, 47].

Most recent versions of such AI/TP methods have been developed mainly on a fixed Mizar/MPTP corpus [28], to allow easy comparisons with previously developed methods. In particular, there the strongest 3-phase single-strategy version of the ENIGMA system (based on E [43, 44]) proves 56.35% of the holdout (test) toplevel theorems in 30s when using human-selected premises [16]. In higher time limits and by combining human and learning-based premise selection, ENIGMA and Vampire [32] today prove 75% of the toplevel Mizar theorems.[1] These are good reasons for transferring the methods to other ITP hammers.

A direct motivation for developing such AI/TP methods for Isabelle was a recent request from the Sledgehammer developers for an optimized version of ENIGMA for their GRUNGE-style [9] evaluation of multiple ATP systems and formats [11]. While it was not possible to do the work described in this paper on a two-week's notice, it prompted us to start exporting and analyzing the Isabelle datasets and developing suitable methods and systems for them.

1.1 Contributions

We significantly improve the performance of the E automated theorem prover on the Isabelle Sledgehammer problems by combining learning and theorem proving in several ways. First, in Section 2 we extract two large datasets of untyped first-order (FOF) and many-sorted first-order (TFF, TF0) Isabelle Sledgehammer problems, using the Isabelle tool Mirabelle. This results in almost 300000 aligned problems in each of the exports, spanning in total 1902 Isabelle theory files and covering a large number of topics in mathematics and formal verification. To our knowledge, these are so far the largest corpora of Isabelle Sledgehammer problems available today for training and evaluation of AI/TP systems. Section 2.1 analyzes the corpora, showing that they significantly differ from other large AI/TP datasets such as the Mizar/MPTP toplevel theorems [28] and the HOL4/GRUNGE toplevel theorems [9].

In Section 3, we find optimized E strategies and parameters for the corpora, which already improve on standard E on the problems. They are suitable also for combinations with the ENIGMA guidance, which is introduced in Section 4. We also describe there several extensions to ENIGMA that were developed to handle the Isabelle untyped and typed problems. Section 5 discusses the neural premise selection that we use and its extensions for the typed Isabelle setting. Section 6 evaluates the methods in several loops interleaving proving and learning from the proofs. Our ultimate performance results are: (i) improving in 15s the original E auto-schedule with the MePo filter by 25.3%, when using a single ENIGMA strategy with the best neural predictor, (ii) considerably improving over all other ATPs and SMTs by a single ENIGMA strategy combined with the best neural predictor, (iii) improving the performance of all other systems by using the neural predictor, and (iv) outperforming with ENIGMA all other ATPs and SMTs even when they are combined with our predictor.

2 Isabelle Problems

To train and evaluate the Isabelle ENIGMA, we need a dataset of Sledgehammer problems, which correspond to the proof obligations that users encounter when using Isabelle as an interactive prover. We decided to focus on all proof-intermediate goals visible to the users. This task has been tried as early as in the first versions of the MPTP system [46]. In Isabelle, it has been known as the "Judgement Day" evaluation, based on the paper with that title [8]. We have used the Isabelle/Mirabelle infrastructure to export all the problems encountered

[1] `https://github.com/ai4reason/ATP_Proofs/blob/master/75percent_announce.md`

when building 179 Isabelle sessions. These sessions originate from 75 sessions distributed with Isabelle 2021-1, 80 selected sessions from the AFP [6], as well as 24 sessions distributed as part of IsaFoR [45]. All the sessions include in total 1902 Isabelle theory files. The sessions with most problems can be categorized as Analysis, Algebra, Java Semantics, Category Theory, Protocols, Term Rewriting, and Probability Theory with the largest 26 sessions listed in Table 1.

Table 1 The largest included sessions and their respective problem numbers.

HOL-Nonstandard-Analysis	1699	Groebner-Macaulay	4227
Category2	1776	HOL-ODE-Numerics	4422
Poincare-Bendixson	1983	HOL-MicroJava	5183
HOL-Number-Theory	2071	HOL-Auth	5304
MonoidalCategory	2238	HOL-Complex-Analysis	5489
HOL-Cardinals	2268	Groebner-Bases	5710
Core-DOM	2280	HOL-Computational-Algebra	6280
HOL-IMP	2324	Jordan-Normal-Form	6786
HOL-Data-Structures	2353	Category3	6818
Dirichlet-Series	2435	HOL-Probability	6954
Slicing	2517	HOL-Decision-Procs	7103
HOLCF	2524	CR	7341
Formal-SSA	2899	HOL-Bali	7804
HOL-UNITY	2938	HOL	7818
HOL-Homology	3022	Goedel-HFSet-Semanticless	8697
HOL-ex	3047	HOL-Algebra	9674
CTRS	3328	HRB-Slicing	10052
HOL-Hoare-Parallel	3733	Jinja	11520
Signature-Groebner	3762	HOL-Library	15627
Valuation	3786	Bicategory	16965
Ordinary-Differential-Equations	3885	HOL-Nominal-Examples	17145
Smith-Normal-Form	4045	Group-Ring-Module	19718
Differential-Dynamic-Logic	4158	HOL-Analysis	44172

The Sledgehammer export allows multiple encodings of types, lambdas, and other options [5]. Since we are interested in the performance of learning-based first-order ATPs, we exported the problems in two first-order formats: TFF (also called TF0), i.e., many-sorted first-order logic, and FOF, i.e. untyped first-order logic. For all problems we pre-selected 512 relevant premises using the heuristic MePo filter [35] before the translation. This slightly overshoots the best performance (256 premises) obtained by most of the top systems[2] on the FOF and TFF problems in the recent Sledgehammer evaluation [11]. We use 512 premises because the heuristic MePo filter is known to be weaker than state-of-the-art selection systems (possibly pruning out some good premises too early), and also because the 512-premise results of the best systems in [11] are nearly identical[3] to the 256-premise results.[4]

[2] Vampire is an exception: in [11] it is best with 512 premises, likely due to its optimized SInE filter [19].
[3] In particular, CVC5 - the winner in [11] - is only 3.7% (2626/2533) stronger with 256 premises.
[4] We could have used also 1024 premises, however already with 512 premises the datasets are becoming very large, making also the training of the ML systems technically challenging.

For the other parameters, for E and Vampire we used the ones corresponding to the slice selected when no-slicing is used for a particular prover. Additionally, when extracting the FOF problems, we used the parameters used for such a slice in first-order E in the previous Isabelle version. These parameters have been optimized by the Isabelle/Sledgehammer developers based on experiments described in previous papers, e.g., [5]. To ease comparison with [11], we use the polymorphic *g*?? [11] encoding together with lambda-lifting [21] for FOF and the native monomorphic encoding with lambda-lifting for TFF0.

Since the Mirabelle export has occasional problems with some theories and encodings (theory compilation fails or does not terminate with a particular export), we initially get different numbers of problems for the FOF (293587) and TFF (386619) exports. To align the two exports, we remove the non-overlapping problems, thus obtaining 276363 problems both in FOF and TFF that correspond to each other. As usual in machine learning, we then divide this dataset into the *training*, *devel*opment (validation) and *holdout* (ultimate testing) parts. This is done by randomly shuffling the list of the problems and dividing the shuffled list 90:5:5. This means that we have 248727 problems to train our systems on, 13818 development problems for controlling the hyperparameters of the learning and building the best portfolios, and 13818 holdout problems on which the trained systems are ultimately evaluated. We also sometimes use a 13818-big subset of the training set (*small trains*). The total size of the FOF dataset is about 50G compressed by gzip to 5.4G, while for the TFF dataset it is about 90G, compressed by gzip to 7.7G. The complete datasets are publicly released at our accompanying repository.[5]

The translation of the Isabelle/HOL problems to TPTP does not preserve the names across the problems. The naming inconsistency can be as simple as the naturals being given the constant name `nat` or `nat2` in an encoded TPTP problem (this one happens because the projection int-to-nat is also called `nat` in Isabelle), depending on the order of defined constants in a given problem. Additionally, Isabelle mangles names as part of the encoding. For example in the basic theorem `List.distinct`, which states that an empty list is not equal to an applied list constructor, an instance of the empty list can look like `nil_Pr1308055047at_nat` for an empty list of products of pairs of naturals. This motivates our use of anonymous methods for ENIGMA and premise selection in this work (Section 4,5).

2.1 Differences to Related ITP/ATP Datasets

The FOF and TFF Isabelle exports we use are intended to be sound but generally sacrifice completeness to optimize ATP performance. The possible sources of incompleteness include:

- The heuristic premise filter [35] pre-selecting only a fixed number of premises that are generally not guaranteed to justify the conjecture in Isabelle.
- In the encodings, polymorphic types (such as `'a list`) are heuristically pre-instantiated (*monomorphized*) by ground types. This is an established optimization going back at least to Harrison's implementation of the MESON tactic [18] in HOL Light [17], which can be seen as a particular kind of an abstraction step when reasoning in large theories. Without a full abstraction-refinement loop [34], this is an obvious source of incompleteness, in a similar way as premise selection with a fixed premise limit.
- Limited treatment of higher-order constructs such as lambda abstraction, typically not fully encoded in the FOF and TFF problems. The encodings employ lambda-lifting, which is usually improving the ATP performance in practice, but is generally incomplete.

[5] `https://github.com/ai4reason/isa_enigma_paper`

When developing new strategies, ATPs and premise selection methods, such optimizations may be premature, having different beneficial or adverse effects on the methods. In particular, in the experiments conducted by us, we detect small amount of incompleteness already with the baseline systems. For example, CVC5 reports 256 problems in the whole TFF dataset to be countersatisfiable. On the other hand, once a proof is found, it is typically comparatively easy to replay from the minimized set of premises by any ATP.

In this sense, the monomorphized Isabelle datasets considerably differ from other datasets used for large AI/TP experiments such as the toplevel theorems in the Mizar and HOL 4 libraries [9]. There, replaying the minimized proofs may still be quite hard for ATPs, and the exports are typically striving for completeness, fully delegating various abstraction-refinement methods such as monomorphization and premise selection to the AI/TP systems that may implement more complicated procedures for them.

We measure this in more detail by comparing the clausified premise-minimized ATP problems solved by Vampire and E on the Isabelle FOF dataset (88888 problems) and the Mizar dataset (113332 problems) using several metrics computed in Table 2. The table

Table 2 Statistics of the Isabelle and Mizar clausified premise-minimized FOF problems solvable by E and Vampire. AC is the average number of clauses per problem, VC is the average number of clauses with variables per problem, EC is that for clauses with equality, iProver-10s is the number of problems solved by iProver limited to inst-gen calculus in 10s, and iProver-10s ratio is the ratio of that to the total number of problems.

Dataset	Problems	AC	VC	EC	iProver-10s	iProver-10s ratio
Isabelle FOF	88888	10.15	4.51	2.63	83015	0.93
Mizar	113332	35.55	23.16	10.31	65679	0.58
Ratio Miz/Isa		3.50	5.14	3.92		0.62

shows that the number of clauses per minimized problem is 3.5 times higher in Mizar. This may indicate the difference between the (generally harder) toplevel ITP problems and the intermediate goals. The most interesting difference is that about two thirds of the clauses in the Mizar problems contain variables, while in Isabelle this is only 44.4% of the clauses. Combined with the much higher number of clauses in the Mizar problems, this leads to 5.14 times more clauses with variables in the Mizar problems. For clauses with equality, this ratio is 3.92, i.e., also slightly higher than the ratio of the clauses. This means that the Isabelle problems are (after minimization) much more ground and non-equational, and thus likely much more amenable to instantiation-based methods than the Mizar problems. We confirm this by running iProver [31] on both sets of minimized problems using only its Inst-Gen calculus. In Mizar it solves 58% of the problems while in Isabelle 93%, i.e., 60% more.

3 Strategy Optimization for E and ENIGMA

ATP *strategies* play a critical role when proving theorems. Their targeted invention, optimization, and construction of their portfolios (*schedules*) may significantly improve the performance of the ATPs in different domains. We have also found that some ATP strategies behave better in combination with learning-based guidance of the ATPs than others, and that it often seems preferable to use a single strategy to produce the training data for ENIGMA.[6]

[6] The use of single vs multiple strategies in combination with ENIGMA is not yet strongly experimentally explored. See, e.g., [15] for a recent related analysis.

Our initial goals are thus to (i) find a strong set of E strategies for the datasets, and in particular, to (ii) find a single strong E strategy that behaves well in combination with the ENIGMA guidance. We start exploring this on the FOF dataset, evaluating our 550 BliStr/Tune [23, 47] strategies previously invented on the Mizar, Sledgehammer, HOL, AIM and TPTP problems. This is done in two rounds. In the first round, we run all the 553 strategies on a smaller sample of 500 randomly selected FOF problems solvable by Vampire's CASC mode in 30 seconds.[7] After that, the 76 most performing and orthogonal strategies from the first run are evaluated on a bigger sample of 2000 Vampire-solvable problems. This yields the following top 2 strategies in the greedy cover:

```
protokoll_X----_auto_sine13 :995
protocol_eprover_f171197f65f27d1ba69648a20c844832c84a5dd7 :198
```

The first strategy uses E's auto-mode with a strong SInE filter, selecting up to 100 premises. Unlike in the Mizar problems, the `hypos` parameter of SInE is used here, giving the same importance to the local assumptions (TPTP role `hypothesis`) as to the conjecture. We have confirmed that this performs better than SInE without the parameter on the problems. This leads us to construct the ENIGMA features differently for Isabelle problems in Section 4.

The second strategy in the greedy cover (`f1711`) is the one working best in the Mizar/MPTP setting, where it also performs well when combined with the ENIGMA guidance. It is however significantly weaker (921 vs 995 solved problems) than the first auto-mode strategy. We conjecture that this is because it does not use SInE. Adding a strong SInE filter (with the "hypos" parameter) indeed improves its performance to 1022 problems, making it the strongest E strategy on the problems. Since it is also well behaved with the ENIGMA guidance, we use it in all further experiments. The base strategy (`f1711`) without any SInE filter will be denoted as $\mathcal{B}_{\mathsf{base}}$, while the version with the SInE as $\mathcal{B}_{\mathsf{sine}}$. With the clausification changes explained next we obtain two more strategies $\mathcal{B}_{\mathsf{base3}}$ and $\mathcal{B}_{\mathsf{sine3}}$.

3.1 Clausification

Clausification can have a large influence on the operation and performance of ATPs. In a setting with many complicated formulas, naive clausification can lead to exponential blow-ups. State-of-the-art ATPs counter that by introducing definitions for subformulas. E's clausifier uses heuristic counting of the occurrences of each subformula to decide when to introduce a new definition. The default factor (called `definitional-cnf`, `dc` for short) for this used by E has been experimentally optimized to be 24 many years ago on the TPTP benchmark. This may be however suboptimal for newer large-theory corpora, especially in encodings with type guards. Also, a possible explanation for the relatively large improvement of E by the aggressive SInE filter is that the clausification explodes quite frequently on the unfiltered problems. We investigate this in several ways.

First, we simply try to clausify all FOF problems with the default E options and a timeout of 60s. This results in a gzipped total size of 21G, i.e. four times the size of the gzipped FOF problems. This is however without 28212 (10% of all) problems that fail to get clausified within 60s. This is a lot, because ITP hammers typically give the ATPs a timeout of 15-30s to solve the whole problem.

[7] We use here Vampire as a quick pre-filter for targeting the solvable problems by E strategies because in our preliminary experiments Vampire performed significantly better than E.

Table 3 Influence of the `dc` values on the clausification timeouts and size of the clausal problems.

definitional-cnf (`dc`)	1	2	3	4	24
clausifications timed out in 60s (out of 1000)	0	0	0	51	125
gzipped size of all clausified problems (MB)	36	47	163	120	77

Table 4 15s $\mathcal{B}_{\mathsf{base}}$ runs with/out SInE with different `dc` values on 1000 sample problems.

definitional-cnf (`dc`)	1	2	3	4	24
problems solved with SInE	242	268	271	266	263
problems solved without SInE	219	251	243	241	218

This leads us to an experiment with smaller values for the definitional-cnf (`dc`) parameter on a sample of 1000 training problems. We use a 60s timeout for the clausification, measure the total size of gzipped cnfs, and the number of files where the clausification timed out. The results are shown in Table 3. The `dc` value of 3 is the last one where there are no timeouts, but it already gives a 4-time blowup over `dc` = 2.

Both more aggressive premise selection and more aggressive introduction of new definitions can be used to counter the clausification blowup on the Isabelle problems. Since the SInE filter is only heuristic and usually inferior to trained premise selection, we prefer more aggressive use of new definitions. To measure how much the two methods interact, we evaluate our chosen strategy $\mathcal{B}_{\mathsf{base}}$ with and without SInE and with various `dc` values in 15s on our sample of 1000 problems. The results are summarized in Table 4. They confirm that the two methods interact a lot. Setting `dc` = 2 replaces a lot of the improvement obtained by SInE with the default `dc` = 24. Since the SInE and non-SInE versions peak at `dc` = 3 and `dc` = 2 respectively, we experiment with these values of `dc` in our further experiments. We denote $\mathcal{B}_{\mathsf{base3}}$ and $\mathcal{B}_{\mathsf{sine3}}$ the strategies obtained from $\mathcal{B}_{\mathsf{base}}$ and $\mathcal{B}_{\mathsf{sine}}$ by setting `dc` = 3.

4 ENIGMA for Isabelle

State-of-the-art automated theorem provers (ATP), such as E, Prover9, and Vampire [32], are based on the saturation loop paradigm and the *given clause algorithm* [38]. The input problem, in first-order logic (FOF), is translated into a refutationally equivalent set of clauses, and a search for contradiction is initiated. The ATP maintains two sets of clauses: *processed* (initially empty) and *unprocessed* (initially the input clauses). At each iteration, one unprocessed clause is selected (*given*), and all of the possible inferences with all the processed clauses are generated (typically using resolution, paramodulation, etc.), extending the unprocessed clause set. The selected clause is then moved to the processed clause set. Hence the invariant holds that all inferences among processed clauses have been computed.

The selection of the "right" given clause is known to be vital for the success of the proof search. The first ENIGMA systems [14, 24–26] successfully implemented various ways of machine learning guidance for the clause selection based on gradient boosting decision trees (GBDT). Next generation ENIGMA [10, 22] abstracts from symbol names with anonymization methods and additionally employs graph neural network models (GNN) for clause selection. The latest ENIGMA [16] additionally implements clause filtering of generated clauses (*parental guidance*), and overcomes a slower speed of GNN models with amortizing evaluation server.

4.1 Model Training and Given Clause Guidance

The training of ENIGMA models is usually done in a training/evaluation loop. This general approach applies both to clause guidance and when filtering the generated clauses.

1. The training data $\mathcal{T}$ are gathered from a number of previous successful proof searches. From each proof search, the training data consists of clauses processed during the proof search, labeled by flags *positive* or *negative* depending on whether they appear in the final proof. These labeled clauses are translated to a suitable format for the underlying selection model (vectors for GBDT models, and tensors for GNNs).
2. Based on data $\mathcal{T}$, a GBDT (or a GNN model) $\mathcal{M}$ is trained. This model is capable of recognizing *positive* clauses from *negatives* by assigning a score to an arbitrary clause.
3. The model $\mathcal{M}$ can be combined with an ordinary E's strategy $\mathcal{S}$ in a *cooperative* way, yielding the ENIGMA strategy $\mathcal{S} \oplus \mathcal{M}$. The ENIGMA strategy $\mathcal{S} \oplus \mathcal{M}$ uses the model $\mathcal{M}$ to guide the given clause selection inside E, and it inherits other behaviour from $\mathcal{S}$. In the cooperative setting, about 50% of the given clauses are selected as suggested by $\mathcal{M}$, while the remaining clauses are selected by the standard clause selection mechanism inherited from $\mathcal{S}$. Thus, ENIGMA compensates for a possible mistaken predictions of $\mathcal{M}$.
4. With new training data from new strategies, this process can be iterated.

4.2 Parental Guidance and Generated Clause Filtering

ENIGMA models are applied within E in two capacities: (1) given clause selection and (2) parental guidance for filtering of the generated clauses. Clausal parental guidance evaluates a new clause C based only on the features of the parents of C. Parental guidance thus serves as a fast rejection filter: generated clauses with scores below a chosen threshold are put into the *freezer* set and are only revived if E runs out of unprocessed clauses. Furthermore, such frozen clauses are never evaluated by other (possibly more expensive) heuristics. This mechanism thus effectively (and in a complete way) curbs the typically quadratic growth of the set of generated clauses. Full details can be found in previous work [16] where it was found that the the parental guidance is most effective when the concatenated feature vectors of the parents are used as an input to the machine learning model. The data for training parental guidance is generated by classifying parents of proof clauses as *positive* and all other generated clauses during a proof search as *negative*. To balance the data, the ratio of negative to positive examples is a valuable hyperparameter.

4.3 Experiments with ENIGMA

ENIGMA was so far used only with first-order logic (FOF) data in the TPTP format. In this work, we extend the usability of ENIGMA models also to simply typed first-order formulas (TFF) of the TPTP format. In the case of GBDTs models, we simply forget the type annotations. Because GBDT ENIGMA models perform symbol name anonymization by replacing symbol names by their arities, all the simple type names would get translated to the same name anyway. In the case of GNN models, we embed the type information in the clause graphs by giving nodes representing variables of the same type by the same trained numerical representation (see Section 5).

ENIGMA models embed information about the conjecture being proved inside clause vectors/tensors. In this way, ENIGMA provides conjecture-specific suggestions. The conjectures are marked in the input format with the TPTP role `conjecture`. In these experiments, we additionally treat clauses with the TPTP role `hypothesis` just like conjectures. This helps to further differentiate among various Isabelle problems.

In this work, we use ENIGMA GBDT models for clause guidance inside E (for given clause selection and filtering of generated clauses), and we use the GNN models only for the task of premise selection. Section 5 describes how the GNN models are used for premise selection. The experimental evaluation described in Section 6 presents the results of training ENIGMA models for clause selection and parental guidance.

5 Premise Selection for Isabelle via Graph Neural Networks

A number of learning-based premise selection methods have been developed for large ITP corpora and hammers in the last two decades. See [2, 7, 33, 48] for their overviews. In a large evaluation done over the Mizar corpus,[8] the strongest method turned out to be a property-invariant graph neural network (GNN) based on the architecture previously used in several settings [22, 37, 50]. We use this algorithm also for the Sledgehammer problems here.

GNNs, and in particular this architecture preserve several invariants of theorem proving data, such as insensitivity to clause ordering and literal ordering. The inference (decisions) about which premises are relevant for a conjecture are based on several rounds of neural message passing in a special graph constructed from the clauses corresponding to the formulas. The property invariant architecture also strives to be fully anonymous, in the sense that it is invariant to all symbol names: the representations of symbols are only based on their connectivity with other elements in the formula. It also has a specific encoding for argument order that allows the network to partially preserve this information and it has a special handling of negation: terms of opposite polarity are related by the corresponding operation $* -1$ in the float based representation of the network.

This set of properties allows the architecture to perform well in various theorem-proving settings. On our Isabelle datasets, the symbol and name anonymity of the GNN is particularly important. As mentioned in Section 2, the symbol names and the formula names are not used consistently here, which would make the use of non-anonymous premise selection methods difficult. In this work, a 10-layer GNN was used. The sizes of the first layer embeddings were 4, 1, 4 for the *term*, *symbol* and *clause* nodes respectively. For the rest of the layers, the term, symbol and clause nodes were represented by vectors of size 32, 64 and 32 respectively. The last, non-message passing layer that has the task of predicting a probability for each premise had 128 neurons.

The GNN was newly modified to parse and make use of the typed TFF input. To take advantage of the type information, we train separate embeddings for all types (2539 in Section 6.4) that occur more than 10.000 times in the data. The GNN uses this type embedding when reading in a variable, and the type embedding can contain information about the type of the variable. Here, for simplicity, we chose to directly learn the embeddings (initial GNN values before the start of the message passing) for the typed variable nodes. This however does not fully preserve the anonymity of the symbols in the GNN, which is one the core design principles of this neural architecture. Adding instead an extra node in the GNN for each type would allow us to preserve the anonymity also for types. In this setting the GNN would learn to understand the types based only on their use in the current problem, possibly thus generalizing better. This approach is however more complicated than our current solution and is left as future work here.

The Isabelle problems are big and their clausification by our GNN parser may result in graphs with many clauses, even when we heuristically pre-reduce the initial set of formulas proposed by the MePo filter. This poses problems with the GPU memory (32 GB on our machines) both during training the GNN and when using it for predicting the relevant clauses.

[8] `https://github.com/ai4reason/ATP_Proofs`

To counter that, we have introduced several limits related to the number of nodes in the clause graphs that allow us to skip very large clausified problems. The limit that we currently use skips any problem that contains more than 50000 term nodes after clausification (this corresponds roughly to the 95th percentile for the amount of term nodes in the problems).

6 Evaluation

We experiment with four variants of Isabelle problems. The first two are (1) FOF and (2) TFF without premise selection. Then there are two versions result from the GNN premise selector applied to the TFF data: (3) PRE_1 and (4) PRE_2.

First, Section 6.1 describes experiments with given clause guidance, and Section 6.2 describes experiments with adding parental guidance. These two experiments were partially used to obtain the training data for premise selection described in Section 6.3 and Section 6.4.

6.1 Evaluation of ENIGMA Given Clause Guidance

We perform three separate evaluations of the GBDT (LightGBM [30]) ENIGMA clause selection models on three different presentations of Isabelle problems. (1) On the FOF translation (without premise selection) in Section 6.1.1, (2) on the TFF translation (without premise selection) in Section 6.1.2, and (3) on the TFF translation with GNN premise selection in Section 6.1.3. The second premise selection dataset PRE_2 is not used here.

We experiment with combining training samples from different strategies. Different E strategies might use different term orderings affecting the clause normalization. Since the ENIGMA models are syntax based, we only combine training samples from *compatible* strategies, which perform equivalent clause normalization. At this point, we consider strategies to be *compatible* when they use the same term ordering and literal selection function.

6.1.1 Experiment FOF: First-Order Translation

Setup. First, we experiment with the FOF translations of Isabelle problems without any premise selection method applied. E supports *sine* filters to reduce the number of axioms of large problems. Since the problems have no premise selection applied, we use two versions of the E strategy to obtain training problems: $\mathcal{B}_{\mathsf{sine}}$ uses a manually selected sine filter[9] and $\mathcal{B}_{\mathsf{base}}$ does not use a sine filter. We perform three training/evaluation loops as follows.

1. *Initial training data* $\mathcal{T}_0$: Evaluation of $\mathcal{B}_{\mathsf{base}}$ and $\mathcal{B}_{\mathsf{sine}}$ on the training problems.
2. Train the model $\mathcal{L}$ on the current data $\mathcal{T}$.
3. Evaluate $\mathcal{B}_{\mathsf{base}} \oplus \mathcal{L}$ and $\mathcal{B}_{\mathsf{sine}} \oplus \mathcal{L}$ on the training problems.
4. Extend data $\mathcal{T}$ and continue with step 2.

We combine the two base strategies with model $\mathcal{L}$ in a cooperative way. With model $\mathcal{L}$ we obtain two strategies with ENIGMA guidance, that is, $\mathcal{B}_{\mathsf{base}} \oplus \mathcal{L}$ and $\mathcal{B}_{\mathsf{sine}} \oplus \mathcal{L}$.

Learning Statistics. Table 5 presents training data statistics and models evaluation for the three training/evaluation loops performed in this FOF experiments. There is:

- **training**: The column *probs* is the number of training problems in the training data, while the column *proofs* is the number of different successful proof runs, where we can have multiple proofs for a single problem. The column *rows* signifies the number of vectors in the training data, each vector corresponding to one clause in the proofs. The column *filesize* is the file size of the *compressed* training samples.

[9] `-sine='GSinE(CountFormulas,hypos,1.1,03,20000,1.0)'`

Table 5 Experiment FOF: Learning statistics (Section 6.1.1).

	notation		training				accuracy[%]			model	
l	*trains*	*model*	*probs*	*proofs*	*rows*	*filesize*	*acc*	*pos*	*neg*	*time*	*filesize*
0	$\mathcal{T}_0^{\text{FOF}}$	$\mathcal{L}_0^{\text{FOF}}$	70K	114K	8M	1.1G	92.8	89.8	93.4	0:12	54.8M
1	$\mathcal{T}_1^{\text{FOF}}$	$\mathcal{L}_1^{\text{FOF}}$	81K	255K	16M	2.3G	87.8	82.1	89.0	0:20	54.9M
2	$\mathcal{T}_2^{\text{FOF}}$	$\mathcal{L}_2^{\text{FOF}}$	84K	400K	23M	3.2G	85.6	81.9	86.5	0:31	55.1M

Table 6 Experiment FOF: ATP performance (Section 6.1.1).

	strategy		trains solved by				devels solved by			
l	*base*	*sine*	*base*	*sine*	*both*	*total*	*base*	*sine*	*both*	*total*
-	$\mathcal{S}_{\text{base}}$	$\mathcal{S}_{\text{size}}$	56 921	65 124	75 080	75 080	3114	3567	4084	4084
0	$\mathcal{S}_\star \oplus \mathcal{L}_0^{\text{FOF}}$		77 084	72 869	85 903	86 661	3888	3886	4552	4784
1	$\mathcal{S}_\star \oplus \mathcal{L}_1^{\text{FOF}}$		80 613	74 191	87 734	89 886	3933	3851	4516	4947
2	$\mathcal{S}_\star \oplus \mathcal{L}_2^{\text{FOF}}$		81 640	74 878	88 566	91 261	3963	3894	4558	5036

- **accuracy**: Columns *acc*, *pos*, *neg* show testing accuracies of each model on the testing set in percents. Column *acc* show the overall model accuracy, while columns *pos* and *neg* show testing accuracy on positive and negative testing samples separately.
- **model**: The column *time* shows the time needed for model training (in hours and minutes), and the column *size* shows the LightGBM model file size. Model file size is an important suggestion of the model ATP performance, since the model size influences the model loading time and prediction times in E.

When training a model, we set aside 5% of the training data in order to compute the testing **accuracy**. The model is trained on the remaining 95%. [10] This split is done on the level of solved problem names rather than on proofs or vectors so that all the proofs of a single problem will appear either in the 95% training subset, or all in the 5% testing subset. This is important to keep the testing set unbiased. Otherwise, the testing data can partially overlap with the training data, since two proofs of the same problem tend to be quite similar. This split on solved problem names is computed independently in every loop iteration. This split is done only on the training problems of the global training/development/holdout split used for further experiment in this paper.

From the numbers in Table 5, we can see that the number of solved problems (column *probs*) in the data increases with every loop iteration but much more slowly than the value in the column *proofs*. This means that we are obtaining duplicate proofs for already solved problems, since we include all the proofs for all solved problems in the training data in this experiment. Note that the testing accuracies decrease with increasing training data size. All the models have been built in less than 30 minutes and result in a similarly sized model file. Also note that number of *proofs* grows much faster than the problems solved (*probs*). It shows that we often prove the same problems.

[10] The numbers in the **training** columns are only on the training 95% subset.

Table 7 Experiment TFF: Learning statistics (Section 6.1.2).

	notation		**training**				**accuracy**[%]			**model**	
l	*trains*	*model*	*probs*	*proofs*	*rows*	*size*	*acc*	*pos*	*neg*	*time*	*size*
0	$\mathcal{T}_0^{\text{TFF}}$	$\mathcal{L}_0^{\text{TFF}}$	108K	186K	10,3M	1.2G	89.6	86.2	90.2	12:36	54.8M
1	$\mathcal{T}_1^{\text{TFF}}$	$\mathcal{L}_1^{\text{TFF}}$	114K	383K	19,6M	2.2G	85.0	78.8	86.2	20:29	55.0M
2	$\mathcal{T}_2^{\text{TFF}}$	$\mathcal{L}_2^{\text{TFF}}$	117K	587K	27,9M	3.1G	82.6	77.4	83.8	20:52	55.1M
3	$\mathcal{T}_3^{\text{TFF}}$	$\mathcal{L}_3^{\text{TFF}}$	122K	822K	39,3M	4.3G	81.4	77.8	82.2	23:17	55.2M
4	$\mathcal{T}_4^{\text{TFF}}$	$\mathcal{L}_4^{\text{TFF}}$	123K	1.03M	48,6M	5.3G	80.9	77.6	81.7	29:46	55.3M

ATP Evaluation. Table 6 shows the ENIGMA models performance separately on training (**trains**) and on development problems (**devel**). Since the development problems were not used during the training in any way, this evaluation tells how much are the ENIGMA model over-fitting on the training files.

Every row describes the performance of two strategies specified in the column **strategy**. Problems solved by the two strategies individually are in the first two **bold** columns. *Italics* values display a total cover of set of strategies. The column *both* shows the number of problems solved both by the two strategies together. This is helpful to estimate the complementarity of *base* and *sine* strategies. Two strategies are *complementary*, when they solve different problems. The column *total* shows the cumulative number of problems solved by all the current strategies (above in the table).

In Table 6, we see that the *sine* strategy performed better than *base* initially. However, from the first learning the *base* strategy dominates. This suggests that ENIGMA learns to do premise selection on its own to some extent (when trained on the samples from the *sine* strategy). All *base* and *sine* strategies are, however, quite complementary. In total, we start with 75 080 solved problems and we end up with 91 261 after the learning, almost 22% improvement on trains (23% on devels). The best single strategy is improved by 25% on trains (and by 11% on devels).

It is interesting to observe, how the *base* strategies in one iteration improves on both *base* and *sine* from the previous iteration, as if merging the two strategies into one. It suggests that additional proof samples from compatible but complementary strategies could lead to an additional improvement. We further investigate this in the next experiment (Section 6.1.2).

6.1.2 Experiment TFF: Typed First-Order Formulae

Setup. We perform a similar experiment as for the FOF, but this time targeted to the TFF Isabelle translation.

1. Again, we start with the training data obtained by the evaluation of $\mathcal{B}_{\mathsf{base}}$ and $\mathcal{B}_{\mathsf{sine}}$.
2. We run three iterations of the training/evaluation loop.
3. After the three iterations, we additionally evaluate two more pure E strategies $\mathcal{B}_{\mathsf{base3}}$ and $\mathcal{B}_{\mathsf{sine3}}$ which improve on $\mathcal{B}_{\mathsf{base}}$ and $\mathcal{B}_{\mathsf{sine}}$ by adjusting E's clausification algorithm (switch E's option "`definitional-cnf`" from 24 to 3).
4. We perform two more training/evaluation loops with the expanded training data.

Table 8 Experiment TFF: ATP performance (Section 6.1.2).

	strategy		trains solved by				devels solved by			
l	*base*	*sine*	*base*	*sine*	*both*	*total*	*base*	*sine*	*both*	*total*
-	$\mathcal{S}_{\mathsf{base}}$	$\mathcal{S}_{\mathsf{size}}$	100 259	98 317	114 838	114 838	5532	5403	6347	6347
0	$\mathcal{S}_\star \oplus \mathcal{L}_0^{\mathrm{TFF}}$		108 377	101 271	118 262	121 353	5468	5347	6222	6692
1	$\mathcal{S}_\star \oplus \mathcal{L}_1^{\mathrm{TFF}}$		113 729	103 382	121 995	124 795	5788	5471	6466	6894
2	$\mathcal{S}_\star \oplus \mathcal{L}_2^{\mathrm{TFF}}$		115 790	104 270	123 400	126 344	5934	5505	6547	6894
*	$\mathcal{S}_{\mathsf{base3}}$	$\mathcal{S}_{\mathsf{sine3}}$	106 132	100 904	118 925	132 552	5881	5522	6515	7160
3	$\mathcal{S}_{\star 3} \oplus \mathcal{L}_3^{\mathrm{TFF}}$		122 492	107 035	127 955	133 222	6293	5673	6785	7280
4	$\mathcal{S}_{\star 3} \oplus \mathcal{L}_4^{\mathrm{TFF}}$		122 931	107 316	128 339	133 762	6277	5704	6812	7326

Learning Statistics. Table 7 presents the machine learning evaluation (in the same format as Table 5 described in Section 6.1.1). Before the fourth loop ($l = 3$), we additionally evaluate all the strategies $\mathcal{S} \oplus \mathcal{L}$, for $\mathcal{S}$ ranging over $\mathcal{B}_{\mathsf{base3}}$ and $\mathcal{B}_{\mathsf{sine3}}$, and for $\mathcal{L}$ ranging over the models of the first three loops. This gives us additional training data for the fourth iteration, reflected in the table by a sudden increase in both solved problems (*probs*) and *proof* count (in the row $l = 3$). We see similar training times and model sizes as in the FOF experiment.

ATP Evaluation. Table 8 presents the ATP evaluation (in the same format as Table 6 described in Section 6.1.1). As opposed to the FOF experiment, the *base* strategies dominate from the beginning. Both strategies are still highly complementary. The evaluation of $\mathcal{B}_{\mathsf{base3}}$ and $\mathcal{B}_{\mathsf{sine3}}$ strategies boosts the number of solved trains from 126 344 to 132 552. This highly improves the performance of the best strategy (*base*) in the fourth iteration ($l = 3$) from 115 790 to 122 492, that is, by 5.8%. It shows that additional external training data can be quite useful during the training. We further investigate this issue in the next experiment (Section 6.1.3).

6.1.3 Experiment PRE_1: First GNN Premise Selection

Setup. Here we experiment with GNN premise selection data PRE_1 obtained by applying GNN premise selection to the TFF problems. The GNN premise selection produces several collections of the training problems (called *slices*) with a slightly different clause selection criterion. We experiment with two slices PRE_1^{-1} and PRE_1^{64}, which were experimentally found well performing and complementary. Our first experiment is aimed at generating a large collection of training samples.

1. We perform three loops of training/evaluation, just as in the TFF experiment, separately on PRE_1^{-1} and PRE_1^{64}. We loop with the base strategies $\mathcal{B}_{\mathsf{base3}}$ and $\mathcal{B}_{\mathsf{sine3}}$.
2. We merge the training data from the previous two separate experiments and perform three more loops on the merged data. However, we drop the *sine* strategies and evaluate only the strategy $\mathcal{B}_{\mathsf{base3}} \oplus \mathcal{L}$ on the two PRE_1 slices.
3. From the above we collect a large database of 108K proved training problems. Since the collection can contain duplicate proofs of a single problem, we select just three proofs per problem. We use the proof pos/neg ratios as a measure of proof similarity, and select proofs thusly different.
4. The training data from the last step, denoted $\mathcal{T}_{\mathsf{three}}^{\mathrm{PRE}_1}$, gives us one final model $\mathcal{L}_{\mathsf{three}}^{\mathrm{PRE}_1}$.

Table 9 Experiment PRE_1: Learning statistics (Section 6.1.3).

notation		training				accuracy[%]			model	
trains	*model*	*probs*	*proofs*	*rows*	*size*	*acc*	*pos*	*neg*	*time*	*size*
$\mathcal{T}_{\mathsf{three}}^{\mathrm{PRE}_1}$	$\mathcal{L}_{\mathsf{three}}^{\mathrm{PRE}_1}$	108K	186K	10,3M	1.2G	89.6	86.2	90.2	12:36	54.8M
$\mathcal{T}_{\mathsf{six}}^{\mathrm{PRE}_1}$	$\mathcal{L}_{\mathsf{six}}^{\mathrm{PRE}_1}$	133K	763K	28,6M	3.26G	77.6	76.8	78.0	25:44	77.9M

Table 10 Experiment PRE_1: ATP performance (Section 6.1.3).

	trains solved by				devels solved by			
strategy	PRE_1^{-1}	PRE_1^{64}	*both*	*total*	PRE_1^{-1}	PRE_1^{64}	*both*	*total*
$\mathcal{S} = \mathcal{S}_{\mathsf{base3}}$	122 196	117 341	126 323	126 323	6706	6462	6955	6955
$\mathcal{S} \oplus \mathcal{L}_{\mathsf{three}}^{\mathrm{PRE}_1}$	127 606	120 248	129 971	132 431	6800	6495	6994	7251
$\mathcal{S} \oplus \mathcal{L}_{\mathsf{six}}^{\mathrm{PRE}_1}$	132 063	123 229	134 544	135 823	6994	6591	7153	7380

Next experiment tries to gather even more training samples.

1. We additionally consider training data from the previous TFF experiments.
2. We gather even more valuable training samples from ENIGMA parental guidance experiments on slices PRE_1^{-1} and PRE_1^{64}.
3. We select three proofs per problem from TFF samples.
4. We select three proofs per problem from PRE_1 samples.
5. The training data $\mathcal{T}_{\mathsf{sixes}}^{\mathrm{PRE}_1}$ contain six proofs per problem and yield model $\mathcal{L}_{\mathsf{sixes}}^{\mathrm{PRE}_1}$.

Learning Statistics. Table 9 presents the machine learning evaluation (in the same format as Table 5 described in Section 6.1.1). Note the huge difference in the number of *proofs*, resulting in much larger *training data size*. The second training data include proofs of more than 25K additional problems (*probs*). Training times and model sizes clearly reflect the training file size.

ATP Evaluation. Table 10 presents the ATP evaluation (in the same format as Table 6 described in Section 6.1.1). Here, however, we evaluate the single strategy $\mathcal{B}_{\mathsf{base3}} \oplus \mathcal{L}$ on slices PRE_1^{-1} and PRE_1^{64}, instead of using two *base* and *sine* strategies.

Firstly, we note the effect of the premise selection itself. The performance on $\mathcal{B}_{\mathsf{sine3}}$ improved by more than 15% from the previous experiment (from 106 132 to 122 196). The model $\mathcal{L}_{\mathsf{three}}^{\mathrm{PRE}_1}$ performs quite well, being trained on proofs 108K problems, it solves almost 128K problems. The model $\mathcal{L}_{\mathsf{six}}^{\mathrm{PRE}_1}$ further boosts the performance, showing that combining of training data from various compatible sources might be beneficial. Comparing the performance on trains with the performance on devels, we can conclude that ENIGMA LightGBM clause selection models slightly overfit but they are still capable of generalization.

6.2 Evaluation of the Parental Guidance

Setup. Parental guidance models are co-trained with clause selection models in a series of loops over the training data. In each loop iteration, the LightGBM parameters for parental guidance models are tuned using a series of grid searches with Optuna [1]. These are the

Table 11 Parental guidance iterations on small trains, devel, and holdout (13 818 problems in 15s). Loops $\mathcal{L}_1$ and $\mathcal{L}_2$ are run on TFF data, $\mathcal{L}_3$ to $\mathcal{L}_8$ on PRE_1^{-1}, and $\mathcal{L}_9$ and $\mathcal{L}_{10}$ on PRE_2^{-1}.

	$\mathcal{L}_1$	$\mathcal{L}_2$	$\mathcal{L}_3$	$\mathcal{L}_4$	$\mathcal{L}_5$	$\mathcal{L}_6$	$\mathcal{L}_7$	$\mathcal{L}_8$	$\mathcal{L}_9$	$\mathcal{L}_{10}$
small trains	6475	6718	7081	7140	7312	7351	7407	7417	7647	**7705**
devel	6241	6462	6928	6892	6566	6850	7070	7115	7277	**7379**
holdout	6251	6459	6886	6843	6581	6816	7015	7062	7352	**7395**

number of leaves, the bagging fraction and frequency, the minimum number of samples to create a new leaf, and L1 and L2 regularization. The learning rate is fixed at 0.15, the maximum tree depth is capped at 256 and the number of trees is 250. The number of leaves is varied between 256 and 3333. The best result of each grid search is used for the next parameter's grid search. Accuracy on positive training examples is considered twice as important as the accuracy on negatives when choosing which parameters perform best. There are multiple reasons for this. A primary reason is that the confidence in positive examples is higher than confidence for the classification of negatives because a negative clause in one successful proof search could be positive in another proof search. The resulting model is evaluated with the nine parental filtering thresholds, $\{0.03, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7\}$, over a set of 300 problems from the development set for 30 seconds. This is done for the vanilla TFF problems as well as the problems with premise selection slices. The best run (as evaluated by a greedy cover) on each version of the problems is then run on the full training set. Then the problems from runs in the total greedy cover are used as data for the next iteration of looping. This means that some problems can have over 10 proofs in the training data.

Iterations. The training of parental guidance was done with the aim to develop as strong a performance as possible, using diverse data. The models for loops $\mathcal{L}_1$ and $\mathcal{L}_2$ are run and trained on the TFF data that do not use premise selection. Models $\mathcal{L}_3$ and $\mathcal{L}_4$ are only run on the small trains set. The models $\mathcal{L}_3$ to $\mathcal{L}_8$ are run on PRE_1. Finally, models $\mathcal{L}_9$ and $\mathcal{L}_{10}$ are run on PRE_2^{-1}. The largest performance jumps correspond to the addition of premise selection (Table 11). The strongest parental guidance models are always on the PRE^{-1} premise selection data and the PRE_1^{64} slices provide fewer complementary problems than the baseline TFF problems.

The best model $\mathcal{L}_{10}$, with the second premise selection slices, PRE_2^{-1}, proves 168 problems (56%) in 30s on the parameter tuning development set, and 137 893 problems (55.4%) on the training set in 15s. In 30s, $\mathcal{L}_{10}$ proves 7472 problems (54.1%) on the development set and 7466 problems (54%) on the holdout. Without premise selection, $\mathcal{L}_{10}$ proves 133 390 training problems (53.6%), which indicates that training on the premise selection data transfers back to the original problems. The remaining results are presented in Table 11. This parental+ENIGMA model is our final product. It solves 7395 holdout problems in 15s, thus significantly improving over unguided E and also over all other ATPs and SMTs. It also outperforms all other ATPs and SMTs even when they use our best premises (Table 12).

6.3 First Training of Premise Selection on TFF Problems (PRE_1)

We have done several large experiments with the GNN-based premise selection (Section 5), first in the untyped and then in the typed setting. For lack of space we include below only the two final experiments on the TFF data, where most of the ENIGMA runs were done.

For the first round of training the GNN on the TFF data we are using the proof data produced only by the base sine/nosine TFF runs of unguided E and the first three ENIGMA iterations on the TFF training set. Altogether these runs produce 1701284 proof dependencies. These dependencies are first deduplicated to 353875, and then we also for every problem P remove all premise sets subsumed by a smaller premise set. This further decreases the size of the dataset to 242432 proof dependencies, for 131309 unique solved problems. Most of the solved problems (80993) have after this redundancy elimination only one solution, while for the remaining ones we get from 2 to 16 different solutions. Since problems with 1 to 3 solutions dominate the dataset (163650 out of the total 242432 solutions), we do not do further pruning of over-represented proofs for the training (as in the next training run).

For this first training and prediction we do not yet use the new typed extensions of the GNN. Instead, all TFF formulas are stripped of their type information and given to the network as untyped FOF. Each problem uses its original conjecture, the positives are the premises used in a given proof and the negatives are all other premises for that problem (i.e., all the MePo premises). This sometimes leads to large training inputs, so we normalize them to have size at most 500KB by randomly removing negatives. The whole training dataset has size 46GB. We then train the GNN on it with batch size 10, learning rate 0.005, and with balancing the loss on the positive and negative premises.

The training for two full epochs on an NVIDIA Volta 100 takes about 12 hours, saving the weights 16 times. The balanced accuracy increases from 0.8533 to the final 0.9067 (0.9061 vs 0.9073 on positives vs negatives) in our final snapshot, which we then use for prediction over all 276363 problems. This is parallelized over four GPUs, taking several hours. For each problem we use the predictions to produce 5 premise selections based on the GNN score threshold (1,0,-1,-2,-3,-4), and 5 premise selections based on old-style top slices of the ranked premises (16,32,64,128,256). We do a small search with 200 development problems and the base strategy over this grid, which is won by the -1-based predictions, best complemented by the 64-based predictions. These premise selections are denoted PRE_1^{-1} and PRE_1^{64} in the other parts of this paper. E/ENIGMA are then evaluated on both of them, while we also evaluate other systems only on the -1-based predictions (Table 12).

6.4 Second Training of Premise Selection on TFF Problems (PRE_2)

The second premise selection training is done by the typed version of the GNN (Section 5), using explicitly 2539 types that occur with frequency higher than 10000 in the training data. The remaining types (over 300000 in the training set) are mapped to the same generic embedding, which means that the GNN treats them all as the same type. The overhead for the 2539 distinguished most frequent types increases the size of the GNN only by 100kb. The training uses again a batch size of 20 and a learning rate of 0.0005. The training dataset is created from all TFF training problems solved in the previous loops, both by E/ENIGMA and CVC5 and Vampire. This gives 823141 unique premise selections for 146576 solved problems. The 823141 unique premise selections are again minimized with respect to subsumption, reducing them to 488186 minimal premise selections. To address the imbalance caused by having various numbers of proofs for a single problem in the training set, we keep at most three proofs for each problem. This further reduces the set to 292080 examples. The examples are again all reduced to a size of at most 500KB.

Table 12 Final comparison with non-ENIGMA systems: E2.6 with its auto-schedule, CVC5, and Vampire-CASC (master 4909). Each run standalone (MePo predictions) and with the first/second -1 GNN TFF predictions. The last entry is the final/best $Loop_{10}$ (parental) ENIGMA (Section 6.2).

method	E auto-sched.	CVC5	Vampire	$\mathcal{L}_{10}$ ENIGMA
15s devel, no premsel	5891	7053	6452	7133
15s holdout, no premsel	5903	7051	6454	7139
30s holdout, no premsel	6089	7140	6945	7170
15s devel, preds -1 (1st round)	6968	7211	7023	7191
15s holdout, preds -1 (1st round)	6956	7158	6978	7155
15s devel, preds -1 (2nd round)	7074	7394	7132	7379
15s holdout, preds -1 (2nd round)	7066	7372	7118	**7395**
30s holdout, preds -1 (2nd round)	7139	7398	7397	**7466**

This results in our final training set with an overall size of about 60GB. Since the reduction of the TFF inputs does not generally guarantee to prevent a blow-up during the clausification, we also further use here a size limit of 50000 nodes inside the GNN parser (Section 5) and filter out such large graphs which may otherwise deplete the GPU memory. The GNN is trained for full two epochs on the data, taking about one day on a single NVIDIA V100 GPU and storing the weight files 15 times per epoch. For producing the final predictions, we take the 28th weights with the highest balanced accuracy of 0.9221 (0.9391 / 0.9051 for positives/negatives). We produce the same grid of predictions as in the first round for all problems. The -1-based predictions are again the winner, best complemented by the 0-based predictions. These premise selections are denoted PRE_2^{-1} and PRE_2^0 in the other parts of this paper. Table 12 shows that also all non-ENIGMA systems benefit from the GNN predictions, and that the second round improves over the first round of predictions for all of them.

7 Conclusion

We have developed versions of the ENIGMA systems and neural premise selectors for the Isabelle Sledgehammer problems. Our best single-strategy system using the parental ENIGMA guidance and the typed GNN premise selection solves 7395 holdout problems in 15s, improving on original E's auto-schedule performance (5903) by 25.3%. It also improves on all other ATPs and SMTs, both when used standalone and when used in conjunction with our best neural premise selection. To achieve this, we have produced large corpora of Isabelle problems for training and evaluation of the AI/TP methods, and developed new extensions of our systems, especially for the typed setting.

References

1 Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

2 Jesse Alama, Tom Heskes, Daniel Kühlwein, Evgeni Tsivtsivadze, and Josef Urban. Premise selection for mathematics by corpus analysis and kernel methods. *J. Autom. Reasoning*, 52(2):191–213, 2014. doi:10.1007/s10817-013-9286-5.

3 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.

4 Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. The tactician - A seamless, interactive tactic learner and prover for coq. In *CICM*, volume 12236 of *Lecture Notes in Computer Science*, pages 271–277. Springer, 2020.

5 Jasmin Christian Blanchette, Sascha Böhme, Andrei Popescu, and Nicholas Smallbone. Encoding monomorphic and polymorphic types. In Nir Piterman and Scott A. Smolka, editors, *TACAS*, volume 7795 of *LNCS*, pages 493–507. Springer, 2013. doi:10.1007/978-3-642-36742-7_34.

6 Jasmin Christian Blanchette, Max W. 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 - International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings*, volume 9150 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2015. doi:10.1007/978-3-319-20615-8_1.

7 Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *J. Formalized Reasoning*, 9(1):101–148, 2016. doi:10.6092/issn.1972-5787/4593.

8 Sascha Böhme and Tobias Nipkow. Sledgehammer: Judgement Day. In Jürgen Giesl and Reiner Hähnle, editors, *IJCAR*, volume 6173 of *LNCS*, pages 107–121. Springer, 2010. doi:10.1007/978-3-642-14203-1_9.

9 Chad E. Brown, Thibault Gauthier, Cezary Kaliszyk, Geoff Sutcliffe, and Josef Urban. GRUNGE: A grand unified ATP challenge. In *CADE*, volume 11716 of *Lecture Notes in Computer Science*, pages 123–141. Springer, 2019.

10 Karel Chvalovský, Jan Jakubův, Martin Suda, and Josef Urban. ENIGMA-NG: efficient neural and gradient-boosted inference guidance for E. In Pascal Fontaine, editor, *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, volume 11716 of *Lecture Notes in Computer Science*, pages 197–215. Springer, 2019. doi:10.1007/978-3-030-29436-6_12.

11 Martin Desharnais, Petar Vukmirović, Jasmin Blanchette, and Makarius Wenzel. Seventeen provers under the hammer, 2022. URL: https://matryoshka-project.github.io/pubs/seventeen.pdf.

12 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.

13 Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. Tactictoe: Learning to prove with tactics. *J. Autom. Reason.*, 65(2):257–286, 2021.

14 Zarathustra Goertzel, Jan Jakubův, and Josef Urban. Enigmawatch: Proofwatch meets ENIGMA. In Serenella Cerrito and Andrei Popescu, editors, *Automated Reasoning with Analytic Tableaux and Related Methods - 28th International Conference, TABLEAUX 2019, London, UK, September 3-5, 2019, Proceedings*, volume 11714 of *Lecture Notes in Computer Science*, pages 374–388. Springer, 2019. doi:10.1007/978-3-030-29026-9_21.

15 Zarathustra Amadeus Goertzel. Make E smart again (short paper). In *IJCAR (2)*, volume 12167 of *Lecture Notes in Computer Science*, pages 408–415. Springer, 2020.

16 Zarathustra Amadeus Goertzel, Karel Chvalovský, Jan Jakubuv, Miroslav Olsák, and Josef Urban. Fast and slow enigmas and parental guidance. In Boris Konev and Giles Reger, editors, *Frontiers of Combining Systems - 13th International Symposium, FroCoS 2021, Birmingham, UK, September 8-10, 2021, Proceedings*, volume 12941 of *Lecture Notes in Computer Science*, pages 173–191. Springer, 2021. doi:10.1007/978-3-030-86205-3_10.

17 John Harrison. HOL Light: A tutorial introduction. In Mandayam K. Srivas and Albert John Camilleri, editors, *FMCAD*, volume 1166 of *LNCS*, pages 265–269. Springer, 1996. doi:10.1007/BFb0031814.

18 John Harrison. Optimizing Proof Search in Model Elimination. In M. McRobbie and J.K. Slaney, editors, *Proceedings of the 13th International Conference on Automated Deduction*, number 1104 in LNAI, pages 313–327. Springer, 1996.

19 Krystof Hoder and Andrei Voronkov. Sine qua non for large theory reasoning. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *CADE*, volume 6803 of *LNCS*, pages 299–314. Springer, 2011. doi:10.1007/978-3-642-22438-6_23.

20 Edvard K. Holden and Konstantin Korovin. Heterogeneous heuristic optimisation and scheduling for first-order theorem proving. In *CICM*, volume 12833 of *Lecture Notes in Computer Science*, pages 107–123. Springer, 2021.

21 R. J. M. Hughes. Super-combinators a new implementation method for applicative languages. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, LFP '82, pages 1–10, New York, NY, USA, 1982. Association for Computing Machinery. doi:10.1145/800068.802129.

22 Jan Jakubův, Karel Chvalovský, Miroslav Olsák, Bartosz Piotrowski, Martin Suda, and Josef Urban. ENIGMA anonymous: Symbol-independent inference guiding machine (system description). In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 448–463. Springer, 2020. doi:10.1007/978-3-030-51054-1_29.

23 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.

24 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.

25 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.

26 Jan Jakubův and Josef Urban. Hammering Mizar by learning clause guidance. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*, pages 34:1–34:8. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ITP.2019.34.

27 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.

28 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.

29 Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olsák. Reinforcement learning of theorem proving. In *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.

30 Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *NIPS*, pages 3146–3154, 2017.

31 Konstantin Korovin. iprover - an instantiation-based theorem prover for first-order logic (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 292–298. Springer, 2008. doi:10.1007/978-3-540-71070-7_24.

32 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.

33 Daniel Kühlwein, Twan van Laarhoven, Evgeni Tsivtsivadze, Josef Urban, and Tom Heskes. Overview and evaluation of premise selection techniques for large theory mathematics. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *IJCAR*, volume 7364 of *LNCS*, pages 378–392. Springer, 2012. doi:10.1007/978-3-642-31365-3_30.

34 Julio César López-Hernández and Konstantin Korovin. An abstraction-refinement framework for reasoning with large theories. In *IJCAR*, volume 10900 of *Lecture Notes in Computer Science*, pages 663–679. Springer, 2018.

35 Jia Meng and Lawrence C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *J. Applied Logic*, 7(1):41–57, 2009. doi:10.1016/j.jal.2007.07.004.

36 Yutaka Nagashima and Ramana Kumar. A proof strategy language and proof script generation for isabelle/hol. In *CADE*, volume 10395 of *Lecture Notes in Computer Science*, pages 528–545. Springer, 2017.

37 Miroslav Olsák, Cezary Kaliszyk, and Josef Urban. Property invariant embedding for automated reasoning. In Giuseppe De Giacomo, Alejandro Catalá, Bistra Dilkina, Michela Milano, Senén Barro, Alberto Bugarín, and Jérôme Lang, editors, *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 1395–1402. IOS Press, 2020. doi:10.3233/FAIA200244.

38 Ross A. Overbeek. A new class of automated theorem-proving algorithms. *J. ACM*, 21(2):191–200, April 1974. doi:10.1145/321812.321814.

39 Bartosz Piotrowski and Josef Urban. ATPboost: Learning premise selection in binary setting with ATP feedback. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*, volume 10900 of *Lecture Notes in Computer Science*, pages 566–574. Springer, 2018. doi:10.1007/978-3-319-94205-6_37.

40 Bartosz Piotrowski and Josef Urban. Stateful premise selection by recurrent neural networks. In *LPAR*, volume 73 of *EPiC Series in Computing*, pages 409–422. EasyChair, 2020.

41 Michael Rawson and Giles Reger. lazycop: Lazy paramodulation meets neurally guided search. In *TABLEAUX*, volume 12842 of *Lecture Notes in Computer Science*, pages 187–199. Springer, 2021.

42 Simon Schäfer and Stephan Schulz. Breeding theorem proving heuristics with genetic algorithms. In Georg Gottlob, Geoff Sutcliffe, and Andrei Voronkov, editors, *Global Conference on Artificial Intelligence, GCAI 2015, Tbilisi, Georgia, October 16-19, 2015*, volume 36 of *EPiC Series in Computing*, pages 263–274. EasyChair, 2015. URL: http://www.easychair.org/publications/paper/Breeding_Theorem_Proving_Heuristics_with_Genetic_Algorithms, doi:10.29007/gms9.

43 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.

44 Stephan Schulz, Simon Cruanes, and Petar Vukmirovic. Faster, higher, stronger: E 2.3. In Pascal Fontaine, editor, *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, volume 11716 of *Lecture Notes in Computer Science*, pages 495–507. Springer, 2019. doi:10.1007/978-3-030-29436-6_29.

45 René Thiemann and Christian Sternagel. Certification of termination proofs using ceta. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 452–468. Springer, 2009. doi:10.1007/978-3-642-03359-9.

46 Josef Urban. MPTP - Motivation, Implementation, First Experiments. *J. Autom. Reasoning*, 33(3-4):319–339, 2004. doi:10.1007/s10817-004-6245-1.

47 Josef Urban. BliStr: The Blind Strategymaker. In Georg Gottlob, Geoff Sutcliffe, and Andrei Voronkov, editors, *Global Conference on Artificial Intelligence, GCAI 2015, Tbilisi, Georgia, October 16-19, 2015*, volume 36 of *EPiC Series in Computing*, pages 312–319. EasyChair, 2015. URL: http://www.easychair.org/publications/paper/BliStr_The_Blind_Strategymaker, doi:10.29007/8n7m.

48 Josef Urban. ERC project AI4Reason final scientific report, 2021. URL: http://grid01.ciirc.cvut.cz/~mptp/ai4reason/PR_CORE_SCIENTIFIC_4.pdf.

49 Robert Veroff. Using hints to increase the effectiveness of an automated reasoning program: Case studies. *J. Autom. Reasoning*, 16(3):223–239, 1996. doi:10.1007/BF00252178.

50 Zsolt Zombori, Josef Urban, and Miroslav Olsák. The role of entropy in guiding a connection prover. In *TABLEAUX*, volume 12842 of *Lecture Notes in Computer Science*, pages 218–235. Springer, 2021.

Accelerating Verified-Compiler Development with a Verified Rewriting Engine

Jason Gross
CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA
Machine Intelligence Research Institute, Berkeley, CA, USA

Andres Erbsen
CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

Jade Philipoom
CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA
Google, London, UK

Miraya Poddar-Agrawal
Reed College, Portland, OR, USA

Adam Chlipala
CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

Abstract

Compilers are a prime target for formal verification, since compiler bugs invalidate higher-level correctness guarantees, but compiler changes may become more labor-intensive to implement, if they must come with proof patches. One appealing approach is to present compilers as sets of algebraic rewrite rules, which a generic engine can apply efficiently. Now each rewrite rule can be proved separately, with no need to revisit past proofs for other parts of the compiler. We present the first realization of this idea, in the form of a framework for the Coq proof assistant. Our new Coq command takes normal proved theorems and combines them automatically into fast compilers with proofs. We applied our framework to improve the Fiat Cryptography toolchain for generating cryptographic arithmetic, producing an extracted command-line compiler that is about 1000× faster while actually featuring simpler compiler-specific proofs.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Equational logic and rewriting; Software and its engineering → Compilers; Software and its engineering → Translator writing systems and compiler generators

Keywords and phrases compiler verification, rewriting engines, cryptography

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.17

Related Version *Full Version*: `https://arxiv.org/abs/2205.00862`

Supplementary Material *Software (Source Code)*: `https://github.com/mit-plv/rewriter/tree/ITP-2022-perf-data`; archived at `swh:1:rev:1787ab401a7e71afc9937010e2e155e4b1594ab5`
Software (Source Code): `https://github.com/mit-plv/fiat-crypto/tree/perf-testing-data-ITP-2022-rewriting`; archived at `swh:1:rev:72fe0dddee5e6dceeab0b8a2e6a745abf5287d3e`

Funding This work was supported in part by a Google Research Award, National Science Foundation grants CCF-1253229, CCF-1512611, and CCF-1521584, and the National Science Foundation Graduate Research Fellowship under Grant Nos. 1122374 and 1745302. Any opinion, 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

Formally verified compilers like CompCert [13] and CakeML [12] are success stories for proof assistants, helping close a trust gap for one of the most important categories of software infrastructure. A popular compiler cannot afford to stay still; developers will add new backends, new language features, and better optimizations. Proofs must be adjusted as these improvements arrive. It makes sense that the author of a new piece of compiler code must prove its correctness, but ideally there would be no need to revisit old proofs. There has been limited work, though, on avoiding that kind of coupling. Tatlock and Lerner [17] demonstrated a streamlined way to extend CompCert with new verified optimizations driven by dataflow analysis, but we are not aware of past work that supports easy extension for compilers from functional languages to C code. We present our work targeting that style.

One strategy for writing compilers modularly is to exercise foresight in designing a core that will change very rarely, such that feature iteration happens outside the core. Specifically, phrasing the compiler in terms of rewrite rules allows clean abstractions and conceptual boundaries [11]. Then, most desired iteration on the compiler can be achieved through iteration on the rewrite rules.

It is surprisingly difficult to realize this modular approach with good performance. Verified compilers can either be proof-producing (certifying) or proven-correct (certified). Proof-producing compilers usually operate on the functional languages of the proof assistants that they are written in, and variable assignments are encoded as let binders. All existing proof-producing rewriting strategies scale at least quadratically in the number of binders. This performance scaling is inadequate for applications like Fiat Cryptography [8] where the generated code has 1000s of variables in a single function. Proven-correct compilers do not suffer from this asymptotic blowup in the number of binders.

In this paper, we present **the first proven-correct compiler-builder toolkit parameterized on rewrite rules**. Arbitrary sets of Coq theorems (quantified equalities) can be assembled by a single new Coq command into an extraction-ready verified compiler. We did not need to extend the trusted code base, so our compiler compiler need not be trusted. We achieve both good performance of compiler runs and good performance of generated code, via addressing a number of scale-up challenges vs. past work.

We evaluate our toolkit by replacing a key component of Fiat Cryptography [8], a Coq library that generates code for big-integer modular arithmetic at the heart of elliptic-curve-cryptography algorithms. Routines generated (with proof) with Fiat Cryptography now ship with all major Web browsers and all major mobile operating systems. With our improved compiler architecture, it became easy to add two new backends and a variety of new supported source-code features, and we were easily able to try out new optimizations.

Replacing Fiat Cryptography's original compiler with the compiler generated by our toolkit has two additional benefits. Fiat Cryptography was previously only used successfully to build C code for two of the three most widely used curves (P-256 and Curve25519). Our prior version's execution timed out trying to compile code for the third most widely used curve (P-384). Using our new toolkit has made it possible to generate compiler-synthesized code for P-384 while generating completely identical code for the primes handled by the previous version, about 1000× more quickly. Additionally, Fiat Cryptography previously required source code to be written in continuation-passing style, and our compiler has enabled a direct-style approach, which pays off in simplifying theorem statements and proofs.

1.1 Related Work

Assume our mission is to take libraries of purely functional combinators, apply them to compile-time parameters, and compile the results down to lean C code. Furthermore, we ask for machine-checked proofs that the C programs preserve the behavior of the higher-order functional programs we started with. What good ideas from the literature can we build on?

Hickey and Nogin [11] discuss at length how to build compilers around rewrite rules. "All program transformations, from parsing to code generation, are cleanly isolated and specified as term rewrites." While they note that the correctness of the compiler is thus reduced to the correctness of the rewrite rules, they did not prove correctness mechanically. Furthermore, it is not clear that they manage to avoid the asymptotic blow-up associated with proof-producing rewriting of deeply nested let-binders. They give no performance numbers, so it is hard to say whether or not their compiler performs at the scale necessary for Fiat Cryptography. Their rewrite-engine driver is unproven OCaml code, while we will produce custom drivers with Coq proofs.

$\mathcal{R}_{tac}$ [14] is a more general framework for verified proof tactics in Coq, including an experimental reflective version of `rewrite_strat` supporting arbitrary setoid relations, unification variables, and arbitrary semidecidable side conditions solvable by other verified tactics, using de Bruijn indexing to manage binders. We found that $\mathcal{R}_{tac}$ misses a critical feature for compiling large programs: preserving subterm sharing. As a result, our experiments with compiler construction yielded clear asymptotic slowdown vs. what we eventually accomplished. $\mathcal{R}_{tac}$ is also more heavyweight to use, for instance requiring that theorems be restated manually in a deep embedding to bring them into automation procedures. Furthermore, we are not aware of any past experiments driving verified compilers with $\mathcal{R}_{tac}$.

Aehlig et al. [1] came closest to a fitting approach, using *normalization by evaluation (NbE)* [3] to bootstrap reduction of open terms on top of full reduction, as built into a proof assistant. However, it was simultaneously true that they expanded the proof-assistant trusted code base in ways specific to their technique, and that they did not report any experiments actually using the tool for partial evaluation (just traditional full reduction), potentially hiding performance-scaling challenges or other practical issues. For instance, they also do not preserve subterm sharing explicitly, and they represent variable references as unary natural numbers (de Bruijn-style). They also require that rewrite rules be embodied in ML code, rather than stated as natural "native" lemmas of the proof assistant. We will follow their basic outline with important modifications.

Our implementation builds on fast full reduction in Coq's kernel, via a virtual machine [9] or compilation to native code [5] (neither verified). Especially the latter is similar in adopting NbE for full reduction, simplifying even under λs, on top of a more traditional implementation of OCaml that never executes preemptively under λs. Neither approach unifies support for rewriting with proved rules, and partial evaluation only applies in very limited cases.

A variety of forms of pragmatic partial evaluation have been demonstrated, with Lightweight Modular Staging [16] in Scala as one of the best-known current examples. The LMS-Verify system [2] can be used for formal verification of generated code after-the-fact. Typically LMS-Verify has been used with relatively shallow properties (though potentially applied to larger and more sophisticated code bases than we tackle), not scaling to the kinds of functional-correctness properties that concern us here.

So, overall, to our knowledge, no past compiler as a set of rewrite rules has come with a full proof of correctness as a standalone functional program. Related prior work with mechanized proofs suffered from both performance bottlenecks and usability problems, the latter in requiring that eligible rewrite rules be stated in special deep embeddings.

1.2 Our Solution

Our variant on the technique of Aehlig et al. [1] has these advantages:

- It integrates with a general-purpose, foundational proof assistant, **without growing the trusted code base**.
- For a wide variety of initial functional programs, it provides **fast** partial evaluation with reasonable memory use.
- It allows reduction that **mixes** *rules of the definitional equality* with *equalities proven explicitly as theorems*.
- It allows **rapid iteration** on rewrite rules with *minimal verification overhead*.
- It **preserves sharing** of common subterms.
- It also allows **extraction of standalone compilers**.

Our contributions include answers to a number of challenges that arise in scaling NbE-based partial evaluation in a proof assistant. First, we rework the approach of Aehlig et al. [1] to function *without extending a proof assistant's trusted code base*, which, among other challenges, requires us to prove termination of reduction and encode pattern matching explicitly (leading us to adopt the performance-tuned approach of Maranget [15]). We also improve on Coq-specific related work (e.g., of Malecha and Bengtson [14]) by allowing rewrites to be written in natural Coq form (not special embedded syntax-tree types), while supporting optimizations associated with past unverified engines (e.g., Boespflug [4]).

Second, using partial evaluation to generate residual terms thousands of lines long raises *new scaling challenges*:

- Output terms may contain so *many nested variable binders* that we expect it to be performance-prohibitive to perform bookkeeping operations on first-order-encoded terms (e.g., with de Bruijn indices, as is done in $\mathcal{R}_{tac}$ by Malecha and Bengtson [14]). For instance, while the reported performance experiments of Aehlig et al. [1] generate only closed terms with no binders, Fiat Cryptography may generate a single routine (e.g., multiplication for curve P-384) with nearly a thousand nested binders.
- Naive representation of terms without proper *sharing of common subterms* can lead to fatal term-size blow-up.
- Unconditional rewrite rules are in general insufficient, and we need *rules with side conditions*. E.g., Fiat Cryptography depends on checking lack-of-overflow conditions.
- However, it is also not reasonable to expect a general engine to discharge all side conditions on the spot. We need integration with *abstract interpretation*.

Briefly, our respective solutions to these problems are the *parametric higher-order abstract syntax (PHOAS)* [6] term encoding, a *let-lifting* transformation threaded throughout reduction, extension of rewrite rules with executable Boolean side conditions, and a design pattern that uses decorator function calls to include analysis results in a program.

Finally, we carry out the *first large-scale performance-scaling evaluation* of a verified rewrite-rule-based compiler, covering all elliptic curves from the published Fiat Cryptography experiments, along with microbenchmarks.

We pause to give a motivating example before presenting the core structure of our engine (Section 3), the additional scaling challenges we faced (Section 4), experiments (Section 5), and conclusions. Our implementation is attached.

2 A Motivating Example

Our compilation style involves source programs that mix higher-order functions and inductive types. We want to compile to C code, reducing away uses of fancier features while seizing opportunities for arithmetic simplification. Here is a small but illustrative example.

```
Definition prefixSums (ls : list nat) : list nat :=
  let ls' := combine ls (seq 0 (length ls)) in
  let ls'' := map (λ p, fst p * snd p) ls' in
  let '(_, ls''') := fold_left (λ '(acc, ls''') n,
    let acc' := acc + n in (acc', acc' :: ls''')) ls'' (0, []) in ls'''.
```

This function first computes list `ls'` that pairs each element of input list `ls` with its position, so, for instance, list $[a; b; c]$ becomes $[(a, 0); (b, 1); (c, 2)]$. Then we map over the list of pairs, multiplying the components at each position. Finally, we compute all prefix sums.

We would like to specialize this function to particular list lengths. That is, we know in advance how many list elements we will pass in, but we do not know the values of those elements. For a given length, we can construct a schematic list with one free variable per element. For example, to specialize to length four, we can apply the function to list `[a; b; c; d]`, and we expect this output:

```
let acc := b + c * 2 in let acc' := acc + d * 3 in [acc'; acc; b; 0]
```

We do not quite have C code yet, but, composing this code with another routine to consume the output list, we easily arrive at a form that looks almost like three-address code and is quite easy to translate to C and many other languages.

Notice how subterm sharing via `let`s is important. As list length grows, we avoid quadratic blowup in term size through sharing. Also notice how we simplified the first two multiplications with $a \cdot 0 = 0$ and $b \cdot 1 = b$ (each of which requires explicit proof in Coq), using other arithmetic identities to avoid introducing new variables for the first two prefix sums of `ls''`, as they are themselves constants or variables, after simplification.

To set up our compiler, we prove the algebraic laws that it should use for simplification, starting with basic arithmetic identities.

```
Lemma zero_plus : ∀ n, 0 + n = n.      Lemma times_zero : ∀ n, n * 0 = 0.
Lemma plus_zero : ∀ n, n + 0 = n.      Lemma times_one  : ∀ n, n * 1 = n.
```

Next, we prove a law for each list-related function, connecting it to the primitive-recursion combinator for some inductive type (natural numbers or lists, as appropriate). We also use a further marker `ident.eagerly` to ask the compiler to simplify a case of primitive recursion by complete traversal of the designated argument's constructor tree.

```
Lemma eval_map A B (f : A -> B) l
: map f l = ident.eagerly list_rect _ _ [] (λ x _ l', f x :: l') l.
Lemma eval_fold_left A B (f : A -> B -> A) l a
: fold_left f l a = ident.eagerly list_rect _ _ (λ a, a) (λ x _ r a, r (f a x)) l a.
Lemma eval_combine A B (la : list A) (lb : list B)
: combine la lb =
list_rect _ (λ _, []) (λ x _ r lb, list_case (λ _, _) [] (λ y ys, (x,y)::r ys) lb) la lb.
Lemma eval_length A (ls : list A)
: length ls = list_rect _ 0 (λ _ _ n, S n) ls.
```

With all the lemmas available, we can package them up into a rewriter, which triggers generation of a specialized compiler and its soundness proof. Our Coq plugin introduces a new command `Make` for building rewriters

```
Make rewriter := Rewriter For (zero_plus, plus_zero, times_zero, times_one, eval_map,
  eval_fold_left, do_again eval_length, do_again eval_combine,
  eval_rect nat, eval_rect list, eval_rect prod) (with delta) (with extra idents (seq)).
```

Most inputs to `Rewriter For` list quantified equalities to use for left-to-right rewriting. However, we also use options `do_again`, to request that some rules trigger extra bottom-up passes after being used for rewriting; `eval_rect`, to queue up eager evaluation of a call to a primitive-recursion combinator on a known recursive argument; `with delta`, to request evaluation of all monomorphic operations on concrete inputs; and `with extra idents`, to inform the engine of further permitted identifiers that do not appear directly in any of the rewrite rules.

Our plugin also provides new tactics like `Rewrite_rhs_for`, which applies a rewriter to the right-hand side of an equality goal. That last tactic is just what we need to synthesize a specialized `prefixSums` for list length four, along with its correctness proof.

```
Definition prefixSums4 :
{f:nat→nat→nat→nat→list nat | ∀ a b c d, f a b c d = prefixSums [a;b;c;d]}
  := ltac:(eexists; Rewrite_rhs_for rewriter; reflexivity).
```

That compiler execution ran inside of Coq, but an even more pragmatic approach is to *extract* the compiler as a standalone program in OCaml or Haskell. Such a translation is possible because the `Make` command produces a proved program in Gallina, Coq's logic. As a result, our reworking of Fiat Cryptography compilation culminated in extraction of a command-line OCaml program that developers in industry have been able to run without our help, where Fiat Cryptography previously required installing and running Coq, with an elaborate build process to capture its output. It is also true that the standalone program is about 10× as fast as execution within Coq, though the trusted code base is larger.

3 The Structure of a Rewriter

We are mostly guided by Aehlig et al. [1] but made a number of crucial changes. Let us review the basic idea of the approach of Aehlig et al. First, their supporting library contains:

1. Within the logic of the proof assistant (Isabelle/HOL, in their case), a type of syntax trees for ML programs is defined, with an associated (trusted) operational semantics.
2. They also wrote a reduction function in (deeply embedded) ML, parameterized on a function to choose the next rewrite, and proved it sound once-and-for-all.

Given a set of rewrite rules and a term to simplify, their main tactic must:

1. *Generate a (deeply embedded) ML program that decides which rewrite rule, if any, to apply at the top node of a syntax tree*, along with a proof of its soundness.
2. *Generate a (deeply embedded) ML term standing for the term we set out to simplify*, with a proof that it means the same as the original.
3. Combining the general proof of the rewrite engine with proofs generated by reification (the prior two steps), conclude that an application of the reduction function to the reified rules and term is indeed an ML term that generates correct answers.
4. "Throw the ML term over the wall," using a general code-generation framework for Isabelle/HOL [10]. Trusted code compiles the ML code into the concrete syntax of Standard ML, and compiles it, and runs it, asserting an axiom about the outcome.

Here is where our approach differs at that level of detail:

- Our reduction engine is written *as a normal Gallina functional program*, rather than within a deeply embedded language. As a result, we are able to prove its type-correctness and termination, and we are able to run it within Coq's kernel.
- We do *compile-time specialization of the reduction engine* to sets of rewrite rules, removing overheads of generality.

3.1 Our Approach in Ten Steps

Here is a bit more detail on the steps that go into applying our Coq plugin, many of which we expand on in the following sections. For `Make` to precompute a rewriter:

1. The given lemma statements are scraped for which named identifiers to encode.
2. Inductive types enumerating all available primitive types and functions are emitted. This allows us to achieve the performance gains attributed in Boespflug [4] to having native metalanguage constructors for all constants, without manual coding.
3. Tactics generate all of the necessary definitions and prove all of the necessary lemmas for dealing with this particular set of inductive codes. Definitions include operations like Boolean equality on type codes and lemmas like "all types have decidable equality."
4. The statements of rewrite rules are reified and soundness and syntactic-well-formedness lemmas are proven about each of them.
5. Definitions and lemmas needed to prove correctness are assembled into a single package.

Then, to rewrite in a goal, the following steps are performed:

1. Rearrange the goal into a single quantifier-free logical formula.
2. Reify a selected subterm and replace it with a call to our denotation function.
3. Rewrite with a theorem, into a form calling our rewriter.
4. Call Coq's built-in full reduction (**`vm_compute`**) to reduce this application.
5. Run standard call-by-value reduction to simplify away use of the denotation function.

The object language of our rewriter is nearly simply typed.

$$e ::= \texttt{App}\ e_1\ e_2 \mid \texttt{Let}\ v = e_1\ \texttt{In}\ e_2 \mid \texttt{Abs}\ (\lambda v.\, e) \mid \texttt{Var}\ v \mid \texttt{Ident}\ i$$

The `Ident` case is for identifiers, which are described by an enumeration specific to a use of our library. For example, the identifiers might be codes for $+$, $\cdot$, and literal constants. We write $[\![e]\!]$ for a standard denotational semantics.

3.2 Pattern-Matching Compilation and Evaluation

Aehlig et al. [1] feed a specific set of user-provided rewrite rules to their engine by generating code for an ML function, which takes in deeply embedded term syntax (actually *doubly* deeply embedded, within the syntax of the deeply embedded ML!) and uses ML pattern matching to decide which rule to apply at the top level. Thus, they delegate efficient implementation of pattern matching to the underlying ML implementation. As we instead build our rewriter in Coq's logic, we have no such option to defer to ML.

We could follow a naive strategy of repeatedly matching each subterm against a pattern for every rewrite rule, as in the rewriter of Malecha and Bengtson [14], but in that case we do a lot of duplicate work when rewrite rules use overlapping function symbols. Instead, we adopted the approach of Maranget [15], who describes compilation of pattern matches in OCaml to decision trees that eliminate needless repeated work (for example, decomposing an expression into $x + y + z$ only once even if two different rules match on that pattern).

There are three steps to turn a set of rewrite rules into a functional program that takes in an expression and reduces according to the rules. The first step is pattern-matching compilation: we must compile the left-hand sides of the rewrite rules to a decision tree that describes how and in what order to decompose the expression, as well as describing which rewrite rules to try at which steps of decomposition. Because the decision tree is merely a decomposition hint, we require no proofs about it to ensure soundness of our rewriter. The second step is decision-tree evaluation, during which we decompose the expression as per the decision tree, selecting which rewrite rules to attempt. The only correctness lemma needed for this stage is that any result it returns is equivalent to picking some rewrite rule and rewriting with it. The third and final step is to actually rewrite with the chosen rule. Here the correctness condition is that we must not change the semantics of the expression.

While pattern matching begins with comparing one pattern against one expression, Maranget's approach works with intermediate goals that check multiple patterns against multiple expressions. A decision tree describes how to match a vector (or list) of patterns against a vector of expressions. It is built from these constructors:

- `TryLeaf k onfailure`: Try the k^{th} rewrite rule; if it fails, keep going with `onfailure`.
- `Failure`: Abort; nothing left to try.
- `Switch icases app_case default`: With the first element of the vector, match on its kind; if it is an identifier matching something in `icases`, which is a list of pairs of identifiers and decision trees, remove the first element of the vector and run that decision tree; if it is an application and `app_case` is not `None`, try the `app_case` decision tree, replacing the first element of each vector with the two elements of the function and the argument it is applied to; otherwise, do not modify the vectors and use the `default`.
- `Swap i cont`: Swap the first element of the vector with the i^{th} element (0-indexed) and keep going with `cont`.

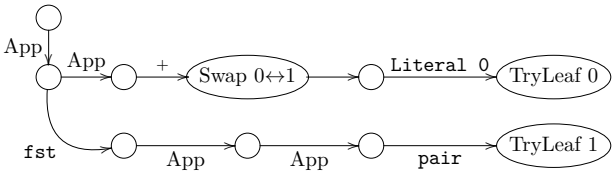
Consider the encoding of two simple example rewrite rules, where we follow Coq's $\mathcal{L}_{tac}$ language in prefacing pattern variables with question marks.

$$?n + 0 \to n \qquad\qquad \texttt{fst}_{\mathbb{Z},\mathbb{Z}}(?x, ?y) \to x$$

We embed them in an AST type for patterns, which largely follows our ASTs for expressions.

```
0. App (App (Ident +) Wildcard) (Ident (Literal 0))
1. App (Ident fst) (App (App (Ident pair) Wildcard) Wildcard)
```

The decision tree produced is



where every nonswap node implicitly has a "default" case arrow to `Failure` and circles represent `Switch` nodes.

We implement, in Coq's logic, an evaluator for these trees against terms. Note that we use Coq's normal partial evaluation to turn our general decision-tree evaluator into a specialized matcher to get reasonable efficiency. Although this partial evaluation of our partial evaluator is subject to the same performance challenges we highlighted in the introduction, it only has to be done once for each set of rewrite rules, and we are targeting cases where the time of per-goal reduction dominates this time of metacompilation.

For our running example of two rules, specializing gives us this match expression.

```
match e with
| App f y => match f with
  | Ident fst => match y with
    | App (App (Ident pair) x) y => x | _ => e end
  | App (Ident +) x => match y with
    | Ident (Literal 0) => x | _ => e end | _ => e end | _ => e end.
```

3.3 Adding Higher-Order Features

Fast rewriting at the top level of a term is the key ingredient for supporting customized algebraic simplification. However, not only do we want to rewrite throughout the structure of a term, but we also want to integrate with simplification of higher-order terms, in a way where we can prove to Coq that our syntax-simplification function always terminates. Normalization by evaluation (NbE) [3] is an elegant technique for adding the latter aspect, in a way where we avoid needing to implement our own λ-term reducer or prove it terminating.

To orient expectations: we would like to enable the following reduction

$$(\lambda f\ x\ y.\ f\ x\ y)\ (+)\ z\ 0 \rightsquigarrow z$$

using the rewrite rule

$$?n + 0 \to n$$

We begin by reviewing NbE's most classic variant, for performing full β-reduction in a simply typed term in a guaranteed-terminating way. Our simply typed λ-calculus syntax is:

$$t ::= t \to t \mid b \qquad\qquad e ::= \lambda v.\, e \mid e\ e \mid v \mid c$$

with v for variables, c for constants, and b for base types.

We can now define normalization by evaluation. First, we choose a "semantic" representation for each syntactic type, which serves as an interpreter's result type.

$$\mathrm{NbE}_t(t_1 \to t_2) = \mathrm{NbE}_t(t_1) \to \mathrm{NbE}_t(t_2) \qquad\qquad \mathrm{NbE}_t(b) = \mathtt{expr}(b)$$

Function types are handled as in a simple denotational semantics, while base types receive the perhaps-counterintuitive treatment that the result of "executing" one is a syntactic expression of the same type. We write $\mathtt{expr}(b)$ for the metalanguage type of object-language syntax trees of type b, relying on a type family `expr`.

Now the core of NbE, shown in Figure 1, is a pair of dual functions reify and reflect, for converting back and forth between syntax and semantics of the object language, defined by primitive recursion on type syntax. We split out analysis of term syntax in a separate function reduce, defined by primitive recursion on term syntax, when usually this functionality would be mixed in with reflect. The reason for this choice will become clear when we extend NbE.

We write v for object-language variables and x for metalanguage (Coq) variables, and we overload λ notation using the metavariable kind to signal whether we are building a host λ or a λ syntax tree for the embedded language. The crucial first clause for reduce replaces object-language variable v with fresh metalanguage variable x, and then we are somehow tracking that all free variables in an argument to reduce must have been replaced with metalanguage variables by the time we reach them. We reveal in Subsection 4.1 the encoding decisions that make all the above legitimate, but first let us see how to integrate

$$\begin{aligned}
\text{reify}_t &: \text{NbE}_t(t) \to \text{expr}(t) \\
\text{reify}_{t_1 \to t_2}(f) &= \lambda v.\, \text{reify}_{t_2}(f(\text{reflect}_{t_1}(v))) \\
\text{reify}_b(f) &= f \\
\\
\text{reflect}_t &: \text{expr}(t) \to \text{NbE}_t(t) \\
\text{reflect}_{t_1 \to t_2}(e) &= \lambda x.\, \text{reflect}_{t_2}(e(\text{reify}_{t_1}(x)) \\
\text{reflect}_b(e) &= e
\end{aligned}$$

$$\begin{aligned}
\text{reduce} &: \text{expr}(t) \to \text{NbE}_t(t) \\
\text{reduce}(\lambda v.\ e) &= \lambda x.\ \text{reduce}([x/v]e) \\
\text{reduce}(e_1\ e_2) &= (\text{reduce}(e_1))\,(\text{reduce}(e_2)) \\
\text{reduce}(x) &= x \\
\text{reduce}(c) &= \text{reflect}(c) \\
\\
\text{NbE} &: \text{expr}(t) \to \text{expr}(t) \\
\text{NbE}(e) &= \text{reify}(\text{reduce}(e))
\end{aligned}$$

Figure 1 Implementation of normalization by evaluation.

use of the rewriting operation from the previous section. To fuse NbE with rewriting, we only modify the constant case of `reduce`. First, we bind our specialized decision-tree engine (which rewrites *at the root of an AST only*) under the name rewrite-head.

In the constant case, we still reflect the constant, but underneath the binders introduced by full η-expansion, we perform one instance of rewriting. In other words, we change this one function-definition clause:

$$\text{reflect}_b(e) = \text{rewrite-head}(e)$$

It is important to note that a constant of function type will be η-expanded only once for each syntactic occurrence in the starting term, though the expanded function is effectively a thunk, waiting to perform rewriting again each time it is called. From first principles, it is not clear why such a strategy terminates on all possible input terms.

The details so far are essentially the same as in the approach of Aehlig et al. [1]. Recall that their rewriter was implemented in a deeply embedded ML, while ours is implemented in Coq's logic, which enforces termination of all functions. Aehlig et al. did not prove termination, which indeed does not hold for their rewriter in general, which works with untyped terms, not to mention the possibility of divergent rule-specific ML functions. In contrast, we need to convince Coq up-front that our interleaved λ-term normalization and algebraic simplification always terminate. Additionally, we must prove that rewriting preserves term denotations, which can easily devolve into tedious binder bookkeeping.

The next section introduces the techniques we use to avoid explicit termination proof or binder bookkeeping, in the context of a more general analysis of scaling challenges.

4 Scaling Challenges

Aehlig et al. [1] only evaluated their implementation against closed programs. What happens when we try to apply the approach to partial-evaluation problems that should generate thousands of lines of low-level code?

4.1 Variable Environments Will Be Large

We should think carefully about representation of ASTs, since many primitive operations on variables will run in the course of a single partial evaluation. For instance, Aehlig et al. [1] reported a significant performance improvement changing variable nodes from using strings to using de Bruijn indices [7]. However, de Bruijn indices and other first-order representations remain painful to work with. We often need to fix up indices in a term being

substituted in a new context. Even looking up a variable in an environment tends to incur linear time overhead, thanks to traversal of a list. Perhaps we can do better with some kind of balanced-tree data structure, but there is a fundamental performance gap versus the arrays that can be used in imperative implementations. Unfortunately, it is difficult to integrate arrays soundly in a logic. Also, even ignoring performance overheads, tedious binder bookkeeping complicates proofs.

Our strategy is to use a variable encoding that pushes all first-order bookkeeping off on Coq's kernel or the implementation of the language we extract to, which are themselves performance-tuned with some crucial pieces of imperative code. Parametric higher-order abstract syntax (PHOAS) [6] is a dependently typed encoding of syntax where binders are managed by the enclosing type system. It allows for relatively easy implementation and proof for NbE, so we adopted it for our framework.

Here is the actual inductive definition of term syntax for our object language, PHOAS-style. The characteristic oddity is that the core syntax type `expr` is parameterized on a dependent type family for representing variables. However, the final representation type `Expr` uses first-class polymorphism over choices of variable type, bootstrapping on the metalanguage's parametricity to ensure that a syntax tree is agnostic to variable type.

```
Inductive type := arrow (s d : type) | base (b : base_type).
Infix "→" := arrow.
Inductive expr (var : type -> Type) : type -> Type :=
| Var {t} (v : var t) : expr var t
| Abs {s d} (f : var s -> expr var d) : expr var (s → d)
| App {s d} (f : expr var (s → d)) (x : expr var s) : expr var d
| LetIn {a b} (x : expr var a) (f : var a -> expr var b) : expr var b
| Const {t} (c : const t) : expr var t.
Definition Expr (t : type) : Type := forall var, expr var t.
```

A good example of encoding adequacy is assigning a simple denotational semantics. First, a simple recursive function assigns meanings to types.

```
Fixpoint denoteT (t : type) : Type := match t with
  | arrow s d => denoteT s -> denoteT d
  | base b    => denote_base_type b  end.
```

Next we see the convenience of being able to *use* an expression by choosing how it should represent variables. Specifically, it is natural to choose *the type-denotation function itself* as variable representation. Especially note how this choice makes rigorous last section's convention (e.g., in the suspicious function-abstraction clause of reduce), where a recursive function enforces that values have always been substituted for variables early enough.

```
Fixpoint denoteE {t} (e : expr denoteT t) : denoteT t := match e with
  | Var v     => v
  | Abs f     => λ x, denoteE (f x)
  | App f x   => (denoteE f) (denoteE x)
  | LetIn x f => let xv := denoteE x in denoteE f xv
  | Ident c   => denoteI c  end.
Definition DenoteE {t} (E : Expr t) : denoteT t := denoteE (E denoteT).
```

It is now easy to follow the same script in making our rewriting-enabled NbE fully formal, in Figure 2. Note especially the first clause of `reduce`, where we avoid variable substitution precisely because we have chosen to represent variables with normalized semantic values. The subtlety there is that base-type semantic values are themselves expression syntax trees, which depend on a nested choice of variable representation, which we retain as a parameter throughout these recursive functions. The final definition λ-quantifies over that choice.

```
Fixpoint nbeT var (t : type) : Type :=
match t with
| arrow s d => nbeT var s -> nbeT var d
| base b    => expr var b
end.


Fixpoint reify {var t}
  : nbeT var t -> expr var t :=
match t with
| arrow s d => λ f, Abs (λ x,
    reify (f (reflect (Var x))))
| base b    => λ e, e            end
```

```
with reflect{var t}:expr var t->nbeT var t
  := match t with
| arrow s d => λ e, λ x,
  reflect (App e (reify x))
| base b    => rewrite_head      end.
Fixpoint reduce{var t}(e:expr (nbeT var) t)
  : nbeT var t := match e with
| Abs e     => λ x, reduce (e (Var x))
| App e1 e2 => (reduce e1) (reduce e2)
| Var x     => x
| Ident c   => reflect (Ident c) end.
Definition Rewrite {t} (E:Expr t) : Expr t
  := λ var, reify (reduce (E (nbeT var t))).
```

Figure 2 PHOAS implementation of normalization by evaluation.

One subtlety hidden in Figure 2 in implicit arguments is in the final clause of `reduce`, where the two applications of the `Ident` constructor use different variable representations. With all those details hashed out, we can prove a pleasingly simple correctness theorem, with a lemma for each main definition, with inductive structure mirroring recursive structure of the definition, also appealing to correctness of last section's pattern-compilation operations. (We now use syntax $[\![\cdot]\!]$ for calls to `DenoteE`.)

$$\forall t, E : \texttt{Expr t}.\ [\![\texttt{Rewrite}(E)]\!] = [\![E]\!]$$

To understand how we now apply the soundness theorem in a tactic, it is important to note how the Coq kernel builds in reduction strategies. These strategies have, to an extent, been tuned to work well to show equivalence between a simple denotational-semantics application and the semantic value it produces. In contrast, it is rather difficult to code up one reduction strategy that works well for all partial-evaluation tasks. Therefore, we should restrict ourselves to (1) running full reduction in the style of functional-language interpreters and (2) running normal reduction on "known-good" goals like correctness of evaluation of a denotational semantics on a concrete input.

Operationally, then, we apply our tactic in a goal containing a term e that we want to partially evaluate. In standard proof-by-reflection style, we *reify* e into some E where $[\![E]\!] = e$, replacing e accordingly, asking Coq's kernel to validate the equivalence via standard reduction. Now we use the `Rewrite` correctness theorem to replace $[\![E]\!]$ with $[\![\texttt{Rewrite}(E)]\!]$. Next we ask the Coq kernel to simplify `Rewrite`(E) by *full reduction via native compilation*. Finally, where E' is the result of that reduction, we simplify $[\![E']\!]$ with standard reduction.

We have been discussing representation of bound variables. Also important is representation of constants (e.g., library functions mentioned in rewrite rules). They could also be given some explicit first-order encoding, but dispatching on, say, strings or numbers for constants would be rather inefficient in our generated code. Instead, we chose to have our Coq plugin generate a custom inductive type of constant codes, for each rewriter that we ask it to build with `Make`. As a result, dispatching on a constant can happen in constant time, based on whatever pattern-matching is built into the execution language (either the Coq kernel or the target language of extraction). To our knowledge, no past verified reduction tool in a proof assistant has employed that optimization.

4.2 Subterm Sharing Is Crucial

For some large-scale partial-evaluation problems, it is important to represent output programs with sharing of common subterms. Redundantly inlining shared subterms can lead to exponential increase in space requirements. Consider the Fiat Cryptography [8] example

of generating a 64-bit implementation of field arithmetic for the P-256 elliptic curve. The library has been converted manually to continuation-passing style, allowing proper generation of `let` binders, whose variables are often mentioned multiple times. We ran that old code generator (actually just a subset of its functionality, but optimized by us a bit further, as explained in Subsection 5.3) on the P-256 example and found it took about 15 seconds to finish. Then we modified reduction to inline `let` binders instead of preserving them, at which point the job terminated with an out-of-memory error, on a machine with 64 GB of RAM.

We see a tension here between performance and niceness of library implementation. When we built the original Fiat Cryptography, we found it necessary to CPS-convert the code to coax Coq into adequate reduction performance. Then all of our correctness theorems were complicated by reasoning about continuations. In fact, the CPS reasoning was so painful that at one point most algorithms in the template library were defined twice, once in continuation-passing style and once in direct-style code, because it was easier to prove the two equivalent and work with the non-CPS version than to reason about the CPS version directly. It feels like a slippery slope on the path to implementing a domain-specific compiler, rather than taking advantage of the pleasing simplicity of partial evaluation on natural functional programs. Our reduction engine takes shared-subterm preservation seriously while applying to libraries in direct style.

Our approach is `let`-lifting: we lift `let`s to top level, so that applications of functions to `let`s are available for rewriting. For example, we can perform the rewriting

$$\texttt{map}\ (\lambda x.\, y + x)\ (\texttt{let}\ z := e\ \texttt{in}\ [0; 1; z + 1]) \rightsquigarrow \texttt{let}\ z := e\ \texttt{in}\ [y; y + 1; y + (z + 1)]$$

using the rules

$$\texttt{map}\ ?f\ [] \to []\qquad \texttt{map}\ ?f\ (?x :: ?xs) \to f\ x :: \texttt{map}\ f\ xs\qquad ?n + 0 \to n$$

We define a telescope-style type family called `UnderLets`:

```
Inductive UnderLets {var} (T : Type) := Base (v : T)
| UnderLet {A} (e : @expr var A) (f : var A -> UnderLets T).
```

A value of type `UnderLets T` is a series of `let` binders (where each expression `e` may mention earlier-bound variables) ending in a value of type `T`.

Recall that the NbE type interpretation mapped base types to expression syntax trees. We add flexibility, parameterizing by a Boolean declaring whether to introduce telescopes.

```
Fixpoint nbeT' {var} (with_lets : bool) (t : type) := match t with
  | base t => if with_lets then @UnderLets var (@expr var t) else @expr var t
  | arrow s d => nbeT' false s -> nbeT' true d  end.
Definition nbeT := nbeT' false.      Definition nbeT_with_lets := nbeT' true.
```

There are cases where naive preservation of `let` binders blocks later rewrites from triggering and leads to suboptimal performance, so we include some heuristics. For instance, when the expression being bound is a constant, we always inline. When the expression being bound is a series of list "cons" operations, we introduce a name for each individual list element, since such a list might be traversed multiple times in different ways.

4.3 Rules Need Side Conditions

Many useful algebraic simplifications require side conditions. For example, bit-shifting operations are faster than divisions, so we might want a rule such as

$$?n/?m \to n \gg \log_2 m \quad \text{if} \quad 2^{\lfloor \log_2 m \rfloor} = m$$

The trouble is how to support predictable solving of side conditions during partial evaluation, where we may be rewriting in open terms. We decided to sidestep this problem by allowing side conditions only as executable Boolean functions, to be applied only to variables that are confirmed as *compile-time constants*, unlike Malecha and Bengtson [14] who support general unification variables. We added a variant of pattern variable that only matches constants. Semantically, this variable style has no additional meaning, and in fact we implement it as a special identity function (notated as an apostrophe) that should be called in the right places within Coq lemma statements. Rather, use of this identity function triggers the right behavior in our tactic code that reifies lemma statements.

Our reification inspects the hypotheses of lemma statements, using type classes to find decidable realizations of the predicates that are used, thereby synthesizing one Boolean expression of our deeply embedded term language, which stands for a decision procedure for the hypotheses. The `Make` command fails if any such expression contains pattern variables not marked as constants.

Hence, we encode the above rule as $\forall n, m.\ 2^{\lfloor \log_2({'m}) \rfloor} = {'m} \to n / {'m} = n \gg {'(\log_2 m)}$.

4.4 Side Conditions Need Abstract Interpretation

With our limitation that side conditions are decided by executable Boolean procedures, we cannot yet handle directly some of the rewrites needed for realistic compilation. For instance, Fiat Cryptography reduces high-level functional to low-level code that only uses integer types available on the target hardware. The starting library code works with arbitrary-precision integers, while the generated low-level code should be careful to avoid unintended integer overflow. As a result, the setup may be too naive for our running example rule $?n + 0 \to n$. When we get to reducing fixed-precision-integer terms, we must be legalistic:

$$\texttt{add\_with\_carry}_{64}(?n, 0) \to (0, n) \quad \text{if} \quad 0 \le n < 2^{64}$$

We developed a design pattern to handle this kind of rule.

First, we introduce a family of functions $\texttt{clip}_{l,u}$, each of which forces its integer argument to respect lower bound l and upper bound u. Partial evaluation is proved with respect to unknown realizations of these functions, only requiring that $\texttt{clip}_{l,u}(n) = n$ when $l \le n < u$. Now, before we begin partial evaluation, we can run a verified abstract interpreter to find conservative bounds for each program variable. When bounds l and u are found for variable x, it is sound to replace x with $\texttt{clip}_{l,u}(x)$. Therefore, at the end of this phase, we assume all variable occurrences have been rewritten in this manner to record their proved bounds.

Second, we proceed with our example rule refactored:

$$\texttt{add\_with\_carry}_{64}(\texttt{clip}_{,?l,?u}(?n), 0) \to (0, \texttt{clip}_{l,u}(n)) \quad \text{if} \quad u < 2^{64}$$

If the abstract interpreter did its job, then all lower and upper bounds are constants, and we can execute side conditions straightforwardly during pattern matching.

See Appendix F in the full version for discussion of some further twists in the implementation.

5 Evaluation

Our implementation, available on GitHub at `mit-plv/rewriter@ITP-2022-perf-data` and with a roadmap in Appendix G of the full version, includes a mix of Coq code for the proved core of rewriting, tactic code for setting up proper use of that core, and OCaml plugin code

for the manipulations beyond the tactic language's current capabilities. We report here on evidence that the tool is effective, first in terms of productivity by users and then in terms of compile-time performance.

5.1 Iteration on the Fiat Cryptography Compiler

We ported Fiat Cryptography's core compiler functionality to use our framework. The result is now used in production by a number of open-source projects. We were glad to retire the CPS versions of verified arithmetic functions, which had been present only to support predictable reduction with subterm sharing. More importantly, it became easy to experiment with new transformations via proving new rewrite theorems, directly in normal Coq syntax, including the following, all justified by demand from real users:

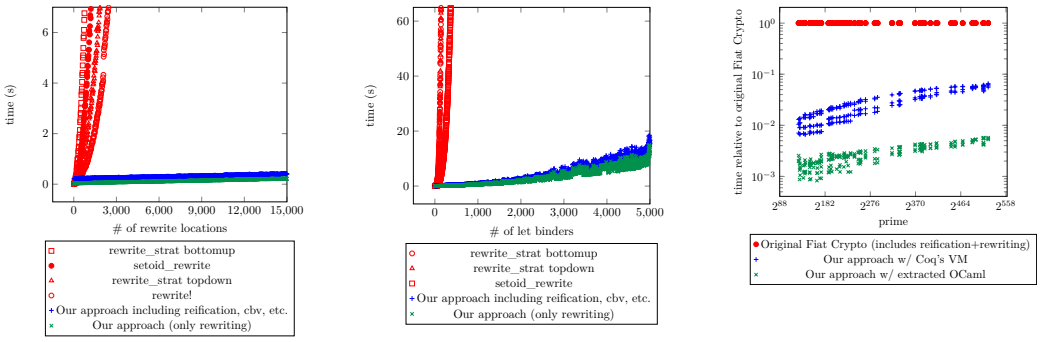
- Reassociating arithmetic to minimize the bitwidths of intermediate results
- Multiplication primitives that separately return high halves and low halves
- Strings and a "comment" function of type $\forall A.\ \texttt{string} \to A \to A$
- Support for bitwise exclusive-or
- A special marker to block C compilers from introducing conditional jumps in code that should be constant-time
- Eliding bitmask-with-constant operations that can be proved as no-ops
- Rules to introduce conditional moves (on supported platforms)
- New hardware backend, via rules that invoke special instructions of a cryptographic accelerator
- New hardware backend, with a requirement that all intermediate integers have the same bitwidth, via rules to break wider operations down into several narrower operations

5.2 Microbenchmarks

Now we turn to evaluating performance of generated compilers. We start with microbenchmarks focusing attention on particular aspects of reduction and rewriting, with Appendix C of the full version going into more detail, including on a few more benchmarks.

Our first example family, *nested binders*, has two integer parameters n and m. An expression tree is built with 2^n copies of an expression, which is itself a free variable with m "useless" additions of zero. We want to see all copies of this expression reduced to just the variable. Figure 3a on the following page shows the results for $n = 3$ as we scale m. The comparison points are Coq's `rewrite!`, `setoid_rewrite`, and `rewrite_strat`. The first two perform one rewrite at a time, taking minimal advantage of commonalities across them and thus generating quite large, redundant proof terms. The third makes top-down or bottom-up passes with combined generation of proof terms. For our own approach, we list both the total time and the time taken for core execution of a verified rewrite engine, without counting reification (converting goals to ASTs) or its inverse (interpreting results back to normal-looking goals). The comparison here is very favorable for our approach so long as $m > 2$. (See Appendix B.1 in the full version for more detailed plots.)

Now consider what happens when we use `let` binders to share subterms within repeated addition of zero, incorporating exponentially many additions with linearly sized terms. Figure 3b on the next page shows the results. The comparison here is again very favorable for our approach. The competing tactics spike upward toward timeouts at just a few hundred generated binders, while our engine is only taking about 10 seconds for examples with 5,000 nested binders.



(a) No binders. **(b)** Nested binders. **(c)** Fiat Cryptography.

Figure 3 Timing of different partial-evaluation implementations.

Although we have made our comparison against the built-in tactics `setoid_rewrite` and `rewrite_strat`, by analyzing the performance in detail, we can argue that these performance bottlenecks are likely to hold for any proof assistant designed like Coq. Detailed debugging reveals five performance bottlenecks in the existing tactics, discussed in Appendix A of the full version.

5.3 Macrobenchmark: Fiat Cryptography

Finally, we consider an experiment (described in more detail in Appendix B.2 of the full version) replicating the generation of performance-competitive finite-field-arithmetic code for all popular elliptic curves by Erbsen et al. [8]. In all cases, we generate essentially the same code as they did, so we only measure performance of the code-generation process. We stage partial evaluation with three different reduction engines (i.e., three `Make` invocations), respectively applying 85, 56, and 44 rewrite rules (with only 2 rules shared across engines), taking total time of about 5 minutes to generate all three engines. These engines support 95 distinct function symbols.

Figure 3c graphs running time of three different partial-evaluation and rewriting methods for Fiat Cryptography, as the prime modulus of arithmetic scales up. Times are normalized to the performance of the original method of Erbsen et al. [8], which relied on standard Coq reduction to evaluate code that had been manually written in CPS, followed by reification and a custom ad-hoc simplification and rewriting engine.

As the figure shows, our approach gives about a 10×–1000× speed-up over the original Fiat Cryptography pipeline. Inspection of the timing profiles of the original pipeline reveals that reification dominates the timing profile; since partial evaluation is performed by Coq's kernel, reification must happen *after* partial evaluation, and hence the size of the term being reified grows with the size of the output code. Also recall that the old approach required rewriting Fiat Cryptography's library of arithmetic functions in continuation-passing style, enduring this complexity in library correctness proofs, while our new approach applies to a direct-style library. Finally, the old approach included a custom reflection-based arithmetic simplifier for term syntax, run after traditional reduction, whereas now we are able to apply a generic engine that combines both, without requiring more than proving traditional rewrites.

The figure also confirms a clear performance advantage of running reduction in code extracted to OCaml, which is possible because our plugin produces verified code in Coq's functional language. The extracted version is about 10× faster than running in Coq's kernel.

6 Future Work

By far the biggest next step for our engine is to integrate abstract interpretation with rewriting and partial evaluation. We expect this would net us asymptotic performance gains as described in Appendix D of the full version. Additionally, it would allow us to simplify the phrasing of many of our post-abstract-interpretation rewrite rules, by relegating bounds information to side conditions rather than requiring that they appear in the syntactic form of the rule.

There are also a number of natural extensions to our engine. For instance, we do not yet allow pattern variables marked as "constants only" to apply to container datatypes; we limit the mixing of higher-order and polymorphic types, as well as limiting use of first-class polymorphism; we do not support rewriting with equalities of nonfully-applied functions; we only support decidable predicates as rule side conditions, and the predicates may only mention pattern variables restricted to matching constants; we have hardcoded support for a small set of container types and their eliminators; we support rewriting with equality and no other relations; and we require decidable equality for all types mentioned in rules.

References

1 Klaus Aehlig, Florian Haftmann, and Tobias Nipkow. A compiled implementation of normalization by evaluation. In *Proc. TPHOLs*, 2008.

2 Nada Amin and Tiark Rompf. LMS-Verify: Abstraction without regret for verified systems programming. In *Proc. POPL*, 2017.

3 U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, July 1991. doi:10.1109/LICS.1991.151645.

4 Mathieu Boespflug. Efficient normalization by evaluation. In Olivier Danvy, editor, *Workshop on Normalization by Evaluation*, Los Angeles, United States, August 2009. URL: https://hal.inria.fr/inria-00434283.

5 Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full reduction at full throttle. In *Proc. CPP*, 2011.

6 Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ICFP'08: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, Victoria, British Columbia, Canada, September 2008. URL: http://adam.chlipala.net/papers/PhoasICFP08/.

7 Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

8 Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic – with proofs, without compromises. In *IEEE Security & Privacy*, San Francisco, CA, USA, May 2019. URL: http://adam.chlipala.net/papers/FiatCryptoSP19/.

9 Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *Proc. ICFP*, 2002.

10 Florian Haftmann and Tobias Nipkow. A code generator framework for Isabelle/HOL. In *Proc. TPHOLs*, 2007.

11 Jason Hickey and Aleksey Nogin. Formal compiler construction in a logical framework. *Higher-Order and Symbolic Computation*, 19(2):197–230, 2006. doi:10.1007/s10990-006-8746-6.

12 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *POPL '14: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 179–191. ACM Press, January 2014. URL: https://cakeml.org/popl14.pdf.

13 Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009. URL: http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf.

14 Gregory Malecha and Jesper Bengtson. *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings*, chapter Extensible and Efficient Automation Through Reflective Tactics, pages 532–559. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. doi:10.1007/978-3-662-49498-1_21.

15 Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 35–46. ACM, 2008. URL: http://moscova.inria.fr/~maranget/papers/ml05e-maranget.pdf.

16 Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *Proceedings of GPCE*, 2010. URL: https://infoscience.epfl.ch/record/150347/files/gpce63-rompf.pdf.

17 Zachary Tatlock and Sorin Lerner. Bringing extensibility to verified compilers. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 111–121, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1806596.1806611.

Automatic Test-Case Reduction in Proof Assistants: A Case Study in Coq

Jason Gross
CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA
Machine Intelligence Research Institute, Berkeley, CA, USA

Théo Zimmermann
Inria, Université Paris Cité, CNRS, IRIF, F-75013, Paris, France

Miraya Poddar-Agrawal
Reed College, Portland, OR, USA

Adam Chlipala
CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

Abstract

As the adoption of proof assistants increases, there is a need for efficiency in identifying, documenting, and fixing compatibility issues that arise from proof-assistant evolution. We present the Coq Bug Minimizer, a tool for *reproducing buggy behavior* with *minimal* and *standalone* files, integrated with coqbot to trigger *automatically* on failures from Coq's reverse dependency compatibility testing. Our tool eliminates the overhead of having to download, set up, compile, and then explore and understand large developments, enabling Coq developers to easily obtain modular test-case files for fast experimentation. In this paper, we describe insights about how test-case reduction is different in Coq than in traditional compilers. We expect that our insights will generalize to other proof assistants. We evaluate the Coq Bug Minimizer on over 150 compatibility testing failures. Our tool succeeds in reducing failures to smaller test cases roughly 75% of the time. The minimizer produces a fully standalone test case 89% of the time, and it is on average about one-third the size of the original test. The average reduced test case compiles in 1.25 seconds, with 75% taking under half a second.

2012 ACM Subject Classification Software and its engineering → Software evolution; Software and its engineering → Maintaining software; Software and its engineering → Compilers; Software and its engineering → Formal software verification

Keywords and phrases debugging, automatic test-case reduction, Coq, bug minimizer

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.18

Related Version *Earlier Version*: `https://jasongross.github.io/papers/2015-coq-bug-minimizer.pdf` [6]

Supplementary Material
Dataset (Data and Data Analysis Script): `https://doi.org/10.6084/m9.figshare.19141952.v1`
Software (Source Code): `https://github.com/JasonGross/coq-tools`
archived at `swh:1:dir:eb2345b367437d8b36b80b18a15059673681b22b`

Funding This work was supported in part by a Google Research Award, National Science Foundation grants CCF-1253229, CCF-1512611, and CCF-1521584, and the National Science Foundation Graduate Research Fellowship under Grant Nos. 1122374 and 1745302. Any opinion, 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

In the world of machine verification, the dream is to prove the correctness of every program. Projects such as Coq Coq Correct! [13] make significant progress towards this dream for even our most foundational tools: proof assistants themselves. However, large swathes of proof-assistant software – such as tactic languages, elaboration hints, and document managers – remain unproven, lacking even adequate test-suite coverage.

As a solution to expanding the test-suite coverage for the Coq proof assistant, developers adopted "reverse dependency compatibility testing" (RDCT) [10, 18], wherein changes in Coq are tested in continuous integration (CI) against a crowdsourced suite of external Coq projects maintained by different teams in different repositories. In this manner, user-centric concerns are well-addressed. To prevent the crowdsourced test suite from shrinking, when Coq evolves in a desired direction but breaks some external project in the process, developers of Coq will fix the compatibility issue in the external project. *We believe that to facilitate the use of proof assistants in industry-scale projects, it is essential to make it easy to find, understand, and fix compatibility issues as a proof assistant continues to evolve.*

Since the external projects in the Coq test suite are large and intricate, debugging and fixing failures reported by RDCT is a time- and effort-intensive process for developers. They must perform many steps before beginning to understand and work on the bug. First, developers will endure the tedium of downloading, setting up, and compiling the external project. Then, they may have to take on the daunting task of figuring out the larger project context, which is not even directly relevant to the bug.

The current debugging process can be significantly optimized for developer experience. Additionally, the current process does not easily yield test cases to add to Coq's internal test suite. Instead the test cases remain buried in external developments, whereas we would like to bring bugs to the center. In order to improve the debugging process, we built the Coq Bug Minimizer[1] which **reproduces buggy behavior** in **minimal** and **standalone** files. Typically, minimized files reduce the total number of lines of code involved in exhibiting buggy behavior by about a factor of three, making it significantly easier for developers to observe, play with, understand, and fix bugs. Furthermore, we have integrated the Coq Bug Minimizer with coqbot [19] to trigger **automatically** on RDCT failures, reducing the friction of building minimized files.

Test-case reduction already has a rich literature [3]. However, it is focused mostly on traditional languages such as C, and even generic reduction techniques may not apply so well to proof assistants. In this paper, we share what we have learned about where test-case reduction is harder and where it is easier in Coq than in traditional compilers, and we describe how we got around the difficulties. Drawing on empirical results from nearly a year of use in Coq's production CI system, we reflect on how effective our style of test-case reduction has been and where the biggest opportunities for improvement remain. We believe that our methods may be of interest for developers of other proof assistants who are also facing a tradeoff between enabling evolution and preserving stability, in a context of industrial use.

The structure of the paper is as follows. Section 2 introduces a constructed example of test-case reduction in Coq, articulating desiderata for test-case reduction in the proof-assistant setting. Section 3 details aspects of traditional-setting test-case reduction that are simpler or irrelevant in Coq and, we expect, in other proof assistants. Then Sections 4, 5, 6, and 7 explore the four desiderata and describe the details of our solution to the more

[1] Available on GitHub in `JasonGross/coq-tools`

important challenges of each; while we expect that most of these details will be specific to Coq, they highlight which aspects of proof-assistant design are relevant to test-case reduction in a way that we expect will generalize to other proof assistants. Section 8 forays into the applicability of the Coq Bug Minimizer for bug-reporter workflow as a secondary use case. Finally, Section 9 presents our deployment in Coq's production CI, with analysis of how effectively different test cases were minimized; Section 10 describes connections to related work; and Section 11 discusses our thoughts on the most worthwhile improvements to make to our tooling.

2 Desiderata

Let us begin with a constructed example of minimizing an RDCT failure. Our objective is to explore the space of file modifications that will aid human understanding of the bug. Consider the following Coq source file:

```
Require Import UsefulTactics.
Definition zero := 0. Definition one := 1.
Definition two := 2. Definition three := 3.
Lemma foo : forall x, x = zero -> S x = one.
Proof. crush. Qed.
```

Suppose the `crush` tactic triggered a new bug in Coq. The most obvious move is to find deletable sentences and delete them, producing a smaller file:

```
Require Import UsefulTactics.
Definition zero := 0. Definition one := 1.
Lemma foo : forall x, x = zero -> S x = one.
Proof. crush.
```

The file still depends on an imported module not native to the Coq standard library. The next move is to inline this dependency, producing a standalone file:

```
Module UsefulTactics.
Ltac head expr := match expr with | ?f _ => head f | _ => expr end.
Ltac head_hnf expr := let expr' := eval hnf in expr in head expr'.
Ltac crush := intros; subst; try reflexivity.
End UsefulTactics.
Import UsefulTactics.
Definition zero := 0. Definition one := 1.
Lemma foo : forall x, x = zero -> S x = one.
Proof. crush.
```

Now we may look for any more opportunities to delete lines, producing a standalone, reduced file:

```
Ltac crush := intros; subst; try reflexivity.
Definition zero := 0. Definition one := 1.
Lemma foo : forall x, x = zero -> S x = one.
Proof. crush.
```

From the above process we can extrapolate desiderata for the Coq Bug Minimizer.

1. **Reproducing buggy behavior,** deciding when two source files indicate the same bug. Many reasonable file simplifications lead to incidental changes in error messages. The Coq Bug Minimizer must trade off between preserving specific details of error messages and aiding human understanding of the underlying bug.
2. **Minimal files,** exploring the space of program simplifications in a smart way with respect to constraints of the proof-assistant setting. Many research papers in the software-engineering community have been written on just this topic [5, 17, 2, 14, 16], but constraints in a proof-assistant setting are uncommon in conventional programming. For instance, highly automated Coq developments often have long compile times even for single files, so we may need to be more frugal in how many program variants we test.
3. **Standalone files,** creating standalone files that illuminate new test cases and can be added to Coq's internal test suite. This is difficult in dependently typed languages with metaprogramming facilities such as Coq. For instance, eliminating needless dependencies in simply typed languages may be trivial, but dependently typed languages eliminate the distinction between runtime and compile time, resulting in tight coupling between files.
4. **Smooth developer experience,** automatically finding which file triggered a bug, with which compilation settings, including path information to find dependencies. The Coq Bug Minimizer must work with the wide variety of build systems used in different Coq libraries.

Achieving each desideratum posed interesting challenges and required making several design choices. Before proceeding to share solutions to these challenges, we note the ways in which test-case reduction is *simpler* in the proof-assistant setting than in other settings.

3 Simplifications of the Proof-Assistant Setting

Classic delta debugging [16] is a technique in test-case reduction for traditional compilers. It employs binary search through program structure to find subprograms that can be removed while preserving properties relevant to triggering specific bugs for the chosen compiler. Coq's lack of forward references permits a simpler method: first remove everything after the error-message-generating line, and then try removing the syntactic units beforehand in-order, one-at-a-time. Unlike in languages from Java to Haskell, where all functions in a file are considered mutually recursive, in Coq there should be no way for one error-message-generating line of a file to change behavior based on modifications to later lines. In this manner we reduce the number of "experiments" on program variants, which is especially useful when each program variant requires significant processing time, as is often the case in Coq.

Our empirical evaluation (Section 9) demonstrates that this strategy is adequately performant. We conjecture that the reason for this adequate performance is that dependency trees of Coq theorems and proofs tend to be relatively deep compared to the number of definitions and theorems in any single file. This hypothesis is borne out by the fact that our typical "minimal" test case tends to be only about a third the size of the total amount of code in all files in the dependency tree of the initial test case. If instead there were orders of magnitude more useless lines than true dependent lines, we expect that a binary-search strategy would be required for adequate performance.

4 Reproducing Buggy Behavior

How do we know modifications to source files are genuine simplifications that have not masked bugs? What does it mean to reproduce the "same" bug? We generate a file that succeeds on the previous version of Coq and continues to fail on the modified version of Coq, with the same error message that showed up in the RDCT failure. However, the error

message of the generated file does not need to be *exactly* the same as in the original file, so long as the reason for the error message is the same. Thus, we modify our goal to reproducing buggy behavior in place of reproducing the "same" bug.

We apply the following relaxations in comparing error messages. While these particular relaxations are specific to Coq, we expect that other proof assistants will have a similarly limited set of relaxations.

1. Universe inconsistencies are how Coq prevents users from proving absurdity by assuming a "set of all sets." The explanations of universe inconsistencies in error messages are sensitive to how many universes are floating around and in what order constraints were added. Rather than requiring output files to mimic the error messages exactly, we only require that they result in *some* universe inconsistency.
2. Any two error messages about "forgotten universes" are considered matching, since these tend to arise only from very specific Coq internal errors.
3. Usually differences in numbering, e.g. in universes or autogenerated identifiers, are incidental and are not treated as implying different error messages. One special case is lengths of universe instances, so we look for the text "Universe instance should have length" in the error message and only use number-insensitive comparison if this text is not found.
4. We consider any error messages containing "Unsatisfied constraints: ... (maybe a bugged tactic)" as equivalent, since related bugs are localized to one relatively small part of the Coq implementation, and small changes to a source file can modify constraints significantly.
5. We also ignore filenames, line references, and word wrapping in comparing error messages.

5 Minimal Files

Test-case reduction is powerful in making long source files more comprehensible to developers. In addition, external projects in Coq can take minutes or hours to compile, so the edit-compile-test-debug loop is long. We have two additional goals to improve this workflow.

1. Finding minimal test cases as quickly as possible, given that experimenting with each program variant has long compilation time
2. Compilation of the test case in seconds or fractions of a second, so that developers can fluidly try hypotheses for solutions

5.1 Making the Minimization Process Itself Fast

In our goal to get the shortest reproducing test case as quickly as possible, it helps to first make any changes that might significantly speed up the execution time, and only after we are done with all of the changes that might improve running time should we try to further minimize the file with changes that are unlikely to impact compile time.

The slowest part of almost all Coq developments is proof scripts. (We expect the same is true of other tactic-based proof assistants.) Hence we attempt to remove proof scripts as early as possible. Since proof assistants check that proofs are valid, we cannot simply remove a proof, like we might remove a function body in a traditional programming language. However, most proof assistants have some mechanism for "giving up" on a proof or "trusting" the user, and Coq is no exception. Its mechanism involves any of `Admitted`, `Admit Obligations`, or the `admit` tactic. Replacing proof blocks with these commands, rather than just removing proof scripts, allows us to make much smaller and faster examples than might otherwise be possible.

5.2 Finding Textually Smaller Test Cases

The simplest function of the bug minimizer is to remove unneeded lines. As noted in the prior section, we try removing one syntactic unit at a time, moving backwards from the unit that triggered the error message.

However, we can easily enough get stuck in local minima, when we remove single commands and check that bug behavior is unchanged. For instance, there may be an irrelevant lemma that we want to remove.

```
Lemma irrelevant : two = 2.
Proof. reflexivity. Qed.
```

Since Coq forbids nested lemmas, removing statements one-at-a-time will not work, as the state

```
Lemma irrelevant : two = 2.
Proof. reflexivity.
```

results in an error about nested proofs, if there is a theorem afterward.

We instead group statements into *definition* blocks to be removed all at once. Coq luckily has a mode[2] that emits information about the boundaries of these definition blocks. This way, we can remove the lemma block all at once.

We could in theory deal with more complicated nesting structure, for example trying to remove an entire section or module at a time. The delta tool [14] is in fact built around preprocessing the file into one that exposes nested structure clearly, then removing well-parenthesized blocks. However, removing statements, grouped into definitions as necessary, suffices for removing time-consuming code.

5.2.1 The `Program` Construct

One Coq construct that does not fit neatly into this approach is `Program`, where a function definition is associated with following proofs of obligations related to dependent typing. We cannot just look for `Program` statements followed by `Obligation` blocks to remove all together, because `Obligation` blocks can be interleaved with other definitions. Luckily, we can replace any obligation block with a use of the `Admit Obligations` command, which admits all remaining obligations – and it happily handles any case with *no* remaining obligations, so we need not worry about introducing duplicate invocations.

5.2.2 Empty Sections and Modules

Removing statements one-at-a-time will not always be able to remove empty sections (nor empty `Module`s or `Module Type`s). That is why we have a pass dedicated to removing empty sections, modules, and module types.

5.2.3 Exporting Modules

Coq's features to import and export modules (e.g., including all definitions of one module inside another) can create some particularly thorny situations for statement-at-a-time shrinking. If we remove just an `Import` commands, then later commands fail because important

[2] It can be accessed by invoking `coqtop -emacs -time`.

identifiers are out-of-scope. If we remove just the definition of the imported module, then the `Import` fails. The solution is to merge these two commands, so that they become a candidate for removal together. We change `Module` commands into `Module Export` commands to this end. Often that change renders later `Import` commands redundant, so they are removed by later passes.

5.2.4 Splitting Definitions

One pass in the minimizer tries to replace traditional definitions with uses of the interactive proof mode, which is a first step toward admitting those proof bodies (i.e., postulating existence of identifiers rather than giving their definitions) in later steps.

5.2.5 Early Removal of Unused Constants

There are some likely-to-succeed steps that we try early on, which are superseded by removing each and every structured block one-at-a-time but may result in faster minimization. The primary example of this sort of step is removing tactics, `Variable` and `Context` statements, and definitions that are not referenced after their definitions.

5.2.6 Splitting `Imports` and `Exports`

It may be the case that users import modules that they never use, such as in `Import unused1 used unused2`. To allow eventual removal of `unused1` and `unused2` even when the `Import used` statement cannot be removed, we have a pass that attempts to split such statements into separate `Import` statements, resulting in `Import unused1. Import used. Import unused2.`

5.3 Finding Test Cases That Coq Processes More Quickly

We mentioned how admitting proofs is a very handy step to shrink files and get them processed more quickly. There are, however, a few gotchas to keep in mind.

The first quirk is around transparency vs. opacity of lemma definitions; that is, whether the generated proof term is accessible to later definitions. Either choice (transparent vs. opaque) can break some developments. Marking a proof-mode definition *opaque* will break later definitions that unfold the definition and then perform further tactic-based surgery on it, while marking a proof-mode definition *transparent* could cause previously failing unfoldings to succeed. Therefore, we always try both styles of marking a lemma admitted.

Some lemma proofs are declared as transparent rather than opaque, where later steps really do depend on their details. If those dependencies are too specific, then our shrinking heuristics are not going to work well. However, one common-enough case is where a later definition uses tactics to *unfold* an earlier definition, going on to use other tactics that may very well be able to adapt to changes in that definition. There are at least two different ways to mark a proof as admitted (`Admitted` vs. using a preexisting `Axiom`), which can switch up whether the associated definition is considered opaque or transparent.

Additionally, we may want to admit some parts of the proof script without replacing all of it. Currently, we use a rather conservative heuristic: Coq has a tactical `abstract` that executes the tactic it is passed as an argument, making the resulting proof term opaque. Such subproofs should be able to be replaced with `admit` without changing the behavior of the proof script. The details are a little subtle, e.g. to avoid changing which section variables a proof depends on and thus changing its type outside the section.

6 Standalone Files

While the complex structure of external developments is a boon to stress-testing Coq, there are three reasons for wanting to reproduce bugs in standalone files.

1. It is challenging for developers to understand the intricacies of external developments well enough to diagnose root causes.
2. Build systems are necessary to handle multiple files, but using them adds unnecessary overhead in the debugging workflow.
3. Intricate file-dependency structure complicates test-suite infrastructure, whereas having self-contained files results in a simpler test suite.

Naïvely, the way to produce a standalone file is to *linearize* the dependency tree and combine the contents of all files. We saw an example of roughly this strategy in Section 2, and e.g. C compilers follow this strategy in preprocessing `#include` directives.

Two difficulties arise when following this strategy in Coq:

1. As in all languages that allow shadowing of global symbols, inlining files changes what names are available and hence may result in unintended changes of behavior. The dependent typing and metaprogramming facilities of Coq largely eliminate the distinction between runtime and compile time. As a result, we have to inline not just function declarations but also function bodies, and thus the problem of name resolution is comparatively harder in Coq and similar languages than in those with simple types and without metaprogramming facilities. Furthermore, Coq has additional quirks around name resolution and (lack of) namespacing that have to be managed and worked around.
2. Coq has a great deal of global state (e.g., notations, universe polymorphism, the default tactic mode) that changes the way sentences are interpreted. Because there is no way to isolate changes on this global state fully, there may not even be *any* linearization that reproduces the same behavior.

6.1 Addressing Shadowing and Name Resolution

Coq assigns names based on three components: the name and location of the file in which the identifier is defined, the module structure surrounding the identifier, and the final name. For example, the constant `Coq.MSets.MSetPositive.PositiveSet.t` is defined in the file `MSets/MSetPositive.v`, which is bound to `Coq.MSets.MSetPositive`, in the module `PositiveSet`, with the name `t`.

If we were to inline this file into some other file `bug.v`, then the constant becomes `bug.PositiveSet.t`. We now have two choices: we can attempt to adjust the name of the constant on inlining, or we can adjust references to the constant.

We combine these strategies to maximize the chance of successfully inlining dependencies.

First, as shown in the example in Section 2, we wrap the contents in a module whose name matches that of the file (in this case, we wrap the contents in `Module MSetPositive`). Furthermore, since users can refer to this constant as `Coq.MSets.MSetPositive.PositiveSet.t`, `MSets.MSetPositive.PositiveSet.t`, or `MSetPositive.PositiveSet.t`, we can wrap this module in further modules (`Coq` and `MSets`) and `Export` them to make this naming scheme available. Finally, because Coq forbids multiple modules with the same absolute kernel name, we must wrap the top-level module in yet another module, with a uniquely generated identifier. While this strategy is not perfect, running afoul of bug coq/coq#14587 for example, we try a couple of variations on this strategy, and very often one of them is adequate for reproducing buggy behavior.

Second, we want to adjust references so that they still point at the same underlying object after inlining. Coq helpfully emits *globalization* files, which contain information about how Coq resolves almost all names in the file. Since Coq generates and installs these `.glob` files, we can use this information to transform both the names in the files we inline and the names that refer to constants in that file.

However, we cannot just blindly update all names, because these `.glob` files are not perfectly accurate[3] and are not complete[4]. Instead, we have found in practice that the most important names to resolve are those used in `Require`, `Import`, and `Export` statements. `Require` statements are sensitive to the searchpath flags (`-Q` and `-R`) passed to Coq. If we are inlining a file from Flocq into a file from VST,[5] for example, the `Require`s in the Flocq file may not resolve to the same files on disk when compiling with the compiler flags that VST uses. `Import` and `Export` statements, while not dependent on searchpath flags to the same extent as `Require`, still seem empirically more likely to refer to potentially ambiguous names than most other statements. Hence we choose to resolve the names used in `Require`, `Import`, and `Export` statements when inlining, letting Coq determine all other name resolution.

6.2 Addressing Nonlinearizability of Global State

While shadowing and name resolution are mechanically resolvable at least in theory, the global state of Coq is sufficiently disorganized that we are not aware of any fully general technical means of linearizing Coq files.[6] Hence our approach here consists of several partial workarounds.

The most basic technique to attempt to isolate global state is to wrap the inlined file in a module. Most state not explicitly marked as `Global` does not escape the boundaries of the module it is defined inside. As we already use module wrapping to handle name resolution as discussed in Subsection 6.1, we already reap the benefits of this technique.

Our only other technique is to try multiple linearizations and hope that one of them is adequate. We try inserting the file being inlined at the top of the file, as well as at the location where it is `Require`d. In the future, we might also want to try moving `Require`s up higher in the file, to try to handle more situations.

In Section 11, we discuss a few potential future avenues to better handling of global state. For example, we may want to more explicitly manage the state before and after inlining a file by taking advantage of Coq's ability to print the current settings of flags with `Print Options`.

6.3 Getting to Standalone Files Quickly

We have a flag that allows inlining dependencies all-at-once, much like `gcc` inlines all `#include`d files at-once. While originally all files were minimized in that way, having to process such a large file slowed down minimization drastically, often resulting in minimization times of multiple weeks. As a result, the current default behavior is to minimize the current file before inlining other files.

[3] See bugs coq/coq#15497 and coq/coq#14537.
[4] They are missing information, for example, on tactic-name resolution and notation interpretation.
[5] Flocq is a Coq library on floating-point arithmetic, and VST is a Coq library for verification of C code, which relies on Flocq.
[6] The `Require` command results in many side effects, including global setting of flags, opacity, and argument status; behavior of `auto with *`; hint databases; global overwriting of Ltac definitions; presence or absence of constants that change the behavior of built-in tactics such as `tauto`; and even the presence of constants with certain kernel names can change shadowing behavior. Some of these effects can even be set on the command line, and at present there is no way to determine what flags were used to compile a given installed file.

Furthermore, we want to ensure that we only inline files that are actually used. Much like we want to split `Import` and `Export` statements in Subsubsection 5.2.6, we also want to split `Require` statements, for example from `Require unused1 used unused2.` to `Require unused1. Require used. Require unused2.`

Additionally, if the buggy behavior depends on a file only for its own dependencies, we prefer to inline the transitive dependency directly rather than needing to inline the entire intermediate file. To that end, we have a pass that performs the transitive closure of the dependency relation, inserting `Require` statements at the top of the file for all transitive dependencies of the file being minimized. Because we insert the `Require`s in dependency order, removing one statement at a time in reverse order will give us the minimal `Require`s needed to reproduce the error message. This strategy ensures that we only inline dependencies that are actually necessary.

7 Smooth Developer Experience

In order to analyze a specific source file, we need to take a few steps.

1. Unpack and install both the succeeding and failing versions of Coq and the tested projects.
2. Replace the Coq binaries with wrappers that print out the arguments that Coq was called with, as well as `COQPATH` (an environment variable listing directories to be searched for imported modules) and the current directory.
3. Run the succeeding and failing versions of Coq on the tested projects, ensuring that the version that should pass does in fact pass, and the version that should fail has a recognizable error message.
4. Parse the build log to determine the buggy file name and the arguments to pass to Coq, using the extra logging introduced by our wrappers. This workflow means that we need not interface directly with varied build systems of different tested projects.
5. Run the failing version of Coq on the file triggering the buggy behavior.
6. Parse the error message, ensuring that it matches with the error message from the build log. (See Section 4 for subtleties in that comparison)

Again, the goal of the minimizer is to take a tested project that succeeds on the tip of Coq's master branch and fails on a given Coq pull request (PR), emitting a small, standalone file that succeeds on master and fails in the same way on the PR. In order to do so efficiently, we reuse the CI artifacts from Coq. We download the prebuilt versions of Coq from master and from the tip of the PR. From just these artifacts and the name of the failing project, we must assemble enough information to run the bug minimizer. We replicate Coq's generic CI workflow to install Coq as well as any dependencies of this project, into different directories: one for the version of Coq expected to pass and another for the version of Coq expected to fail. We also reuse Coq's generic CI workflow to figure out the error message and the failing file we want to minimize.

Let us justify the extra information that our Coq wrapper programs log. We need `COQPATH` to ensure that we have the right search path for the dependencies of `coqc`, the command-line Coq compiler. We need the command-line arguments so that we know what flags to tell the bug minimizer to pass to `coqc`. Note that we *must not* change relative paths to absolute ones when passing arguments along to `coqc`, because the output of `coqc` is sensitive to the difference between relative and absolute paths, so changes can muddle tests that are meant to produce output files (and did in the past, for example with `ci-elpi`). We can locate the error message by looking for the last instance of `File "`f`", line `ℓ`, characters `n-m:` followed immediately by a line beginning with `Error`. (Note that warning messages also emit

the `File ...` line, but we do not want to catch warnings.) We look for the last instance of the wrapper debug printout information that points at the same file, though, so long as we were careful always to build single-threadedly, we could instead just look for the most recent debug printout before the error message.

Given this information, we adjust the arguments so that we can tell the bug minimizer where the dependencies live for both the passing and failing versions of Coq. We then pass this information to the bug minimizer:

- the location of the file to be minimized;
- the log file containing the error message, which must match the error message that the minimizer believes the file produces;
- the locations of the `coqc`, `coqtop`, and `coq_makefile` programs for the tip of the PR;
- the location of the `coqc` program for the master branch;
- the locations of the dependencies for both the passing and failing versions of Coq, parsed from the command-line arguments and from walking the directories in `COQPATH`;
- any arguments to `coqc` that are neither naming dependency locations nor known to be both irrelevant to the processing of the file and counterproductive to the minimizer's operation (such arguments are `-batch`, which applies only to `coqtop`; `-time`, which will only make logs of the minimizer much longer; and `-noglob`, `-dump-glob`, and `-o`, which interfere with the generation of outputs used by the minimizer).

8 An Alternative Usage Mode

Up to this point, we have talked about using the Coq Bug Minimizer exclusively to minimize RDCT failures for debugging faulty changes in Coq. Our tool can also be used to minimize test cases for newly found bugs in Coq. In this mode, a bug reporter can write a shell script that invokes a *single* version of Coq to produce buggy behavior on some Coq file, asking coqbot to produce a minimal example from this script. When running in this mode, we place an additional constraint on the minimizer that the proof script generating the error message should be left untouched, which allows bug reporters to write proof scripts such as

```
some_tactic; lazymatch goal with
| buggy_goal => fail 0 "bug remains"
| [ |- ?G ]  => fail 0 "bug disappeared!" G end.
```

to customize the desired reproducing case, trusting that the entire file will not be minimized to something silly like `Goal False. fail 0 "bug remains".`

9 Integration in Coq's CI and Evaluation of Results

9.1 Triggering the Minimizer

The Coq project uses a custom bot to automate everyday tasks, including triggering CI and reporting its results to the GitHub repository [19]. We have extended this bot to automatically propose and manage the minimization of failing test cases. The bot posts a comment to propose to run minimization when a PR has passed Coq's internal test suite but has failures with external projects, where these external projects have built successfully on the base commit (on the master branch).

If someone answers with a comment to trigger minimization, then the bot prepares a branch with all the information needed by the minimizer and pushes this branch to an external repository dedicated to running the minimizer. This triggers a GitHub Action

workflow that proceeds with the minimization process. GitHub Action jobs have a 6-hour timeout, so by the limit, the bot answers back with the results of the minimization process. If the minimization was stopped because of the timeout, then the bot automatically restarts it by reusing the file obtained at the previous step.

9.2 Research Questions

To evaluate the usefulness of our bug minimizer, we investigate several research questions:

RQ1: How often does the minimizer successfully produce a reduced test case from the RDCT failure it was triggered on?

RQ2: How often is this reduced test case fully standalone (no dependencies other than Coq's standard library)?

RQ3: How long does it take to produce such reduced test cases?

RQ4: What are the sizes of the reduced cases?

RQ5: How long do the reduced cases take to run?

RQ6: What is the amount of code reduction?

9.3 Data Collection and Analysis

To support reproducing the results, we provide our data collection and analysis code (as a Jupyter notebook) and our dataset (as a CSV file) in the supplementary materials.

We retrieve the runs of the bug minimizer by looking for PRs in the Coq GitHub repository with the words "coqbot ci minimize", and we fetch all comments from the bot (timestamp and body text) from these PRs using GitHub's GraphQL API. We exclude PRs opened by the first author, as most of these PRs were for testing the minimizer integration and debugging issues. When the minimizer is triggered, the bot answers with a comment "I have initiated minimization ..." or "I am now running minimization ...", providing the list of projects on which it is being run. Then, when it finishes minimizing a project, it produces a comment with the minimized file. This file starts with header comments containing useful information about the minimization process. The comment may also contain "interrupted by timeout, being automatically continued" if the minimization process timed out and has to be restarted to go further, which the bot automatically does. We ignore these comments, only looking for final reduction outputs. Finally, the bot posts a comment starting with "Error: Could not minimize file" when it was not able to minimize the requested failure, for instance, because it could not reproduce it or could not reproduce the successful run on the base branch.

We match comments indicating the start of the minimization with comments indicating the end of it, using these two comments to determine if the minimizer was able to produce a reduced test case, find how long it took, and answer our other research questions. To avoid double-counting multiple runs on the same RDCT failure, we only look at the first bug-minimizer trigger on a given PR and a given project.

9.4 Results

9.4.1 RQ1: How often does the minimizer produce a reduced test case?

Looking only at the first minimization runs for a given PR and project, we have identified 191 runs on 51 PRs (very often, several minimization runs are started in the same PR on different projects). On these 191 runs, 75% succeeded in producing reduced test cases. We count as failed runs the ones where the bot reported "Error: Could not minimize file", the

ones where we could not find a comment marking the end of minimization, and the ones where the bot answered with a minimized file but this file was not actually reduced from the initial test case (which we can detect from the header comments).

There were 5 runs for which we found no comment marking the end of minimization. By manually looking at them, we have determined that 4 out of 5 were caught in infinite loops and had to be canceled manually. Loops can arise when the 6-hour timeout of the minimization process is not enough to make any new progress and thus the minimization gets stuck without ever reaching its end. Typical circumstances are when testing out a single change takes over 20 seconds, since we only have enough time to compile a 20-second-long file about a thousand times in six hours. The last case of our 5 seems to be coqbot having failed to post the comment marking the end of the minimization process.

There were 19 runs that concluded with an explicit "Error: could not minimize file" comment. These errors are often due to issues downloading CI artifacts (9 runs), for instance because the corresponding base CI jobs have been skipped or the CI artifacts have expired. Runs concluding with errors can also happen because of bugs in Coq or in the tested projects' build infrastructure that prevent minimization. Virtually all these issues were reported, and most of them are already fixed. For instance, the MetaCoq project alone was responsible for 5 failures because of issues in its build system.

Finally, there were 23 runs ending with comments reporting on supposedly minimized files but where (from the header comments or their absence thereof) we can conclude that the minimization process failed to start properly (e.g., because it could not reproduce the error message). Most of these problems were related to error-message parsing, namespace management, or similar issues that have been fixed by making the bug minimizer more robust to them (see Section 4 to Section 6). A few of these issues have been noted but not yet fixed. Finally, a few of these failed runs were due to the minimizer being misused or called on a project that had failed for a reason that was unrelated to the PR.

The accompanying notebook contains specific comments for each of the failed runs.

9.4.2 RQ2: How often is this reduced test case fully standalone?

We consider that a reduced test case will be most useful if any dependency beyond Coq's standard library was successfully inlined, leaving it possible to run the reduced test case without needing to import any additional dependency. As a result, it is more likely that the test case can be added to Coq's test suite.

To measure how often the reduced test case is standalone, we rely on the minimizer recording when it failed to inline a dependency in the header comments of the minimized file. This feature was only added recently, so we only perform this measurement on the 47 successful runs of the minimizer that had this information available. On these 47 runs, there were only 5 failures to inline dependencies fully, i.e., the minimizer produced a fully standalone file in 89% of the cases.

Looking at the 5 failures to inline dependencies, we observe several types of reasons. One case was related to robustness to changing error messages, one case was related to a build-system issue in the project being minimized, and 3 cases were due to a common issue blocking attempts at all inlining methods. All of these issues have been fixed since then.

9.4.3 RQ3: How long does it take to produce such reduced test cases?

We compute the duration of minimization as the time delta between the start and the end comments. This method overapproximates the actual time spent in the minimization process, since it also includes time setting up a VM and possibly waiting in the queue for an available runner. We can look at this duration for both successful and failed runs.

For failed runs, we observe that the average duration for the minimization to conclude is 5 minutes (306 seconds) and that the maximum duration is 15 minutes (890 seconds).

For successful runs, we observe more variety. The minimum duration is 4 minutes (232 seconds), the maximum duration is 20 hours (73072 seconds), and the average is 104 minutes (6238 seconds). 50% of the successful runs finish in under 20 minutes (1218 seconds), and 80% finish in under 140 minutes (8396 seconds). This number is still reasonable compared to the time that contributors routinely spend waiting for the results of Coq's CI [18].

9.4.4 RQ4: What are the sizes of the reduced cases?

For the last three questions, we focus mainly on the 42 recent minimization runs that are known to have produced standalone files.

The shorter the reduced test case, the more useful it is: it can help developers understand the problem more quickly, and it makes it more likely that it will be added to Coq's test suite. Here again, there is some variety in the size of the reduced cases (counted in number of lines). The average size is 270 lines, and the maximum size is 2648 lines. However, 25% of the reduced cases are under 39 lines, 50% are under 114 lines, and 75% are under 262 lines.

Results on the full set of 144 successful minimization runs are of the same order of magnitude, with an average at 367 lines and a maximum size of 3804 lines.

Developers have the option to perform additional minimization manually and restart the automatic minimization process on their manually reduced cases, which can help obtain even more reduced cases, but we have not evaluated this feature quantitatively.

9.4.5 RQ5: How long do the reduced cases take to run?

Following a recent addition, the minimizer has reported the expected `coqc` compile time as part of the header comments in the minimized file. Our recent 42 standalone cases all had this field available. We observe that the reduced cases take on average 1.25 seconds to run, although 75% of them take under half a second, while the maximum time is 26.5 seconds.

9.4.6 RQ6: What is the amount of code reduction?

To compute how much code reduction there was, we use data that the minimizer records about each minimization step (how many lines it started from and how many lines it ended up with). These numbers go up at times because of the process of inlining external dependencies. On the other hand, dependencies are only inlined if they could not simply be removed, so these numbers do not include the size of the files that were previously imported but did not need to be inlined during the minimization process.

We aggregate these numbers by simply taking the sum of the differences in line count at the beginning and the end of each minimization step. We compute the amount of code reduction by taking the ratio of the final size over the total test-case size, defined as being the sum of the final size and the total number of removed lines. We obtain an average figure of 31%, which means that the final test-case size is on average one-third the size of the original test (including the dependencies that actually matter for the test case).

If we compute the size difference only looking at the initial file we started from and the final file we obtained, without accounting for the inlined dependencies, then we get an average ratio of 50%, which means that the final file is on average half the size of the file we started from. Note that because of dependency inlining, nothing prevents the reduced test case from being longer than the file we started from, which does happen in 6 out of 42 cases. If we look only at the 36 cases for which there was some code reduction, we get that the average reduction is by a factor of 4 to 5. If we look only at the 6 cases for which there was code expansion, we get that the average expansion is by a factor of 2.

9.5 Limitations of our Evaluation

Evaluating a bug minimizer for a proof assistant such as Coq is difficult because there is no preexisting benchmark that it could be run on. In this paper, we have decided to take advantage of the integration of our minimizer in the RDCT infrastructure of Coq to evaluate it on real use cases where Coq developers have felt the need for it.

While we have taken steps to ensure that the evaluation is as unbiased as possible (such as not using reruns of the minimization on the same project in the same PR), our evaluation is still limited by our choice to use real use cases. In particular, it should be noted that our evaluation results are not obtained on a fixed version of the minimizer. On the contrary, the minimizer has evolved (and is still evolving) in reaction to the very same cases on which we have evaluated it. Since we always account only for first runs, many cases where the minimizer has been counted as failing have been eventually fixed and would result in successful runs today. Subsequent runs on other projects or other PRs may have succeeded thanks to earlier fixes.

Other limitations are that our computation of the minimization duration is an overapproximation that also includes the time for e.g. setting up a VM to run the process, and that our evaluation of several research questions is based only on a subset of recent minimizer runs.

Due to all these limitations, our evaluation should only be understood as demonstrating the feasibility of our approach and the usefulness of its application to the development of Coq. However, it should not be understood as a basis that future versions of the minimizer, or alternative minimizers, can compare to, since today's version would already obtain different results if it were rerun on all these cases.

10 Related Work

Our work is at the intersection of two research areas: research on debugging techniques, which is a subdomain of software-engineering research; and research on proof assistants.

Debugging is a thoroughly explored topic, but mostly with a focus on more mainstream and less formal languages than Coq. In this research domain, test-case-reduction techniques have been studied for standard programming languages and compilers [3]. There are two types of approaches that have been proposed. First, there are generic approaches that are supposed to work for any programming language, by using structure information on the program being reduced. Examples include delta debugging [16] but also the generalized tree-reduction algorithm [7] and the syntax-guided Perses tool [7]. These generic techniques would not be likely to work well for Coq programs without careful adaptation, because many Coq programs can be considered syntactically valid even if completely nonsensical. For instance, we have already mentioned the issue with removing a `Qed` statement at the end of a tactic-based proof. Despite breaking a semantic block of code, this change does not actually produce a syntactically invalid Coq program.

Second, there are programming-language-specific approaches, which take advantage of specific knowledge to make the test-case reduction more performant. Our own work is related to this second category, where most tools focus on mainstream languages like C. Some are even dedicated to reducing the output of specific test-generation frameworks such as Csmith [12].

However, work on generating many diverse test cases from nothing has complementary value. Csmith [15] has an effective algorithm based on knowledge of C semantics, to provoke undefined behavior. Techniques like equivalence modulo inputs [9] find compiler bugs via differential testing, where a compiler is run on programs that are known to have the same semantics. Perhaps this generative approach would also be useful for proof assistants, composed fruitfully with test-case reduction as we have presented.

Finally, the literature has identified the issue of *slippage* in test-case reduction [4, 8], which is when the initial and reduced cases produce different compiler bugs. This challenge was one of the main ones we had to account for in designing our bug minimizer (see Section 4).

Proof-assistant ecosystems were already no stranger to testing techniques. For instance, Isabelle/HOL's Nitpick [1] uses Boolean satisfiability to find theorem counterexamples. QuickChick [11] does random test generation to try to falsify Coq theorems. These tools are handy to save users from investing time in trying to prove false theorems. Testing-based approaches to debugging *proof assistants themselves* are a complementary topic.

11 Future Work

We were pleasantly surprised to find that several "shortcuts" in the logistics behind the minimizer led to good results empirically, but some of these may be worth revisiting to improve results even more. In various places, we use workarounds (like `.glob` files) to avoid integrating a proper Coq parser, but there would be advantages like being able to remove specific fields from record types. We remove single commands at a time, rather than removing entire well-balanced command blocks, which probably costs us in minimization time.

Integrating with Coq's parser would also allow us to more naturally handle code associated with Coq developments but that uses different statement-ending conventions than standard Coq code, such as coq-elpi code and OCaml plugin code.

Another broader opportunity is finding related groups of commands that need to be removed together, to avoid changing the error message. For instance, we might want to move a lemma out of a module, to the top level of a file. Removing the commands that open and close the module might suffice, even if removing either one alone disturbs the error message. A general-enough version of this process could replace many specific passes.

One remaining aggravation is proper handling of lemma proofs within sections, where the details of the lemma proof influence which section variables are kept in the lemma's type. We could use the `Set Suggest Proof Using` command to instruct Coq to tell us which section variables are used by each proof; we could then insert `Proof using` clauses to allow us to replace proofs with `Admitted` without changing dependencies on section variables.

As mentioned in Subsection 6.2, we would like to improve the ability of the minimizer to linearize dependency trees and handle Coq's global state. We could, for example, print out the full table of flag settings at a particular point, reset them to the initial values before inlining a file, and then restore them after inlining. To fully handle global state, we would need some way to reconstruct the command-line flags used to compile installed files.

There are further-out ideas that could speed minimization significantly but might require significant modifications to Coq itself. Incremental compilation would be helpful, to save us from rerunning long proof scripts every time we change single lines below them. Minimizing multiple files in parallel, rather than only inlining files, would allow us to take advantage of multicore execution within single minimization jobs.

References

1 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, *Interactive Theorem Proving*, pages 131–146, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. doi:10.1007/978-3-642-14052-5_11.

2 Martin Burger, Karsten Lehmann, and Andreas Zeller. Automated debugging in Eclipse. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 184–185, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1094855.1094926.

3 Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing. *ACM Comput. Surv.*, 53(1), February 2020. doi:10.1145/3363562.

4 Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 197–208, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2491956.2462173.

5 Holger Cleve and Andreas Zeller. Finding failure causes through automated testing. In Mireille Ducassé, editor, *Proceedings of the Fourth International Workshop on Automated Debugging, AADEBUG 2000, Munich, Germany, August 28-30th, 2000*, 2000. arXiv:cs/0012009.

6 Jason Gross. Coq bug minimizer, January 2015. Presented at The First International Workshop on Coq for PL (CoqPL'15). URL: https://jasongross.github.io/papers/2015-coq-bug-minimizer.pdf.

7 Satia Herfert, Jibesh Patra, and Michael Pradel. Automatically reducing tree-structured test inputs. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 861–871, Urbana-Champaign, IL, USA, 2017. IEEE Press. doi:10.1109/ase.2017.8115697.

8 Josie Holmes, Alex Groce, and Mohammad Amin Alipour. Mitigating (and exploiting) test reduction slippage. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*, A-TEST 2016, pages 66–69, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2994291.2994301.

9 Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 216–226, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2594291.2594334.

10 Lina Ochoa, Thomas Degueule, and Jean-Rémy Falleri. BreakBot: Analyzing the impact of breaking changes to assist library evolution. In *44th IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2022*. IEEE, 2022.

11 Zoe Paraskevopoulou, Cătălin Hriţcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. Foundational property-based testing. In *ITP 2015 - 6th conference on Interactive Theorem Proving*, volume 9236 of *Lecture Notes in Computer Science*, Nanjing, China, August 2015. Springer. doi:10.1007/978-3-319-22102-1_22.

12 John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 335–346, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2254064.2254104.

13 Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq Correct! verification of type checking and erasure for Coq, in Coq. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:10.1145/3371076.

14 Daniel S. Wilkerson and Scott McPeak. delta - delta assists you in minimizing "interesting" files subject to a test of their interestingness, February 2006. Presented at CodeCon 2006. URL: https://github.com/dsw/delta.

15 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1993498.1993532.

16 Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, pages 1–10, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/587051.587053.

17 Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Elsevier, 2009.

18 Théo Zimmermann. *Challenges in the collaborative evolution of a proof language and its ecosystem*. PhD thesis, Université de Paris, 2019. URL: https://hal.inria.fr/tel-02451322.

19 Théo Zimmermann, Julien Coolen, Jason Gross, Pierre-Marie Pédrot, and Gaëtan Gilbert. Advantages of maintaining a multi-task project-specific bot: an experience report. working paper, 2022. URL: https://hal.inria.fr/hal-03479327.

Undecidability of Dyadic First-Order Logic in Coq

Johannes Hostert
Universität des Saarlandes, Saarland Informatics Campus, Saarbrücken, Germany

Andrej Dudenhefner
Universität des Saarlandes, Saarland Informatics Campus, Saarbrücken, Germany

Dominik Kirst
Universität des Saarlandes, Saarland Informatics Campus, Saarbrücken, Germany

Abstract

We develop and mechanize compact proofs of the undecidability of various problems for dyadic first-order logic over a small logical fragment. In this fragment, formulas are restricted to only a single binary relation, and a minimal set of logical connectives. We show that validity, satisfiability, and provability, along with finite satisfiability and finite validity are undecidable, by directly reducing from a suitable binary variant of Diophantine constraints satisfiability. Our results improve upon existing work in two ways: First, the reductions are direct and significantly more compact than existing ones. Secondly, the undecidability of the small logic fragment of dyadic first-order logic was not mechanized before. We contribute our mechanization to the Coq Library of Undecidability Proofs, utilizing its synthetic approach to computability theory.

2012 ACM Subject Classification Theory of computation → Constructive mathematics; Theory of computation → Type theory; Theory of computation → Logic and verification

Keywords and phrases undecidability, synthetic computability, first-order logic, Coq

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.19

Supplementary Material `https://www.ps.uni-saarland.de/extras/fol-dyadic`

Acknowledgements We thank Yannick Forster for valuable feedback on drafts of this paper.

1 Introduction

In 1928, Hilbert and Ackermann [18] mused whether there is a general decision procedure for the *Entscheidungsproblem*, that is, the problem of whether or not a formula of first-order logic is valid in all models. In the following years, several strategies were developed to approach this problem. So-called *reduction theory* [20, 3, 31] tried to reduce the general Entscheidungsproblem to simpler classes of first-order formulas. For formulas over a *monadic signature*, that is, formulas where all function and relation symbols are unary, Löwenheim [26] already established in 1915 that the Entscheidungsproblem is decidable.

Validity for first-order formulas was first shown undecidable for the general case of an unrestricted signature in 1936 by both Church [4] and Turing [34]. Shortly afterwards, in 1937, Kalmár [20] finalized a validity-preserving reduction chain allowing to convert a general first-order formula to one over a *dyadic signature*, that is, one with only a single binary relation symbol (and no function symbols). The resulting (un)decidability classification regarding the signature is as follows:

▶ **Theorem** ((Un-)decidability of validity)**.** Validity for first-order formulas is, depending on the arity of the function and relation symbols in the signature:

1. decidable, if all function and relation symbols are at most unary.
2. decidable, if all relation symbols are nullary.
3. undecidable, if there is an at least binary relation symbol.
4. undecidable, if there is an at least binary function, and a non-nullary relation symbol.

Proof summary.

1. By Löwenheim [26].
2. Quantifiers can be ignored, and thus formulas degenerate into trivial propositional logic.
3. By Kalmár [20], refining Church [4] and Turing [34].
4. By Gurevich [16]. ◀

Seminal works in reduction theory by Kalmár [21, 20] and Ackermann [1] in 1932-1939, which minimized the signature and the quantifier prefix, as well as techniques developed by Gödel in 1929 [14] and Gentzen in 1936 [13], which minimized the fragment of logical connectives, soon established several axes along which formula classes could be determined decidable or undecidable. These mentioned restrictions are all *syntactic*, i.e. they classify formulas by restricting their syntactic formation rules. Complementary to this are *semantic* restrictions, like the one emerging from finite model theory, where validity and satisfiability only involve finite models.

The undecidability of *finite satisfiability*, which is the variant of the Entscheidungsproblem restricted to finite models, was first shown in 1950 by Trakhtenbrot [33], who reduced from a semantic problem on μ-recursive functions. Although Trakhtenbrot's result holds for general first-order formulas, it was already folklore in 1950 that this result can be reduced to the undecidability of finite dyadic first-order logic (e.g. Kalmár [20] already claims that his reduction applies here, albeit without proof). In general, the above theorem applies to finite validity (and satisfiability) mutatis mutandis, as do the results for minimizing the fragment of logical connectives.

In recent years, most of these results have been mechanized in proof assistants, mainly Coq [32]. Notably, a classical mechanization of undecidability results would start by mechanizing a model of computation as a reference point for undecidability, which is cumbersome as it involves programming low-level entities of this model. Alternatively, one can use a synthetic approach to computability theory, which uses the notion of computation implicit in constructive foundations such as the type theory underlying Coq. This approach was developed by Forster et al. [9, 8], building on ideas by Richman [30] and Bauer [2].

Using this framework, Forster et al. [9] mechanize the undecidability of the Entscheidungsproblem in Coq. As they work in a constructive setting logic, they need to separately consider the undecidability of validity, satisfiability, and provability, as the various theorems establishing their many-one equivalence only hold classically. Following this approach, Kirst and Hermes [22] mechanize the undecidability of case (3) of the above theorem by transforming formulas of first-order ZF set theory, shown undecidable as well, into a signature with just a single binary relation $\in$. Kirst and Larchey-Wendling [23] mechanize the general case of Trakhtenbrot's result, as well as a signature compression chain and further decidability results to establish all cases of the above theorem for finite models. All of these results are collected in the Coq Library of Undecidability Proofs [11].

This aforementioned library also contains a mechanization of the undecidability of Diophantine constraints satisfiability, due to Larchey-Wendling and Forster [25]. This problem was first shown undecidable in 1970 by Matiyasevich [27], building upon work by Davis, Putnam, and Robinson [6] and provides the basis for reductions in this paper.

Contributions. Complementing existing results, we develop novel compact reductions establishing the undecidability of dyadic first-order logic. We develop a compactly mechanizable variant of Diophantine constraints satisfiability and directly reduce from it to show validity, satisfiability (for both Tarski and Kripke semantics), and intuitionistic and classical provability undecidable. We develop a similar reduction for finite satisfiability and validity.

Additionally, we strengthen our results over the existing mechanizations, namely by reducing the logical fragment to a minimal one. For some problems, this includes eliminating the falsity constant. In particular, we provide the first mechanization[1] that first-order validity and provability are undecidable for a dyadic signature and minimal logical fragment.

Outline. In Section 2, we recall the basics of synthetic undecidability and the mechanization of first-order logic we use. In Section 3, we introduce the binary variant UDPC of Diophantine constraints satisfiability we base our reductions on and show that it is undecidable. Section 4 contains the reductions from UDPC to validity, satisfiability, and provability, as well as the later minimization of the logical fragment. Section 5 does the same for finite satisfiability and validity, again including a mechanization for the minimal fragment. Finally, Section 6 summarizes our results and compares them to prior work.

2 Preliminaries

2.1 Synthetic Undecidability

We work in the Calculus of Inductive Constructions, a constructive type theory [5, 28]. Our type theory includes dependent functions $(\lambda(x : A).\,(e : B)) : \forall(x : A).\,B$, as well as inductive types, which include the empty type $\emptyset$, the unit type containing $\star : \mathbb{1}$, products $(a, b) : A \times B$, and sums $A + B$, as well as the natural numbers $\mathbb{N} := 0 \mid S\,n$ and booleans $\mathbb{B} := \mathtt{tt} \mid \mathtt{ff}$. Further, we have optionals $\mathcal{O}(X) := \emptyset \mid \lceil x \rceil$ and lists $\mathscr{L}(X) := [] \mid x :: xs$. A type is *listable* if there is a list including all elements of that type.

Type universes form a cumulative hierarchy $\mathbb{T}_0 : \mathbb{T}_1 : \cdots$, along with $\mathbb{P} : \mathbb{T}_1$, the computationally irrelevant type of propositions. By default, this implements an intuitionistic logic, where the law of excluded middle $\mathsf{LEM} := \forall P : \mathbb{P}.\,P \vee \neg P$ is not asserted. Since the hierarchy of types is cumulative, we omit indices wherever not necessary.

Since our type theory is constructive, functions defined in it are computable. In particular, this type system is implemented by the Coq proof assistant [32], which witnesses the computability and allows extraction to other programming languages.

Synthetic undecidability theory [8, 9] describes the approach for mechanizing undecidability proofs underlying the Coq library of Undecidability Proofs [11]. Due to the implicit computability of functions in this type theory, one can specify and verify computable functions without reference to a particular model of computation. For instance, a problem (a unary predicate) is decidable if there is a function computing its truth value. We refer the reader to the mentioned literature [8, 9] for a more detailed justification of the synthetic method.

▶ **Definition 1.** *Let $P : X \to \mathbb{P}$ be a problem on $X : \mathbb{T}$.*

- *The problem $\overline{P}$ such that $\overline{P}\,x := \neg P\,x$ is the* complement *of P.*
- *A function $f : X \to \mathbb{B}$ is a* decider for P *iff $\forall x : X.\,Px \leftrightarrow fx = \mathtt{tt}$.*
- *A function $f : \mathbb{N} \to \mathcal{O}(X)$ is an* enumerator for P *iff $\forall x : X.\,Px \leftrightarrow \exists n.\,fn = \lceil x \rceil$.*
- *P is decidable, written $\mathsf{dec}(P)$, iff there is a decider for P.*
- *P is enumerable, written $\mathsf{enum}(P)$, iff there is an enumerator for P.*

In particular, we are able to mechanize many-one reductions, verify them, and thereby establish that a problem is (synthetically) undecidable if we can many-one reduce to it from the halting problem of the λ-calculus, which is known to be undecidable [4]. We refer to

[1] Accessible from the supplementary web page and hyperlinked with every highlighted statement.

this halting problem as $HALT_\lambda$. Note that while $HALT_\lambda$ is also defined on open terms, it is many-one equivalent to a more canonical version of the halting problem on closed λ-expressions [8].

▶ **Definition 2.** *A problem $P : X \to \mathbb{P}$ on $X : \mathbb{T}$ is undecidable iff* $\mathsf{dec}(P) \to \mathsf{enum}(\overline{HALT_\lambda})$.

▶ **Definition 3.** *A problem $P : X \to \mathbb{P}$ on $X : \mathbb{T}$ many-one reduces to $Q : Y \to \mathbb{P}$, written $P \preccurlyeq Q$, iff there is a function $f : X \to Y$ such that $\forall x : X.\, Px \leftrightarrow Q(fx)$.*

When verifying a reduction, we typically call the $\to$-step preservation and the $\leftarrow$-step reflection.

▶ **Fact 4.** *If $\overline{HALT_\lambda}$ or $HALT_\lambda$ many-one reduces to P, then P is undecidable.*

2.2 First-Order Logic

First-order logic (FOL) is a logic where quantifiers may only range over individuals of the model, and not over functions or relations on these individuals.

We work within the mechanization of first-order logic already given in the Coq Library of Undecidability Proofs [11], which was synthesized by Kirst and Hermes [22] from previous work by Forster et al. [9, 10] and Kirst and Larchey-Wendling [23]. We recollect their definition here for ease of access:

First, fix a signature $\Sigma = (\mathcal{F}_\Sigma; \mathcal{P}_\Sigma)$ of function symbols $f : \mathcal{F}_\Sigma$ and relation symbols $P : \mathcal{P}_\Sigma$ with arities $|f|$ and $|P|$, which are then used to describe terms $t : \mathcal{T}$ and formulas $\varphi : \mathcal{F}$ as inductive types:

$$t ::= \mathsf{x}_n \mid f\,\vec{t} \quad (n : \mathbb{N}, \vec{t} : \mathcal{T}^{|f|}) \qquad \varphi ::= P\,\vec{t} \mid \bot \mid \varphi \to \psi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \forall\varphi \mid \exists\varphi \quad (\vec{t} : \mathcal{T}^{|P|})$$

While this definition and the mechanization uses de Bruijn indices to implement binding [7], we will use named binders on paper to improve readability.

Next, define the usual Tarski semantics providing an interpretation of formulas:

▶ **Definition 5.** *A* model $\mathcal{M}$ *consists of a domain type D as well as functions $f^{\mathcal{M}} : D^{|f|} \to D$ and $P^{\mathcal{M}} : D^{|P|} \to \mathbb{P}$ interpreting the symbols in the signature Σ. Given a variable assignment $\rho : \mathbb{N} \to D$ we define* term evaluation $\hat{\rho} : \mathcal{T} \to D$ *and* formula satisfaction $\rho \vDash \varphi$ *by*

$$\hat{\rho}\,\mathsf{x}_n := \rho\,n \qquad \hat{\rho}\,(f\,\vec{t}) := f^{\mathcal{M}}\,(\hat{\rho}\,\vec{t}) \qquad\qquad \rho \vDash P\,\vec{t} \;:=\; P^{\mathcal{M}}(\hat{\rho}\,\vec{t})$$

where the remaining cases of $\rho \vDash \varphi$ map each logical connective to its meta-level counterpart.

If a model $\mathcal{M}$ satisfies a formula φ for all variable assignments ρ, write $\mathcal{M} \vDash \varphi$.

We also follow Kirst and Hermes' [22] mechanization of provability, which is based on an inductive natural deduction system. We write $A \vdash \varphi$ for a list of formulas A and a formula φ if φ can be deduced from A in the intuitionistic deduction system, and $\vdash_c$ for the classical one. The full set of deduction rules can be found in [22].

In particular, these notions induce the following decision problems on formulas:

▶ **Problem 6** (Variants of the Entscheidungsproblem)**.**

- $\mathsf{VAL}\,\varphi := \forall\mathcal{M}.\, \forall\rho : \mathbb{N} \to D.\, \rho \vDash \varphi$
- $\mathsf{SAT}\,\varphi := \exists\mathcal{M}.\, \exists\rho : \mathbb{N} \to D.\, \rho \vDash \varphi$
- $\mathsf{PRV}\,\varphi := [] \vdash \varphi$
- $\mathsf{PRV}_c\,\varphi := [] \vdash_c \varphi$

When mechanizing finite model theory, we closely follow the mechanization developed by Kirst and Larchey-Wendling [23]. In particular, a finite model is not just listable, but also has decidable relations.

▶ **Definition 7.** A model $\mathcal{M}$ is *finite* iff D is listable and for all relation symbols p, $\mathsf{dec}(p^{\mathcal{M}})$ holds.

▶ **Problem 8** (Variants of the Entscheidungsproblem on finite models)**.**
- FVAL $\varphi := \forall \mathcal{M}.\, \mathcal{M}$ finite $\to \forall \rho : \mathbb{N} \to D.\, \rho \models \varphi$
- FSAT $\varphi := \exists \mathcal{M}.\, \mathcal{M}$ finite $\wedge\, \exists \rho : \mathbb{N} \to D.\, \rho \models \varphi$

3 Uniform Diophantine Pair Constraints

In order to achieve compact reductions to first-order logic, we need to pick a source problem which allows for easy formalization in first-order logic. Our particular problem is satisfiability for a variant of Diophantine constraints, that is, for a collection of equations on $\mathbb{N}$ all having a certain shape. The concrete shape is defined by the relation $\wr$.

▶ **Definition 9** ($\wr$)**.** *$\wr$ is the following relation on $\mathbb{N}^2 \times \mathbb{N}^2$:*

$$(a, b) \wr (c, d) := a + b + 1 = c \wedge b^2 + b = d + d$$

This relation has several suitable features. First of all, it is total and functional. It further allows encoding the successor operation, addition, and squaring, which suffices to express all of natural arithmetic. As Definition 12 shows, it can easily be characterized inductively. Intuitively, this relation encodes the Gaussian sum $d = \frac{b^2+b}{2} = \sum_{i=1}^{b} i$.

To define a constraints collection, we pick $\mathbb{N}$ as a concrete, countably infinite, discrete type of variables $\mathcal{V}$ and define:

▶ **Definition 10** (Uniform Diophantine Pair Constraints)**.** *A* uniform Diophantine pair constraint *is a tuple $((x, y), (z, w))$, with $x, y, z, w : \mathcal{V}$. An assignment $\rho : \mathcal{V} \to \mathbb{N}$* satisfies *such a constraint $((x, y), (z, w))$ iff $(\rho\, x, \rho\, y) \wr (\rho\, z, \rho\, w)$.*

▶ **Problem 11** (UDPC)**.** UDPC *is the following problem:*

$$\mathsf{UDPC}\,(h : \mathscr{L}\left(\mathcal{V}^2 \times \mathcal{V}^2\right)) := \exists \rho. \forall ((x, y), (z, w)) \in h.\, \rho \text{ satisfies } ((x, y), (z, w))$$

As mentioned, the problem also admits an inductive characterization:

▶ **Definition 12** (Inductive $\wr$)**.** *The relation $\wr$ can be equivalently characterized by:*

$$\text{Base}\ \frac{}{(a, 0) \wr (a + 1, 0)}$$

$$\text{Step}\ \frac{(a, b') \wr (c', d') \qquad (d', b') \wr (d, d') \qquad (b', 0) \wr (b, 0) \qquad (c', 0) \wr (c, 0)}{(a, b) \wr (c, d)}$$

This characterization already hints at an axiomatic definition of $\wr$, as the axioms implied by the constructors are almost sufficient. The total axiomatization, consisting of the *Base*, *Step*, and *Tieback* axioms, is given here:

▶ **Lemma 13** (Basic properties of $\wr$)**.**
- Base*:* $(a, 0) \wr (c, 0)$ *iff* $c = a + 1$
- Step*:* $(a, b) \wr (c, d) \wedge b \neq 0$ *iff there are* b', c', d' *such that* $(a, b') \wr (c', d') \wedge (d', b') \wr (d, d') \wedge (b', 0) \wr (b, 0) \wedge (c', 0) \wr (c, 0)$
- Tieback*:* $(a, 0) \wr (c, d) \to d = 0$
- weak Step*:* $(a, b') \wr (c', d') \to (d', b') \wr (d, d') \to (b', 0) \wr (b, 0) \to (c', 0) \wr (c, 0) \to (a, b) \wr (c, d)$

Base and Step are stronger than the corresponding constructors of Definition 12, as they also capture elimination principles. The intuition behind the Step rule is that it encodes a Gaussian sum as we go from b' to $b'+1=b$: We must change $d'=\sum_{i=1}^{b'} i$ to $d=\sum_{i=1}^{b'+1} i$, so $d'+b'+1=d$ must hold, which is ensured by $(d',b')\wr(d,d')$.

▶ **Theorem 14** (Soundness and Completeness). *Any relation $\mathcal{R}$ on $\mathbb{N}^2\times\mathbb{N}^2$ satisfying Base, Step, and Tieback is equivalent to $\wr$. If $\mathcal{R}$ only satisfies Base and weak Step then completeness holds, while soundness does not necessarily do so anymore: only $(a,b)\wr(c,d)\to(a,b)\ \mathcal{R}\ (c,d)$ can be proven.*

▶ **Corollary 15**. *$\wr$ is the smallest relation satisfying the Base and weak Step axiom.*

▶ **Proposition 16** (Irreflexivity of $\wr$). $\forall(a,b):\mathbb{N}^2.\,\neg((a,b)\wr(a,b))$

To show UDPC undecidable, we reduce from a very similar problem called UDC, defined on uniform Diophantine constraints. An undecidability proof for this problem is already mechanized in the Coq Library of Undecidability Proofs.

▶ **Definition 17** (Uniform Diophantine Constraints). *A* uniform Diophantine constraint *is a triple (x,y,z), with $x,y,z:\mathcal{V}$. An assignment $\rho:\mathcal{V}\to\mathbb{N}$* satisfies *a uniform Diophantine constraint (x,y,z) iff*

$$1+\rho\,x+(\rho\,y)^2=\rho\,z$$

▶ **Problem 18** (UDC). UDC *is the following problem:*

$$\mathsf{UDC}\,(l:\mathscr{L}\,(\mathcal{V}^3)):=\exists\rho.\,\forall(x,y,z)\in l.\,\rho\ \textit{satisfies}\ (x,y,z)$$

▶ **Fact 19**. UDC *is undecidable.*

Fact 19 is mechanized by reducing from the general undecidability result for Diophantine constraints satisfiability by Larchey-Wendling and Forster [25]. Hence, our undecidability proof fundamentally relies on the general undecidability of Diophantine constraints satisfiability.

We can reduce from UDC to show UDPC undecidable. We only sketch the reduction.

▶ **Theorem 20**. $\mathsf{UDC}\preccurlyeq\mathsf{UDPC}$

Proof sketch. For each variable v appearing in the instance of UDC, have five new variables $v_i, 0\le i\le 4$. Then, for each constraint $1+x+y^2=z$, encode it using these new constraints:

$$(y_1,y_1)\wr(y_2,y_4),\quad (y_3,y_0)\wr(y_2,y_1),\quad (y_3,x_0)\wr(z_0,x_1)$$

The new variables v_i are thereby assigned the following values based on the old value of v:

i	0	1	2	3	4
v_i	v	$\frac{v^2+v}{2}$	v^2+v+1	v^2	$\frac{v_1^2+v_1}{2}$

◀

▶ **Theorem 21**. UDPC *is undecidable.*

Proof. By Theorem 20 and Fact 19. ◀

4 Undecidability of Validity

We now fix the concrete dyadic signature $\Sigma_{\wr}$, where the binary relation $\wr$ is the only symbol, and proceed to work within this signature unless explicitly mentioned.

4.1 Reducing from UDPC

For our reduction, we are given a constraints set $h : \mathscr{L}(\mathcal{V}^2 \times \mathcal{V}^2)$, and have to construct a formula $F^{\mathsf{VAL}}\, h$ such that $F^{\mathsf{VAL}}\, h$ is valid in all models if and only if h had a solution – formally, $\mathsf{UDPC}\, h \leftrightarrow \mathsf{VAL}\, (F^{\mathsf{VAL}}\, h)$. For this, we construct a first-order formalization of $\wr$, which later allows us to translate concrete constraints into FOL.

During the reflection step, we need to instantiate the proof that $F^{\mathsf{VAL}}\, h$ is valid with a specific model $\mathcal{M}_\wr$, which we develop now. Fix $D := \mathbb{N} + \mathbb{N}^2$ as the type of objects in $\mathcal{M}_\wr$. We can then define the interpretation.

▶ **Definition 22** (Interpretation of $\wr$ for $\mathcal{M}_\wr$).

l \ r	$y : \mathbb{N}$	$(c, d) : \mathbb{N}^2$
$x : \mathbb{N}$	$x = y$	$x = c$
$(a, b) : \mathbb{N}^2$	$y = b$	$(a, b) \wr (c, d)$

Since we develop our axioms based on this model, we consider it the *standard model* of the following theory. We understand $\wr$ as a binary relation on pairs, and also add the numbers these pairs are made up from as individuals. Our interpretation of $\wr$ then also allows describing the pair's components. The definition for $x \wr y$ with $x, y : \mathbb{N}$ allows discriminating pairs and numbers, since $\wr$ (on pairs) is irreflexive (Proposition 16).

We can then begin axiomatizing $\wr$, starting with a few shorthands:

$$N\, k := k \wr k \qquad P\, p\, l\, r := \neg N\, p \land N\, l \land N\, r \land l \wr p \land p \wr r$$

$$R\, a\, b\, c\, d := \exists pq. P\, p\, a\, b \land P\, q\, c\, d \land p \wr q$$

In the standard model, $N\, k$ iff k is a number, and $P\, k\, l\, r$ iff k is the pair (l, r). $R\, a\, b\, c\, d$ is satisfied iff $(a, b) \wr (c, d)$.

We can now formalize our axioms in first-order logic. Due to Theorem 14, the Base and weak Step axioms are sufficient for the purpose of mechanizing this reduction.

Additionally, since our original relation is built on the linear order of natural numbers, we need to mechanize enough properties of natural numbers to re-establish this linear order in our relation. Thus, we first add axioms characterizing the natural numbers.

$$A_1^{\mathsf{VAL}} := \forall k. N\, k \to \exists k'. R\, k\, \dot{0}\, k'\, \dot{0} \qquad A_2^{\mathsf{VAL}} := N\, \dot{0}$$

Axioms A_1^{VAL} and A_2^{VAL} postulate that $\dot{0}$ is a natural number, and that there are successors. The encoding of $S\, x = y$ is $(x, \dot{0}) \wr (y, \dot{0})$, implicitly encoding the Base axiom.

$$A_3^{\mathsf{VAL}} := \forall abcdb'c'd'. R\, a\, b'\, c'\, d' \land R\, d'\, b'\, d\, d' \land R\, b'\, \dot{0}\, b\, \dot{0} \land R\, c'\, \dot{0}\, c\, \dot{0} \to R\, a\, b\, c\, d$$

Axiom A_3^{VAL} formalizes the weak Step axiom.

Throughout these axioms, we have used $\dot{0}$ as if it was a nullary function symbol (i.e. a constant). In order to keep our signature minimal, we add this as an outermostly quantified variable to our formula. We call such a construction a *mock constant*.

We can now construct our reduction function:

$$\begin{aligned} F^{\mathsf{VAL}}, \mathit{code} &: \mathscr{L}\left(\mathcal{V}^2 \times \mathcal{V}^2\right) \to \mathcal{F} \\ F^{\mathsf{VAL}}\, h &:= \forall \dot{0}. A_1^{\mathsf{VAL}} \to A_2^{\mathsf{VAL}} \to A_3^{\mathsf{VAL}} \to \bigsqcup_{v \in \mathcal{V}(h)} \mathit{code}\, h \\ \mathit{code}\, [] &:= \top \\ \mathit{code}\, (((a, b), (c, d)) :: h) &:= R\, a\, b\, c\, d \land \mathit{code}\, h \end{aligned}$$

Besides ensuring the axioms hold, the reduction function encodes the *satisfaction condition* using one ∃-quantifier per variable in the constraint set h, thereby requiring that the model has a solution satisfying the first-order-encoded constraints.

▶ **Lemma 23** (Reflection). $\mathsf{VAL}\,(F^{\mathsf{VAL}}(h)) \to \mathsf{UDPC}\,h$

Proof. We have that $F^{\mathsf{VAL}}(h)$ is valid in all models, so it in particular is satisfied by $\mathcal{M}_\wr$, which satisfies A^{VAL}_{1-3}. The satisfaction condition ensures the model "knows" a solution, which we can extract since the interpretation of $\wr$ in $\mathcal{M}_\wr$ is faithful. ◀

For preservation, we are given a solution ρ satisfying h, and have to reason in an abstract model. By construction of F^{VAL}, we can assume the axioms A^{VAL}_{1-3} and must now instantiate a solution for the satisfaction condition. To do so, we first need to find the elements in our model corresponding to natural numbers. For this, we define a data structure called "chain":

▶ **Definition 24** (Chain). *A* chain *up to* $m : \mathbb{N}$ *is a function* $f : \mathbb{N} \to D$ *such that all of*
1. $f\,0 = \dot{0}$
2. *for all* $n < m$*, we have* $R\,(f\,n)\,\dot{0}\,(f\,(n+1))\,\dot{0}$

If $f\,n = d$*, then* d represents n.

Such a chain is just a partial function that maps elements of $\mathbb{N}$ to their representatives in the model.

Starting in Definition 24, we abuse notation by using first-order formulas and terms to denote properties in and elements of the model.

▶ **Proposition 25**. *Given a chain* f *up to* m *and an* $n \le m$*, we have* $N\,(f\,n)$.

Proof. For 0, use A^{VAL}_2. Otherwise, by definition of R. ◀

In order to give a proof of $M \vDash \exists_{v \in \mathcal{V}(h)}\, code\, h$, we need to first construct a chain:

▶ **Proposition 26** (Chain construction). *Given* m*, there exists a chain* f *up to* m.

Proof. Induction on m:
- $m = 0$: The chain is given by $f\,n := \dot{0}$. This satisfies both properties, using Proposition 25.
- $m = m' + 1$, where f' is a chain up to m' by induction. We apply A^{VAL}_1 to $f'\,m'$ and get d such that $R\,(f'\,m')\,0\,d\,0$. Our chain f up to m is chosen as $f'[m \mapsto d]$. Chain property 1 is shown by induction, and property 2 also is using Proposition 25, except for $n = m'$, where it is satisfied because A^{VAL}_1 gave us d already fulfilling this required property. ◀

We now use Proposition 26 to build a chain f up to $\max_{v \in \mathcal{V}(h)} \rho\, v$. Then, we can proceed to prove the satisfaction condition by instantiating as follows: For a variable v, chose $f\,(\rho\, v)$. The remaining goals are now of shape $R\,(f\,(\rho x))\,(f\,(\rho y))\,(f\,(\rho z))\,(f\,(\rho w))$, for all $((x, y), (z, w)) \in h$. Finally, this can be shown using another lemma:

▶ **Proposition 27** (Chain steps). *If* $(a, b) \wr (c, d)$*,* $a, b, c, d < m$ *and* f *is a chain up to* m*, then* $M \vDash R\,(f\,a)\,(f\,b)\,(f\,c)\,(f\,d)$.

Proof. Induction on the inductive characterization of $(a, b) \wr (c, d)$:
- $a + 1 = c, b = d = 0$: Since $a < m$ and f is a chain, we are done by the chain properties.
- We have $R\,(f\,a)\,(f\,b')\,(f\,c')\,(f\,d')$, $R\,(f\,d')\,(f\,b')\,(f\,d)\,(f\,d')$, $R\,(f\,b')\,(f\,\dot{0})\,(f\,b)\,(f\,\dot{0})$, and $R\,(f\,c')\,(f\,\dot{0})\,(f\,c)\,(f\,\dot{0})$ by induction as $a, b, c, d, b', c', d' < n$ holds. We can apply A^{VAL}_3 and are done. ◀

With this lemma, we can conclude the complete proof of $M \vDash F^{\mathsf{VAL}}\,h$ for an arbitrary M.

▶ **Lemma 28** (Preservation). $\mathsf{UDPC}\, h \to \mathsf{VAL}\,(F^{\mathsf{VAL}}(h))$

Proof. We have ρ, a solution to h, and the fact that our model fulfills the axioms A^{VAL}_{1-3}. Thus, we can build a chain f up to $\max_{v \in \mathcal{V}(h)} \rho\, v$ by Proposition 26. This chain allows us to instantiate the satisfaction condition, concretely using $f(\rho\, v)$ for given $v : \mathcal{V}$. We conclude by Proposition 27 for the remaining goals created by *code*. ◀

▶ **Theorem 29**. $\mathsf{UDPC} \preccurlyeq \mathsf{VAL}$ *restricted to dyadic formulas.*

Proof. F^{VAL} is a reduction function by Lemma 23 and Lemma 28. ◀

▶ **Theorem 30**. $\overline{\mathsf{UDPC}} \preccurlyeq \mathsf{SAT}$ *restricted to dyadic formulas.*

Proof. Using $F'\,\varphi := \neg(F^{\mathsf{VAL}}\,\varphi)$ as reduction function. ◀

4.2 Minimizing the Logical Fragment

Next, we adapt the just outlined reduction to not only witness the undecidability of validity over the minimal signature, but to also establish the undecidability over a[2] minimal set of logical operators, namely the forall-implicative fragment.

▶ **Definition 31** (Forall-implicative fragment). *A formula using only $\bot$, the logical connective $\to$, and the $\forall$-quantifier is within the* $(\forall, \to, \bot)$-fragment.

▶ **Definition 32** (Negation). *A formula in the $(\forall, \to, \bot)$-fragment not containing $\bot$ is within the* $(\forall, \to)$-fragment. *In general, a formula without $\bot$ is in a fragment* without negation.

We do so employing a mostly standard translation process, which combines ideas of both Gödel-Gentzen double negation translation [14, 13] and Friedman's A-translation [12], as outlined by Forster et al. [9]. Put simply, we apply a double-negation translation, and replace all uses of $\bot$ by another formula. The resulting formula then is within the $(\forall, \to)$-fragment.

This transformation has not yet been mechanized to hold in general, so we manually apply it to our concrete formula, and construct a new reduction with this reduced formula.

Further, the transformation usually introduces a new relation symbol for $\bot$, which is not an option here, as we want to keep the signature minimal. Instead, we slightly change our standard model, exploiting as-of-yet unused space. To adapt our reduction, we define new syntactic sugar, subsuming the previous definitions.

$$\bot_w := c_1 \not\approx c_2 \qquad \bot_s := \forall ab.\, a \not\approx b \qquad \neg_w \varphi := \varphi \to \bot_w$$
$$N\, k := k \not\approx k \qquad P\, p\, l\, r\, \psi := (N\, p \to \bot_s) \to N\, l \to N\, r \to l \not\approx p \to p \not\approx r \to \psi$$
$$R\, p\, q\, a\, b\, c\, d\, \psi := P\, p\, a\, b\, (P\, q\, c\, d\, (p \not\approx q \to \psi))$$

There are two versions of $\bot$: $\bot_s$ and $\bot_w$, where $\bot_w$ is the canonical replacement of falsity. $\bot_s$ is actually false in our standard model, whereas $\bot_w$ will not be, and this allows us to simplify verifying the reduction:

A canonical Friedman A-translation would just replace all occurrences of $\bot$ by $\bot_w$ and insert sufficient double negations. Since working in doubly negated contexts is often cumbersome, our reduction aims to avoid this as much as possible. Using $\bot_s$, which is actually equivalent to falsity in the standard model, helps, since it makes the elimination

[2] There is no canonical unique minimal set of logical operators. We use $(\forall, \to)$, following Forster et al. [9], while Gentzen [13] used $(\forall, \wedge)$. In either case, a quantifier and a binary connective is necessary.

lemmas not be doubly negated, thereby eliding a significant amount of double negations. For instance, we can show that in $\mathcal{M}$, $(N\,p \to \bot_s)$ implies that p is a pair, and not just that $\neg_w\neg_w(p \text{ is a pair})$.

Along with this, we also translate our axioms:

$$A_2^\bot := N\,0 \qquad A_3^\bot := \forall abcdb'c'd'p_1 \ldots p_8.\ R\,p_1\,p_2\,a\,b'\,c'\,d'\ (R\,p_3\,p_4\,d'\,b'\,d\,d'\ (R\,p_5\,p_6\,b'\,0\,b\,0\ (R\,p_7\,p_8\ c'\,0\,c\,0\ (\neg_w\forall p_9 p_{10}.R\,p_9\,p_{10}\,a\,b\,c\,d\,\bot_w))))$$

$$A_1^\bot := \forall k.\ N\,k \to \neg_w\forall pqk'.\ R\,p\,q\,k\,\dot{0}\,k'\,\dot{0}\,\bot_w$$

Axiom 3 becomes larger as we have changed the definition of R. Previously, this hid away two pairs each, which are now explicitly universally quantified. This again serves to make our reduction easier to verify. The full reduction function is as follows:

$$\begin{aligned} F^\bot, \mathit{code} &: \mathscr{L}\left(\mathcal{V}^2 \times \mathcal{V}^2\right) \to \mathcal{F} \\ F^\bot\,h &:= \forall\dot{0}\,c_1\,c_2.\ A_1^\bot \to A_2^\bot \to A_3^\bot \to \neg_w \bigvee_{v\in\mathcal{V}(h)} \mathit{code}\,h \\ \mathit{code}\,[] &:= \bot_w \\ \mathit{code}\,(((a,b),(c,d)) :: h) &:= \neg_w(\forall p_1p_2.R\,p_1\,p_2\,a\,b\,c\,d\,\bot_w) \to \mathit{code}\,h \end{aligned}$$

We need to slightly adjust our interpretation to make the Friedman translation work.

▶ **Definition 33** (Interpretation of $\wr$ for Friedman-translation).

l \ r	$y \in \mathbb{N}$	$(c,d) \in \mathbb{N}^2$
$x \in \mathbb{N}$	$x = y \vee \underline{(x = 0 \wedge y = 1 \wedge \mathsf{UDPC}\,h)}$	$x = c$
$(a,b) \in \mathbb{N}^2$	$y = b$	$(a,b) \wr (c,d)$

Here, the aforementioned encoding of reified falsity is underlined. The encoding stipulates that $0 \wr 1 \Leftrightarrow \mathsf{UDPC}\,h$. This becomes relevant during reduction reflection, as it allows us to "break out" of the double-negated context by instantiating the mock constants c_1, c_2 with $c_1 := 0, c_2 := 1$. Intuitively[3], when replacing $\bot$ with A when Friedman-translating φ, the resulting formula is equivalent to $\varphi \vee A$. Thus, by choosing $0 \wr 1$ as A, we are able to extract a solution as we did before, and if we are not, we still have our result.

▶ **Proposition 34** (Proposition 26 for the minimal fragment). *Let $m : \mathbb{N}$. To show $\bot_w$, it suffices to show $\forall f.(f \text{ is a chain up to } m) \to \bot_w$.*

▶ **Proposition 35** (Proposition 27 for the minimal fragment).
If $(a,b) \wr (c,d)$, $a,b,c,d < m$ and f is a chain up to m, then showing $M \vDash \bot_w$ requires showing $M \vDash \forall pq.R\,p\,q\,(f\,a)\,(f\,b)\,(f\,c)\,(f\,d)\,\bot_w$.

Proposition 34 is the double-negated version of Proposition 26. Instead of posing the existence of a chain f, we allow adding hypotheses when aiming to prove $\bot_w$, which is equivalent under double negation. Proposition 35 is translated similarly.

Proving these propositions is very technical, as one has to work in a doubly-negated context most of the time. As mentioned, the axioms and the syntactic sugar we defined before aim to minimize double negations, which explains most of the unorthodox constructions (e.g. having both $\bot_w$ and $\bot_s$). With these two lemmas, our reduction is complete.

[3] This intuition only holds in classical logic, but is still useful for conceptualizing the translation

▶ **Theorem 36.** UDPC $\preccurlyeq$ VAL *restricted to dyadic formulas over the* $(\forall, \to)$*-fragment.*

Proof. Similar to Theorem 29, except for the changes outlined in this chapter. ◀

▶ **Theorem 37.** $\overline{\text{UDPC}} \preccurlyeq$ SAT *restricted to dyadic formulas over* $(\forall, \to, \bot)$*-fragment.*

Proof. Using $F'\,\varphi := \neg(F\,\varphi)$ as reduction function. ◀

SAT is trivially decidable over the $(\forall, \to)$-fragment, since one can construct a model where everything is true. It thus becomes undecidable as soon as there is a single use of $\bot$.

4.3 Undecidability of Provability

We now turn towards the undecidability of the provability predicate PRV.

In a classical meta-theory, this would directly follow from completeness [15]. In our intuitionistic meta-theory, however, this does not trivially hold [10, 24], and we instead have to manually mechanize our reduction for the deduction system. The reduction function stays the same, but instead of reasoning in an abstract model, we prove that the formula is deducible in our deduction system. This mainly impacts the preservation step, as we can use soundness for the reflection step:

▶ **Lemma 38** (Reflection). $\text{PRV}\,(F^{\bot}(h)) \to \text{UDPC}\,h$.

Proof. Immediate using Theorem 36 and soundness of the deduction system. ◀

For preservation, the proof structure follows the previous proofs, while the low-level goals change to accommodate the object-level deduction. For brevity, we only show the new definition of a chain, as it demonstrates changes necessary to construct a syntactic proof in the first-order deduction system.

▶ **Definition 39** (Chain for provability). *A* proto-chain up to n *is a list on* $\mathcal{V}^3$ *of length* n*, containing triples* (m, l, r)*. Every proto-chain* c *has a* head number *head* c*, which is* $\dot{0}$ *for* $[]$ *and* m *for* $[(m, l, r), \ldots]$*. The* chain hypotheses *hyp* $n\,c : \mathscr{L}(\mathcal{F})$ *for a proto-chain* c *up to* n *are defined inductively:*

$$\begin{aligned} \mathit{hyp}\,0\,[] &:= [N\,\dot{0}] \\ \mathit{hyp}\,(n'+1)\,((m,l,r) :: cr) &:= N\,m :: (N\,l \to \bot_s) :: (N\,r \to \bot_s) \\ &\quad :: l \between (\mathit{head}\,cr) :: \dot{0} \between l :: r \between m :: \dot{0} \between r :: l \between r :: \mathit{hyp}\,n'\,cr \end{aligned}$$

The definition now strongly separates data (the objects in our chain) and hypotheses this data must satisfy. Compared to this, Definition 24 kept the data implicit: for a number $n : \mathbb{N}$, we had $m = f\,n$, and the proof that f is a chain at n. That property (being a chain) is defined by an existential quantifier, and so a proof of it contained two pairs l, r as witnesses. This was possible since Tarski models directly embed into the meta-logic, so first-order existentials are represented as meta-level existentials, which just are dependent pairs (truncated into $\mathbb{P}$). For provability, these objects are now explicitly stored as the chain entry (m, l, r), as working within the deduction system prevents us from building a data structure (like dependent pairs) containing both these objects and the proofs about them.

Proposition 34 now looks like this, for instance:

▶ **Proposition 40** (Proposition 34 for PRV).
Let $m : \mathbb{N}$ *and* $A : \mathscr{L}(\mathcal{F})$ *be a list of hypotheses including* $A^{\text{VAL}^{\bot}}_{1-3}$*. To construct a proof* $A \vdash \bot_w$*, it suffices to construct a proof* $(\mathit{hyp}\,m\,c) + A \vdash \bot_w$ *for all proto-chains* c *up to* m*.*

This theorem builds a proof for $\bot_w$ from another proof for $\bot_w$ with more assumptions. Specifically, it allows us to assume that there is some proto-chain, and to also assume that it actually fulfills the chain hypotheses. As mentioned, data (the variables defining the chain) and the hypotheses about this data are maintained separately, since the hypotheses are part of the object-level deduction, while the data is not. Proposition 35 and the other lemmas from before undergo similar changes, which we omit for brevity.

▶ **Lemma 41** (Preservation). $\mathsf{UDPC}\, h \to \mathsf{PRV}\,(F^{\bot}(h))$.

Proof. Using the reformulated variants of Proposition 34 and Proposition 35, as outlined above, following the approach of Lemma 28. ◀

With this, we have shown that validity, satisfiability, and provability are undecidable for the minimal signature and minimal logical fragment.

▶ **Theorem 42.** $\mathsf{UDPC} \preccurlyeq \mathsf{PRV}$ *restricted to dyadic formulas over the* $(\forall, \to)$*-fragment. Assuming* LEM*, this holds for* PRV_c*.*

Proof. $F^{\bot}$ is a reduction function by Lemmas 38 and 41. For PRV_c, we need LEM to establish soundness in Lemma 38. ◀

Related Decision Problems. From the undecidability of provability, we additionally get the undecidability of Kripke validity (KVAL) and Kripke satisfiability (KSAT). Since we do not work in Kripke models, we refrain from formally introducing them here, and refer to Forster et al. [9] and Herbelin and Lee [17].

▶ **Theorem 43.** $\mathsf{UDPC} \preccurlyeq \mathsf{KVAL}$ *restricted to dyadic formulas over the* $(\forall, \to)$*-fragment.*

▶ **Theorem 44.** $\overline{\mathsf{UDPC}} \preccurlyeq \mathsf{KSAT}$ *restricted to dyadic formulas over the* $(\forall, \to, \bot)$*-fragment.*

5 Undecidability of Finite Satisfiability

For finite satisfiability (FSAT), we need to construct a new reduction function. The axioms used so far allow us to construct infinitely many numbers, while the reduction requires us to transport arbitrary large solution assignments. Both of these become impossible when working with finite models, potentially requiring fragile upper bounds on model size. Additionally, only $\overline{\mathsf{UDPC}}$ many-one reduces to FVAL, as both are co-recursively enumerable. Thus, a more straightforward approach is a reduction from UDPC towards FSAT. Such reductions work "inversely" in that the axioms typically resemble elimination schemes, instead of constructors.[4] Our axioms then merely deconstruct larger values into smaller ones, thus side-stepping the issue of making the axioms work without conflicting with the finiteness of the model.

The following reduction often needs to decide whether an arbitrary first-order formula is satisfied in some finite model. Since we required decidable relation interpretations, we can construct a general decider, as noted in [23]:

[4] Explicitly noted and discussed in [23]

▶ **Proposition 45** (Decidability of finite satisfaction). *Given a fixed finite model $\mathcal{M}$ and a fixed environment $\rho : \mathcal{V} \to \mathcal{M}$, the predicate $\mathcal{M} \vDash_\rho \varphi$ on $\varphi : \mathcal{F}$ is decidable.*

Proof. All relation symbol interpretations of $\mathcal{M}$ are decidable. Quantified formulas are decidable since finite quantification is decidable and $\mathcal{M}$ is finite. ◀

For the axioms, we need first-order indistinguishably $\equiv$ in order to guard our elimination axioms against constructing a predecessor of $\dot{0}$, since such a predecessor does not exist in our standard model. We can encode it as $a \equiv b := \forall k.a \wr k \leftrightarrow b \wr k \land k \wr a \leftrightarrow k \wr b$. The axioms of Lemma 13 are formalized as follows:

$$
\begin{aligned}
A_1^{\mathsf{FSAT}} &:= \forall k'.\, N\, k' \to k' \not\equiv \dot{0} \to \exists k.\, (k, \dot{0}) \wr (k', \dot{0}) \\
A_2^{\mathsf{FSAT}} &:= \forall abcd.\, (a,b) \wr (c,d) \to b \not\equiv \dot{0} \to \\
&\qquad \exists b'c'd'.\, (b', \dot{0}) \wr (b, \dot{0}) \land (c', \dot{0}) \wr (c, \dot{0}) \land (a, b') \wr (c', d') \land (d', b') \wr (d, d') \land d' < d \\
A_3^{\mathsf{FSAT}} &:= \forall aa'd.\, (a, \dot{0}) \wr (a', d) \to d \equiv \dot{0}
\end{aligned}
$$

These axioms are based on the Base, strong Step, and Tieback laws respectively. We only need the "backwards" direction since we will only deconstruct a given relation.

These axioms alone are not sufficient to extract constraints solutions from a model. While these allow us to unpack a constraint into smaller constraints, the model might be cyclic, such that smaller constraints eventually unpack into themselves. To prevent this, we need to make the predecessor relation $(k, 0) \wr (k', 0)$ well-founded. While well-foundedness is not first-order expressible in general, it is possible to ensure that a relation on a finite model is well-founded:

▶ **Fact 46** (Well-founded relations for finite types). *Let D be a listable type and $\prec$ be a transitive, irreflexive relation on D. Then $\prec$ is well-founded.*

To make our predecessor relation well-founded, we need to construct its transitive closure. This is again not first-order expressible without adding a new relation symbol, which we can not do. Our solution is to exploit encoding space in the standard model in order to fit a new binary relation on numbers into the standard model. We define the following syntactic sugar, and add more axioms:

$$
\begin{aligned}
A_4^{\mathsf{FSAT}} &:= \forall lr.\, (l, \dot{0}) \wr (r, \dot{0}) \to l < r \land \forall k.\, k < r \to k \le l & \qquad a < b &:= a \le b \land a \not\equiv b \\
A_5^{\mathsf{FSAT}} &:= \forall abc.\, a < b \to b < c \to a < c & \qquad a \le b &:= N\, a \land N\, b \land a \wr b
\end{aligned}
$$

So $<$ becomes our transitive, irreflexive relation encompassing the predecessor relation. The actual reduction function can now be given. It also features an upper bound to ensure that we can build a sufficiently large anti-chain.

$$
\begin{aligned}
& F^{\mathsf{FSAT}}, \mathit{code} : \mathscr{L}(\mathcal{V}^2 \times \mathcal{V}^2) \to \mathcal{F} \\
& F^{\mathsf{FSAT}}\, h := \exists \dot{0} m.\, A_1^{\mathsf{FSAT}} \land A_2^{\mathsf{FSAT}} \land A_3^{\mathsf{FSAT}} \land A_4^{\mathsf{FSAT}} \land A_5^{\mathsf{FSAT}} \land \bigsqcap_{v \in \mathcal{V}(h)} \mathit{code}\, h \\
& \quad \mathit{code}\, [] := \top \\
& \quad \mathit{code}\, (((a,b),(c,d)) :: hs) := (a,b) \wr (c,d) \land \mathit{code}\, hs \land a, b, c, d \le m
\end{aligned}
$$

Another mock constant is needed: m is an upper bound on the model size. Since we are reducing from satisfiability, mock constants must be existentially quantified. Reducing from FSAT also "flips" the reduction and preservation steps. We now start with preservation, which now features our standard model.

▶ **Lemma 47** (Preservation). $\mathsf{UDPC}\, h \to \mathsf{FSAT}\,(F^{\mathsf{FSAT}}(h))$

Proof. Given a solution ρ to a constraints set h, let $m := 1 + \max_{v \in \mathcal{V}(h)} \rho\, v$. We denote by $\mathbb{N}_{\leq m}$ the type of natural numbers up to (and including) m. We set our finite domain $D := \mathbb{N}_{\leq m} + \mathbb{N}^2_{\leq m}$, so that each element of the domain is either a natural number not larger than m, or the pair of two such numbers. To finalize our model $\mathcal{M}_{\leq m}$, we define the interpretation of ⟅.

▶ **Definition 48** (Interpretation of ⟅ for FSAT).

l \ r	$y \in \mathbb{N}_{\leq m}$	$(c,d) \in \mathbb{N}^2_{\leq m}$
$x \in \mathbb{N}_{\leq m}$	$x \leq y$	$x = c$
$(a,b) \in \mathbb{N}^2_{\leq m}$	$y = b$	$(a,b) ⟅ (c,d)$

The model interpretation remains unchanged from the previous chapters, except for the top-left cell, which changes to $x \leq y$ from $x = y$. This is where we exploit the additional encoding space given to us by the fact last chapter (almost) only used the diagonal, where $x = y$. The interpretation of ⟅ is also decidable, as is required for finite models.

Showing that the satisfaction condition holds also is straightforward: Choose $\rho\, v$ for each v, using that ⟅ is faithfully interpreted. Each $\rho\, v$ is in $\mathbb{N}_{\leq m}$, since it is smaller than m by construction. Thus we have shown that $F^{\mathsf{FSAT}}(h)$ is finitely satisfied. ◀

For reflection, we need to extract a constraints collection solution from an arbitrary finite model $\mathcal{M} = (D, I)$ satisfying the axioms and the satisfaction condition. For this, we will construct an inverse notion of a chain, which now transports individuals of a model to natural numbers.

▶ **Definition 49** (Anti-chain). *A function $f : D \to \mathcal{O}(\mathbb{N})$ is called a* anti-chain up to $m : D$ representing $n : \mathbb{N}$ *iff all of*

1. $\forall d : D.\, d \leq m \Leftrightarrow f\, m \neq \emptyset$
2. $\forall k : \mathbb{N}.\, (\exists d : D. d \leq m \wedge f\, d = \lceil n \rceil) \Rightarrow k \leq n$
3. $f\, m = \lceil n \rceil \wedge f\, (\dot{0}) = \lceil 0 \rceil$
4. $\forall (d_l\, d_r : D)(k_l\, k_r : \mathbb{N}).\, f\, d_l = \lceil k_l \rceil \wedge f\, d_r = \lceil k_r \rceil \Rightarrow \big(S\, k_l = k_r \Leftrightarrow (d_l, \dot{0}) ⟅ (d_r, \dot{0})\big)$
5. $\forall d d'\, f.\, d \neq \emptyset \Rightarrow (f\, d = f\, d' \Leftrightarrow d \equiv d')$

We call m and n the upper bound of f*, and say that some m'* represents *some n' iff $f\, m' = \lceil n' \rceil$.*

▶ **Proposition 50** (Anti-chain construction). *Given $m : D$ such that $N\, m$, there are n, f such that f is an anti-chain up to m representing n.*

Proof. Well-founded induction using $<$ on m, due to Fact 46. We decide whether $m \equiv \dot{0}$ by Proposition 45:

- $m \equiv \dot{0}$. We choose as our anti-chain the function defined by $f\, \dot{0} = \lceil 0 \rceil$ and $f\, k = \emptyset$ otherwise. The chain properties are easily shown using decidability of $(\equiv)$.
- $m \not\equiv \dot{0}$. We know m has a predecessor m' by A_1^{FSAT}. Applying the induction hypothesis to m' yields f', an anti-chain representing m' up to n'. We choose $f'[k \mapsto \lceil n' + 1 \rceil]$ (i.e. pointwise updating f' at k) as our anti-chain. It is up to m representing $n' + 1$ by case distinctions, using the induction hypothesis and decidability of $(\equiv)$. ◀

Once we are able to construct an anti-chain, we are able to use it to extract solutions to the constraints encoded in h. For this, we show that $\wr$ on D transports to $\wr$ on $\mathbb{N}$ for the represented numbers.

▶ **Proposition 51** (Solution recovery). *If f is an anti-chain up to m representing n, and if $(a, b) \wr (c, d)$ where a, b, c, d are all $\leq m$, then we find a_r, b_r, c_r, d_r such that $f\, a = \lceil a_r \rceil$, $f\, b = \lceil b_r \rceil$, $f\, c = \lceil c_r \rceil$, $f\, d = \lceil d_r \rceil$, and $(a_r, b_r) \wr (c_r, d_r)$.*

Proof. Well-founded induction using $<$ on b with a, c, d quantified, using Fact 46.
We have $f\, a, \ldots f\, d \neq \emptyset$ by anti-chain property 1. Again decide whether $b \equiv \dot{0}$:

- $b \equiv 0$. In this case, by A_3^{FSAT}, we find $d \equiv 0$. So we are given $(a, 0) \wr (c, 0)$, which fits anti-chain property 4 of f. Since $(x, 0) \wr (x + 1, 0)$, we can conclude using properties 3 and 5.
- $b \not\equiv 0$. We can apply A_2^{FSAT}. We further apply the induction hypothesis to $(d', b') \wr (d, d')$ and $(a, b') \wr (c', d')$, which is possible especially since $d < d'$, which we needed to explicitly add to A_2^{FSAT}. The remainder is straightforward by the defining properties of $\wr$ and anti-chain property 4, similar to case 1. ◀

With this, our reduction is complete.

▶ **Lemma 52** (Reflection). $\mathsf{FSAT}(F^{\mathsf{FSAT}}(h)) \to \mathsf{UDPC}\, h$

Proof. We need to first build an anti-chain f up to m using Proposition 50, where m is existentially quantified in F. After this, we are able to extract the numbers making up the solution to h by looking at the elements encoded in the satisfaction condition, and looking up the represented numbers in f. Afterwards, it remains to show that these numbers actually are solutions to h, which follows from Proposition 51 and some minor auxiliary lemmas. ◀

▶ **Theorem 53.** $\mathsf{UDPC} \preccurlyeq \mathsf{FSAT}$ *restricted to dyadic formulas.*

To reduce towards finite validity, simply negate the reduction function used for FSAT.

▶ **Theorem 54.** $\overline{\mathsf{UDPC}} \preccurlyeq \mathsf{FVAL}$ *restricted to dyadic formulas.*

5.1 Minimizing the Logical Fragment for FSAT

We now reduce the logical fragment for FSAT. Using Proposition 45, we can construct and verify a translation converting our formulas into the $(\forall, \to, \bot)$-fragment:

▶ **Definition 55** (Double negation translation). $(\cdot)^N$ *translates formulas $\varphi : \mathcal{F}$ to their double-negated counterpart in the $(\forall, \to, \bot)$-fragment. For instance, $(\varphi \vee \psi)^N = \neg\varphi^N \to \neg\neg\psi^N$ and $(\exists x.\, \varphi)^N = \neg(\forall x.\, \neg\varphi)$. The other cases are defined similarly.*

▶ **Proposition 56.** *Given a finite model $\mathcal{M}$, an environment $\rho : \mathcal{V} \to \mathcal{M}$, and a formula φ, $(M \vDash_\rho \varphi) \Leftrightarrow (M \vDash_\rho (\varphi)^N)$.*

Proof. By induction on φ, using Proposition 45 for $\wedge, \vee, \exists$. ◀

This now strengthens our previous undecidability result to a small logical fragment.

▶ **Theorem 57.** FSAT *reduces to* FSAT *over the $(\forall, \to, \bot)$-fragment, and* FVAL *similarly.*

Proof. The function $(\cdot)^N$ fulfills the reduction properties by Proposition 56. ◀

6 Conclusion

6.1 Summary of Results

We have shown that many common decision problems of FOL are undecidable in their minimal case by constructing many-one reductions from UDPC or $\overline{\text{UDPC}}$.

▶ **Theorem 58** (Undecidability of FOL problems). *The following problems are undecidable for first-order formulas with a single binary relation:*

	Problem	*Fragment*	*By reduction*
FOL	VAL	$(\forall, \to)$	*Theorem 36*
	SAT	$(\forall, \to, \bot)$	*Theorem 37*
	PRV	$(\forall, \to)$	*Theorem 42*
	PRV_c	$(\forall, \to)$	*Theorem 42 assuming* LEM
KFOL	KVAL	$(\forall, \to)$	*Theorem 43*
	KSAT	$(\forall, \to, \bot)$	*Theorem 44*
FFOL	FSAT	$(\forall, \to, \bot)$	*Theorems 53 and 57*
	FVAL	$(\forall, \to, \bot)$	*Theorems 54 and 57*

These results are minimal, except for finite validity (FVAL), where eliminating $\bot$ might be possible. However, we have not mechanized this, because our current reduction for FVAL occupies a local optimum where there is no space remaining in the interpretation of the standard model. Such space is however necessary for a Friedman-like falsity elimination, as done in Section 4.2. KFOL denotes Kripke variants, see [17] for an overview.

▶ **Conjecture 59**. FVAL *restricted to dyadic formulas over the* $(\forall, \to)$*-fragment is undecidable.*

The dependence on LEM in Theorem 42 could potentially be eliminated by performing yet another double negation translation. Apart from this, no axioms are assumed.

Reducing to the particular minimal variants becomes feasible because the source problem UDPC is very easy to axiomatize in first-order logic. We consider UDPC a contribution, since it seems to be a useful source problem for compact undecidability reductions in general.

6.2 Comparison to Existing Work

Our results describe minimal undecidable fragments for formulas along two axes: by minimizing the signature, and by minimizing the logical fragment.

A third axis often considered in classical literature is the quantifier prefix. Already in his 1937 work [20], Kalmár not only proves the result about the minimal signature, but also minimizes the quantifier prefix of a formula in prenex normal form to Σ_4. Later work, e.g. by Kalmár [21] and others, strengthened this result. We do not consider minimizing the quantifier prefix, it remains open for future work one might consider.

Results not using Coq work in a classical meta-theory (denoted by FOL_c, which notably does not include intuitionistic PRV), where PRV_c, VAL, and SAT coincide. Further, some historical papers mentioned below presuppose the general undecidability of FOL or otherwise construct a reduction without explicitly aiming to prove any undecidability results.

All cited mechanized results are in Coq. As far as we are aware, there are no mechanizations of first-order undecidability in other proof assistants.

The main difference between our approach and other mechanized results in the literature is that we directly and compactly reduce into dyadic first-order logic, whereas others either need signature compression steps, or otherwise achieve this result in a less direct way (e.g.

Kirst and Hermes [22], who first mechanize meta-theory of ZF set theory). The price for this compactness is paid by starting at a Diophantine constraints-related problem. The undecidability of Diophantine constraints satisfiability was only shown in 1970 by Matiyasevich [27], building on work by Davis, Putnam, and Robinson [6]. Other mechanizations usually start at the Post correspondence problem (PCP) [29], whose undecidability is easier to prove.

Paper	Problems	Dyadic	Small Fragment	In Coq
Church 1936 [4], Turing 1936 [34]	FOL_c	×	×	×
Kalmár 1937 [20]	FOL_c	✓	×	×
Gentzen 1936, Gödel 1929 [13, 14]	FOL	×	$(\forall, \wedge, \neg)$	×
Forster et al. 2019 [9]	FOL, KFOL	×	$(\forall, \to)$	✓
Kirst and Hermes 2021 [22]	FOL	✓	×	✓
The present work	FOL, KFOL	✓	$(\forall, \to)$	✓
Trakhtenbrot 1950 [33]	FSAT	×	×	×
Kirst and Larchey-Wendling 2020 [23]	FSAT	✓	×	✓
The present work	FFOL	✓	$(\forall, \to, \bot)$	✓

6.3 Remarks on the Coq Mechanization

For the mechanization, we start with the PRV reduction outlined in Section 4.3. This is because showing PRV undecidable mostly suffices to show VAL, SAT, and the Kripke variants undecidable, as using soundness subsumes an explicit proof in an abstract model. Thus, Proposition 26 and Proposition 27 are mechanized for provability and not for validity, although the proof is similar. Also, we immediately start in the $(\forall, \to)$-fragment, since the undecidability results for this small fragment subsume those for the large fragment. A separate mechanization of Theorem 29 is included for comprehensibility.

Note that the notation used in Coq differs from the one presented in this paper. In particular, the problem UDPC is `H10UPC`, UDC is `H10UC` similarly. The defining relation $\wr$ of UDPC is called `h10upc_sem_direct`. For $\wr\!\!\wr$, we used `Pr` and sometimes `#` or `##`. For a complete symbol mapping, see the notation mappings at the start of each Coq file.

Following previous projects, we use de Brujin indices [7] to represent binding, which make formulas hard to read and write, especially when there are many nested quantifiers involved. Yet, this made it possible to encode the constraints of an UDPC instance into a first-order formula without having to worry about variable shadowing, as UDPC is also defined with natural numbers as variables in Coq. Thus, our reduction works by ensuring that there are as many free variables as the highest used index in the UDPC source instance.

Hostert et al. [19] developed tools aimed at simplifying reasoning with both de Bruijn indices and the abstract provability predicate. However, we found these to not be applicable, most importantly because our formulas and syntactic proofs feature unboundedly nested quantifiers and an unbound number of hypotheses.

On paper, we have different definitions of $\wr$, and switch between them freely. In Coq, we mostly use Definition 9, as it allows using `lia`, a tactic which automatically solves simple linear integer arithmetic goals. Use of `lia` could be avoided by using the inductive characterization more often. The only place where this characterization makes the proof simpler is Proposition 27.

Our mechanization of the undecidability of PRV and corollaries for the minimal case requires 1000 LoC, which improves upon the approx. 4000 LoC used by Kirst and Hermes [22] for a similar result. Our mechanization for FSAT takes 1200 lines. This also improves upon the results of Kirst and Larchey-Wendling [23], who required about 5000 LoC. In general,

this makes the mechanization of these results in the library much more approachable, as they can be read without following a long reduction chain. The less complicated reduction of Theorem 29, which just minimizes the signature while working within the large logical fragment, takes less than 300 LoC.

We already mentioned Diophantine constraints satisfiability as a large dependency of our mechanization. For comparison, the mechanization used in the library, due to Larchey-Wendling and Forster [25], is 12,000 lines long, and relies on the undecidability of PCP.

References

1 Wilhelm Ackermann. Beiträge zum Entscheidungsproblem der Mathematischen Logik. *Mathematische Annalen*, 112:419–432, 1936. doi:10.1007/BF01565424.

2 Andrej Bauer. First Steps in Synthetic Computability Theory. *Electronic Notes in Theoretical Computer Science*, 155:5–31, 2006. Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI). doi:10.1016/j.entcs.2005.11.049.

3 Paul Bernays. Beiträge Zur Reduktionstheorie des Logischen Entscheidungsproblems. *Journal of Symbolic Logic*, 2(2):84–85, 1937. doi:10.2307/2267374.

4 Alonzo Church. A note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1(1):40–41, 1936. doi:10.2307/2269326.

5 Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2):95–120, 1988. doi:10.1016/0890-5401(88)90005-3.

6 Martin Davis, Hilary Putnam, and Julia Robinson. The Decision Problem for Exponential Diophantine Equations. *Annals of Mathematics*, 74(3):425–436, 1961. doi:10.2307/1970289.

7 Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

8 Yannick Forster. *Computability in Constructive Type Theory*. PhD thesis, Saarland University, 2021. URL: https://ps.uni-saarland.de/~forster/thesis.php.

9 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*, CPP 2019, pages 38–51, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293880.3294091.

10 Yannick Forster, Dominik Kirst, and Dominik Wehr. Completeness Theorems for First-Order Logic Analysed in Constructive Type Theory: Extended Version. *Journal of Logic and Computation*, 31(1):112–151, January 2021. doi:10.1093/logcom/exaa073.

11 Yannick Forster, Dominique Larchey-Wendling, Andrej Dudenhefner, Edith Heiter, Dominik Kirst, Fabian Kunze, Gert Smolka, Simon Spies, Dominik Wehr, and Maximilian Wuttke. A Coq library of undecidable problems. In *CoqPL 2020 The Sixth International Workshop on Coq for Programming Languages*, 2020.

12 Harvey Friedman. Classically and intuitionistically provably recursive functions. In Gert H. Müller and Dana S. Scott, editors, *Higher Set Theory*, pages 21–27, Berlin, Heidelberg, 1978. Springer Berlin Heidelberg. doi:10.1007/BFb0103100.

13 Gerhard Gentzen. Die Widerspruchsfreiheit der reinen Zahlentheorie. *Mathematische Annalen*, 112:493–565, 1936. doi:10.1007/BF01565428.

14 Kurt Gödel. Zur intuitionistischen Arithmetik und Zahlentheorie – Ergebnisse eines Mathematischen Kolloquiums, 1928-1933.

15 Kurt Gödel. Die Vollständigkeit der Axiome des logischen Funktionenkalküls. *Monatshefte für Mathematik und Physik*, 37(1):349–360, December 1930. doi:10.1007/BF01696781.

16 Yuri Gurevich. The decision problem for the logic of predicates and of operations. *Algebra and Logic*, 8:160–174, May 1969. doi:10.1007/BF02306690.

17 Hugo Herbelin and Gyesik Lee. Formalizing Logical Metatheory – Semantical Cut-Elimination using Kripke Models for first-order Predicate Logic, 2014. URL: `https://formal.hknu.ac.kr/Kripke/`.

18 David Hilbert and Wilhelm Ackermann. *Grundzüge der Theoretischen Logik*. Springer Verlag, 1928. `doi:10.1007/978-3-662-00049-6`.

19 Johannes Hostert, Mark Koch, and Dominik Kirst. A Toolbox for Mechanised First-Order Logic. In *The Coq Workshop 2021*, 2021.

20 László Kalmár. Zurückführung des Entscheidungsproblems auf den Fall von Formeln mit einer einzigen, binären, Funktionsvariablen. *Compositio Mathematica*, 4:137–144, 1937. URL: `http://www.numdam.org/item/CM_1937__4__137_0/`.

21 László Kalmár. On the reduction of the decision problem. First paper. Ackermann prefix, a single binary predicate. *Journal of Symbolic Logic*, 4(1):1–9, 1939. `doi:10.2307/2266211`.

22 Dominik Kirst and Marc Hermes. Synthetic Undecidability and Incompleteness of First-Order Axiom Systems in Coq. In *Interactive Theorem Proving - 12th International Conference, ITP 2021, Rome, Italy*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ITP.2021.23`.

23 Dominik Kirst and Dominique Larchey-Wendling. Trakhtenbrot's Theorem in Coq. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 79–96, Cham, 2020. Springer International Publishing. `doi:10.1007/978-3-030-51054-1_5`.

24 Georg Kreisel. On weak completeness of intuitionistic predicate logic. *Journal of Symbolic Logic*, 27(2):139–158, 1962. `doi:10.2307/2964110`.

25 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, June 24-30, 2019, Dortmund, Germany*. 4th International Conference on Formal Structures for Computation and Deduction, 2019. `doi:10.4230/LIPIcs.FSCD.2019.27`.

26 Leopold Löwenheim. Über Möglichkeiten im Relativkalkül. *Mathematische Annalen*, 76:447–470, 1915. `doi:10.1007/BF01458217`.

27 Yuri V. Matiyasevich. Enumerable sets are Diophantine. *Doklady Akademii Nauk SSSR*, 191:279–282, 1970. `doi:10.1142/9789812564894_0013`.

28 Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In *International Conference on Mathematical Foundations of Programming Semantics*, pages 209–228. Springer, 1989.

29 Emil L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–268, 1946.

30 Fred Richman. Church's thesis without tears. *Journal of Symbolic Logic*, 48(3):797–803, 1983. `doi:10.2307/2273473`.

31 Alfred Tarski. I: A General Method in Proofs of Undecidability. In Alfred Tarski, editor, *Undecidable Theories*, volume 13 of *Studies in Logic and the Foundations of Mathematics*, pages 1–34. Elsevier, 1953. `doi:10.1016/S0049-237X(09)70292-7`.

32 The Coq Development Team. The Coq Proof Assistant, January 2021. `doi:10.5281/zenodo.4501022`.

33 Boris Trakhtenbrot. The Impossibility of an Algorithm for the Decidability Problem on Finite Classes. In *Proceedings of the USSR Academy of Sciences*, 1950.

34 Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936. `doi:10.1112/plms/s2-42.1.230`.

Taming an Authoritative Armv8 ISA Specification: L3 Validation and CakeML Compiler Verification

Hrutvik Kanabar
University of Kent, UK

Anthony C. J. Fox
Arm Limited, Cambridge, UK

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

Abstract

Machine-readable specifications for the Armv8 instruction set architecture have become publicly available as part of Arm's release processes, providing an official and unambiguous source of truth for the semantics of Arm instructions. To date, compiler and machine code verification efforts have made use of unofficial *theorem-proving-friendly* specifications of Armv8, e.g. CakeML uses an L3-based specification. The validity of these verification efforts hinges upon their unofficial ISA specifications being *valid* with respect to the official Arm specification.

Leveraging the Sail language ecosystem, we bridge this validation gap by formally verifying that an L3-based specification simulates the official Arm specification using the HOL4 interactive theorem prover. We exercise this simulation by proving a novel compiler correctness result for CakeML with respect to Arm's official specification of the Armv8.6 A-class instruction set.

2012 ACM Subject Classification Software and its engineering → Software verification; Hardware → Theorem proving and SAT solving; Software and its engineering → Domain specific languages

Keywords and phrases Compiler verification, ISA specification, HOL4, interactive theorem proving

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.20

Supplementary Material Specifications and proofs in the HOL4 interactive theorem prover.
Software (ISA specifications): `https://github.com/HOL-Theorem-Prover/armv8.6-asl-snapshot`
Software (proofs, HOL4 repository): `https://github.com/HOL-Theorem-Prover/HOL/pull/981`
Software (proofs, CakeML repository): `https://github.com/CakeML/cakeml/pull/858`

Funding *Hrutvik Kanabar*: UK Research Institute in Verified Trustworthy Software Systems (VeTSS).
Magnus O. Myreen: Swedish Foundation for Strategic Research.

1 Introduction

Rigorously verifying the behaviour of software requires faithful modelling of the hardware on which it runs. Instruction set architectures (ISAs) provide a convenient abstraction of hardware behaviour: if hardware manufacturers ensure a processor adheres to the ISA it implements, it can be viewed by software as a machine that executes exactly the specification of that ISA.

Instruction set specifications have been used in a wide range of theorem proving projects [3, 9, 11, 18, 25, 29, 32, 40, 52, 54, 55], and much work has gone into using DSLs for the rigorous engineering of accurate ISA specifications [1, 15, 45]. Specifications of the Armv8 architecture are available in three such DSLs: L3 [15], ASL [44], and Sail [2]. ASL emerged in 2011, building on the pseudocode language used in Arm documentation since the late 1990s. L3 and Sail were conceived in academia concurrently, with differing objectives; L3 was implemented in 2011 ahead of Sail.

Though superficially similar, these three languages have distinct design goals:

- L3 supports interactive theorem proving, so all L3 specifications can be reliably extracted to valid HOL4 and Isabelle/HOL, producing tractable and useable (idiomatic) definitions.
- ASL is developed within Arm, with broad design objectives (Section 2). Fully-automated verification and testing are prioritised (model-checking, SMT verification, and simulation). Arm-internal processes produce ASL specifications for Arm ISAs and subject them to rigorous internal testing to ensure full architecture compliance [43, 44, 53]. They are then released publicly, allowing users to examine and even run them.
- Sail was initially designed for use in concurrency tools, but it is now one-size-fits-all: it supports many ISAs, and diverse extraction targets such as theorem proving (interactive and automated), symbolic evaluation, simulation, and documentation. It is therefore more ambitious and featureful than both ASL and L3, with a significant ecosystem of tools. Conversely, it is less directly connected to theorem proving backends than L3.

Sail's ecosystem includes an ASL frontend to translate official ASL specifications into Sail, and a HOL4 backend to translate Sail specifications into HOL4. Put together, these provide a pathway from official ASL specifications to HOL4.

The HOL4 specification of Armv8 derived from this process is one of the largest (488 kLoC) and most complicated known ISA specifications in a theorem prover, matched only by other specifications produced from official Arm ASL via Sail.

However, the extensive Arm-internal validation of ASL specifications provides the closest approximation to ground truth for semantics of the Arm ISA. To the best of our knowledge, there is currently no way to obtain a more faithful Armv8 specification in an interactive theorem prover.

Therefore, we prove once and for all that the L3 Armv8 specification simulates the ASL-derived one, allowing verification efforts to enjoy the ease-of-use of the former while retaining the faithful modelling of the latter. Future work can avoid navigating the complexity of the ASL-derived specification, and other users of the L3 specification benefit from its validation against official ASL. We demonstrate our approach by using this simulation result to prove a novel compiler correctness theorem for the CakeML compiler.

Contributions

We make the following contributions:

- We demonstrate that official Armv8 ISA specifications can be used as reference semantics in interactive proof, through translation to HOL4 via Sail (Section 3) and interactive abstraction to a more theorem-prover-friendly specification.
- We validate the L3 specification for Armv8 by proving it simulates the ASL-derived specification (Section 4.4). Previously this specification had not been rigorously validated.
- We leverage these two results to prove the *first compiler correctness result with respect to an official Arm ISA specification* (Section 5).

Our work is open-source and has been integrated into the HOL4 and CakeML public repositories for future re-use.

This paper also addresses a number of questions about the HOL4 specification of Armv8 derived from ASL via Sail:

- How does it differ from the L3-derived specification? (Section 4.2)
- How useable are its semantics of instructions within the theorem prover? (Section 4.3)
- What can be said about its trustworthiness, particularly in comparison to the L3 specification? (Section 6)
- How could we better adapt it for use in the theorem prover? (Section 6)

2 Background

This paper draws together prior research and industrial work on *architecture specification languages*: DSLs used to define behaviours of ISAs.

ISAs are extremely complex, so their documentation is sizeable – running into thousands of pages and covering topics such as: processor state (operating mode, registers, and memory); instruction encoding, semantics, and assembly syntax; memory models, memory protection (virtualisation), memory attributes, and caching; synchronisation and semaphores; debug, trace, and monitoring; interrupts, timers, and exceptions.

Although machine-readable ISA specifications cannot cover *all* aspects of ISAs comprehensively, they can provide a working reference semantics for machine code. There are many benefits to using machine-readable specifications, including:

- Avoidance of ambiguities and errors detected by parsing and type-checking;
- Robust validation paths through simulation-based testing;
- Formal verification of processor designs through model-checking [21, 53, 55];
- Support for other formal modelling and verification activities, such as work on memory and concurrency models [12,13,22,42], architecture security [5,8,40,54], OS and hypervisor verification [6, 23, 24, 28, 30], compiler and runtime verification [29, 34, 36, 37], and machine code verification [3, 16, 19, 31].

The domain of architecture specification lends itself to first-order, imperative languages with some common features. Scattered functions define each of their clauses separately: encoding, decoding, and execution behaviour of a particular instruction can be grouped together, enhancing readability and modularity. Strong typing enables early error-detection, and type inference reduces the need for type annotations.

ASL. ASL is the Arm-internal architecture specification language. Its initial objective was to reduce errors in the informal pseudocode found in Arm documentation by providing parsing and type-checking [45,50]. The language has since evolved and is now in extensive use within Arm. Its type system supports lightweight dependent typing: bit vector widths can depend on values, but bounds-safe bit vector accesses are not enforced. A built-in "undefined" value models architecturally unknown values, and an exception-handling mechanism streamlines specification of error cases.

Sail. The Sail language [2] and ecosystem is actively developed by the Rigorous Engineering of Mainstream Systems (REMS) project. The language effectively supports a superset of ASL to permit automatic translation from ASL [1]. A type-and-effect system ensures memory and register accesses are visible at the type level, and the lightweight dependent types of Sail are more expressive than those of ASL. In particular, bit vector accesses are statically bounds-checked, necessitating a flow-sensitive type system: bit vector width constraints are propagated according to program flow and discharged by the Z3 SMT solver.

Sail's ecosystem provides toolchains to translate from ASL to Sail, and from Sail to specifications in Coq and Lem, executable simulators in OCaml and C, and SMT formats. Lem in turn is a lightweight language for engineering reusable semantic models, inspired by both functional programming languages and proof assistants [33, 41]. Its ecosystem provides translations to HOL4 and Isabelle/HOL, amongst others.

L3. L3 [15] is designed to manage the complexity of writing ISA specifications for use in theorem provers: previous work constructed verbose specifications directly in HOL4 [14, 17]. L3 targets HOL4 and Isabelle/HOL directly (rather than via Lem), and has a simpler type system than ASL and Sail, supporting bit vector width restrictions on declared function arguments only. This tight coupling with its backends produces streamlined definitions.

L3's extensive automation minimises the effort of specification-writing and maximises usability within its prover backends. Key features include: prioritisation of abstract syntax for instructions; simultaneous declaration of instruction syntax, encoding, and semantics; symmetric syntax for register reads/writes; and extensibly declarable architectural state.

Unofficial L3 specifications have been produced for many different architectures: Armv4 through to Armv8 (the latter AArch64 mode only), MIPS, x86 (core 64-bit mode instructions only), and RISC-V. The Armv7 specification in particular has been extensively validated against Arm hardware. To keep generated prover specifications idiomatic, L3 Arm specifications re-factor official ASL ones in certain key areas, e.g. preferring bit vectors to integers, sharing common logic where possible, and manually defining efficient instruction decoders.

3 Generating a HOL4 specification for Armv8

Our first step in proving against a HOL4 specification of Armv8 is to generate the specification from official ASL. We rely on the Sail ecosystem, but its design choices have important implications for the resulting specification. In this section we review the extraction process in more depth to aid reproducibility and better inform the rest of the paper. Our generated HOL4 specification is available in a public repository.

The Sail ecosystem is under active development, and we believe we are the first to use a Sail-generated ISA specification for interactive proof in HOL4. The diagram below depicts the pipeline for translation from ASL to HOL4. At present, the process requires some manual intervention, and we are indebted to the Sail developers for helping navigate this space. We examine each stage in turn in the upcoming subsections.

$$\text{ASL} \xrightarrow[\text{Section 3.1}]{\texttt{asl\_to\_sail}} \text{Sail} \xrightarrow[\text{Section 3.2}]{\texttt{sail -lem}} \text{Lem} \xrightarrow[\text{Section 3.3}]{\texttt{lem -hol}} \text{HOL4}$$

3.1 ASL to Sail

The tool `asl_to_sail`[1] translates ASL specifications into Sail. It relies on ASLi ("ASL interpreter") [46] for parsing and type-checking of ASL. The MRA tools collection [47] can be used to extract the input ASL from public, XML-format specifications released by Arm. However, we simply use an ASL specification provided by Arm.

As Sail can be considered a superset of ASL, the translation is relatively naïve. Some optimisations are made, e.g. mutable assignments are turned into immutable let-bindings where possible. However, Sail's richer type system does not accept all ASL programs, so a degree of interactive patching is required. In these cases, `asl_to_sail` halts to request a patch, displaying the original ASL, generated Sail, and the failed typing derivation. The changes required are often straightforward restrictions to permit inference of tighter typing constraints. For example, lifting subexpressions to immutable let-bindings, or specialising

[1] `https://github.com/rems-project/asl_to_sail`

type signatures with effect annotations or bit vector width restrictions. However, Sail's type derivation output can be difficult to understand without in-depth knowledge of the typechecker.

We are grateful to Arm Limited and the Sail developers for providing an Armv8.6 A-class ASL specification and the necessary patches to translate it.

3.2 Sail to Lem

Translation from Sail to Lem is more involved: Lem mirrors its HOL4 and Isabelle/HOL backends, so imperative must become functional, and lightweight dependent types must become simple types. We highlight some key aspects of the translation here.

State/exception/non-determinism monad. Sail ships with hand-written Lem libraries encapsulating its built-in types and operations, building on the libraries that ship with Lem itself. These include the implementation of a monad designed to represent imperative, effectful Sail code. During translation, Sail specifications are converted to A-normal form to make explicit the calculation of intermediate values, and embedded into a state/exception/non-determinism monad of the following type (where σ is the state type, α the return type, and ϵ the exception type), with standard return, bind, and exception-handling operators:

$$\mathsf{M}\ \alpha\ \epsilon\ \sigma \stackrel{\mathsf{def}}{=} \sigma \longrightarrow \mathcal{P}\,(\mathsf{Result}\ \alpha\ \epsilon \times \sigma) \qquad\qquad \mathsf{Result}\ \alpha\ \epsilon ::= \mathsf{Value}\ \alpha \ \mid\ \mathsf{Ex}\ \epsilon\,.$$

Non-determinism is modelled by using a set of possible outputs, rather than a single one. Imperative early-return statements are modelled by extending the exception type to a sum, and throwing the early-return value as an exception. The sum type distinguishes between early-return "exceptions" and actual exceptions. All early-return functions are then wrapped with a monadic operator which embeds them back into the original monad.

Data representation. Users can choose to represent bit vectors as lists of three-value logic "trits" or machine-words from default Lem libraries. The Lem libraries for Sail define a typeclass for bit vectors, with trit-list and machine-word instantiations allowing easy switching between the two. There is a key trade-off here: trit-lists are a simpler extraction target, but require extra reasoning about bit vector widths (i.e. list lengths); machine-words have type-backed widths but require further processing (see below) to target Lem's simply-typed setting. We use the machine-word representation: this has stronger library support and more efficient call-by-value evaluation procedures within HOL4.

Sail can refer to registers by reference, so its Lem libraries define a typeclass for register references. Each register is defined by an instance of this typeclass, containing a canonical name and functions to read from/write to the register in the processor state.

Monomorphisation. Sail functions over bit vectors can be both dependently-typed and polymorphic, which is incompatible with Lem's simple types. Arm specifications make heavy use of assertions and case splits on bit widths, producing many dependently-typed functions.

Sail attempts to partially monomorphise such functions: each call-site of the function could in theory produce a width-specialised definition in Lem. The approach is inspired partly by prior work translating ASL to Verilog [48]: case splits are introduced until bit vector widths can remain constant throughout a function, and constant propagation determines these widths. Full monomorphisation is not required as Lem permits polymorphism over bit vector widths (functions over bit vectors can be width-agnostic). If a function can

be case-split to maintain consistent bit widths over each case, the various cases can be recombined into a single polymorphic definition. Bit vector extension operations are inserted as necessary to cast types without changing values, and type signatures are simplified until Lem-compatible. Bit vector slicing operations, which rely heavily on dependent typing, are converted into masking operations wherever possible.

For example, consider a Sail function which accepts any bit vector of width divisible by 8, and returns its length in bytes (of type `forall 'n. bits('n * 8) -> int`). Monomorphisation could produce multiple versions of this function, each with a type specialised to its call-site argument (e.g. of types `bits({8,16,32}) -> int`). However since each specialised version treats the input bit vector uniformly, a single polymorphic definition (of type α word $\rightarrow$ int) suffices. Note that the resulting function accepts bit vectors of widths not divisible by 8, where the original Sail function does not. However, Sail's typechecking guarantees it will never receive such inputs.

This monomorphisation pass can fail, or produce invalid Lem. While Sail remains under active development, the class of specifications which translate error-free is a moving target.

Other code transformations. Scattered functions are collected into single, monolithic functions, with clauses ordered by appearance. Each function clause may be guarded in Sail, so the guards are converted to if-statements.

Both ASL and Sail support an "undefined" built-in, used extensively in Arm specifications to model architecturally unknown values. Lem must explicitly implement this built-in: for each declared type in a specification, Sail automatically generates a function to produce the corresponding undefined value.

Sail unrolls recursive functions whose recursion depth can be determined, removing the need for some termination proofs in theorem prover backends. Users can also manually tweak parts of a specification by providing alternative function implementations to splice in. We make use of this splicing feature to modify our specification (Section 4.1).

3.3 Lem to HOL4

Lem straightforwardly translates to HOL4, though the raw specifications are not human-friendly until parsed/pretty-printed by HOL4: all expressions are type-annotated and fully-bracketed. HOL4 libraries for Sail are automatically translated from the corresponding Lem libraries, and Lem's simple typeclasses are represented as record types.

Some manual intervention is required for well-foundedness checking: Lem automatically generates simple well-foundedness proofs, but some Sail library functions are well-founded for non-trivial reasons. However, naïve definitions for pure and monadic while-looping constructs are not well-founded. We have redefined these functions correctly, deriving theorems showing their behaviour is as originally intended.

We also corrected a minor HOL4 usability issue when working with this substantial specification. We introduced a syntax for let-declarations within monads to enable clear printing of definitions which mix monadic and pure assignments. This permitted interactive inspection of the translated specification.

4 Using the specification for semantics of machine code

We now consider our verification work with respect to the detailed specification we have generated from official ASL. Several previous efforts have used such specifications to prove architectural properties [8, 40, 54, 55]. Instead, we use our HOL4 specification as a semantics

for Arm machine code. In particular, we are interested in proofs of *semantics preservation*, equating the semantics of Arm machine code programs with those of other programs (likely in another language). This is exactly the class of proofs for which L3 and its unofficial specifications were designed.

In this section we modify our specification to adapt it to this goal (Section 4.1), before examining the result more closely (Sections 4.2 and 4.3). Finally, we use an L3 specification to simulate our ASL-derived one, allowing re-use of L3 machinery to reason about ASL-derived semantics of Arm machine code (Section 4.4).

4.1 Modifications to the specification

During translation from ASL to HOL4, we make a number of modifications to our specification. Both original and modified HOL4 models are available in a public repository. We justify our changes here, omitting one minor modification irrelevant to our presentation, that is, disabling a testing harness.

Monad types. We remove the set-based non-determinism from the state/exception/non-determinism monad (Section 3.2), preferring to use HOL4's Hilbert choice operator to express unknown values. This is more idiomatic, and streamlines interactive proof.

More precisely, set-based non-determinism is not compatible with symbolic evaluation. For example, an undefined 64-bit bit vector in Sail is translated to a set of all 2^{64} possible 64-bit bit vectors in HOL4, which is intractable to evaluate. To generate undefined values for enumerated types, Sail libraries create an undefined bit-list of appropriate length and cast it to a natural number, which indexes into a list of all possible elements of the type. This too is more cleanly expressed as a Hilbert choice.

We must be careful with manual changes in a high-fidelity specification. However, our modifications are conservative: we change only the monad implementation in the hand-written Lem libraries for Sail. This gives us confidence in their validity.

Address translation. Address translation is a detail orthogonal to many proofs of semantics preservation: if we assume it is correctly implemented, we can view it as a simple map which abstracts physical memory to virtual memory. This is in keeping with L3 specifications and other specifications modelling the semantics of machine code [11, 56].

We therefore stub out address translation functions to express an identity mapping between virtual and physical memory. However, in Armv8 AArch64 virtual addresses are 64-bit and physical addresses up to 52-bit. Sail's user-splicing feature (Section 3.2) cannot modify types, so we manually modify the specification to convert the type of physical addresses. Again, we keep changes conservative to maintain trust in the specification: we are guided entirely by the Sail typechecker.

4.2 Examining the specification

Table 1 shows metrics taken throughout the extraction processes of L3 and ASL specifications to HOL4. The difference here is clear – to the best of our knowledge, the ASL-derived HOL4 specification is one of the most complex, unwieldy specifications to be used in interactive proof. Other points of comparison include the 0.59×10^6 character Sail specification of RISC-V [2] (3.6×10^6 characters in raw HOL4), and the 1.7×10^6 character ACL2 specification of x86 [20]. We examine these stark differences in more depth.

Table 1 Metrics for the extraction processes of L3 and ASL specifications via the L3 and Sail ecosystems respectively. Character counts in HOL4 are shown both as extracted (raw) and after pretty-printing to 80 columns. Timings taken using Intel® Xeon® E-2186G and 64 GB RAM.

Original specification	No. non-whitespace characters / 10^6					Size / kLoC		Total time to extract	HOL4 build time
	Source	*Sail*	*Lem*	*Raw HOL4*	*HOL4*	*Source*	*Raw HOL4*		
L3	0.053	–	–	0.20	0.070	2.4	8.5	1 s	$<$ 30 s
ASL	4.2	7.4	19.9	26.7	12.2	168	488	$\sim$ 2 hrs	$\sim$ 3 hrs

Why is the ASL specification so much larger than its peers? It covers more of its intended ISA, modelling most modes/instruction sets, where other specifications tend to formalise a particular mode of operation. For example: the L3 specification covers only AArch64 mode, and also omits vector (SIMD) and floating-point instructions; until recently [10] the ACL2 specification of x86 only covered 64-bit mode. L3 specifications model mostly user-level code, omitting most system registers/instructions.

ASL specifications have differing goals to their peers: primarily they provide documentation, focussing on thoroughness rather than simplicity. Other efforts focus on verification: conciseness and usability are primary goals. For example, the ASL specification defines IEEE floating-point semantics from scratch to ensure faithful behaviour, whereas other efforts outsource to libraries.

Why does extraction to HOL4 bloat the ASL specification? Representing sequential, imperative code monadically adds considerable bloat due to conversion to A-normal form and instrumentation with explicit monadic operations. Instead, L3 keeps definitions pure wherever possible (looping constructs remain monadic).

Various translation artefacts are verbose, for example: succinct bit vector slicing in Sail is often converted to masking; Sail's undefined built-in primitive must be implemented for each declared type; registers passed by reference in Sail require explicit definitions in HOL4.

L3 couples tightly with prover backends, using relatively simple types and HOL-flavoured constructs to provide an extremely sophisticated syntactic sugar for higher-order logic. Sail is more general-purpose, and not well-optimised for use with HOL4: bit-vector built-ins must be realised in libraries, and these often re-implement HOL4 operations unnecessarily and non-idiomatically (for example, see Example 1).

Why are extraction and build times so long for the ASL pipeline? Naturally, a significantly larger specification takes longer to extract and build. However, the increase is not proportional to size alone.

The ASL specification must be type-checked multiple times during extraction: in ASL, in Sail, and in Lem. Monomorphisation is a complex process, and lightweight dependent types in ASL/Sail necessitate heavy use of Z3 to discharge constraints.

Sail produces concrete HOL4 syntax via Lem, which must be parsed and type-checked in HOL4. Instead, L3 produces ML constructors which stitch together its definitions. Larger definitions built from Sail's scattered functions are particularly slow, including the decoder. By contrast, L3's decoder is manually defined to minimise its footprint; it tests opcodes in an order which avoids ambiguities from overlapping opcode spaces. The ASL-derived decoder is forced to implement additional machinery to disambiguate opcodes instead.

▶ **Example 1.**

```
l3_HighestSetBit x ≝
  if x = 0w then −1
  else w2i (word_log2 x)
```

```
asl_HighestSetBit x ≝
  catch_early_returnS
    (let
       loop_i_lower = 0;
       loop_i_upper = n2i (word_len x) − 1
     in
       do
         foreachS (index_list loop_i_upper loop_i_lower (−1)) ()
           (λ i unit_var.
              if vec_of_bits [access_vec_dec x i] = 1w then
                early_returnS i
              else returnS ());
         returnS (−1)
       od)
```

4.3 Working with the specification

We can now evaluate the practicality of our ASL-derived specification for interactive proofs of semantics preservation. Unfortunately, there are some significant obstacles.

Opaqueness to inspection and interaction. The ASL-derived specification does not implement an AST for Arm instructions, in contrast to the L3 specification. Users must work directly with opcodes or manually define an AST.

Non-idiomatic data manipulations and explicit monadic operations obfuscate high-level semantics. These are awkward and tedious in interactive proofs, which must step over each monadic operation. L3 instead prefers HOL4's let-declarations, mostly removing monadic sequencing.

Monolithic HOL4 functions coalesced from ASL's scattered functions are unwieldy, e.g. the 23 kLoC decoding/execution function DecodeA64 (for which each instruction implements a clause). Examining the semantics of a particular instruction is therefore challenging.

ASL specifications and Sail libraries use many auxiliary functions: many steps of definition expansion are required to examine or use intended semantics.

Opaqueness to automated evaluation. Non-idiomatic bit vector operations have poor evaluation support in HOL4, often re-implementing functionality in libraries shipped with HOL4. Some operations even convert machine-word operands to bit-lists, perform operations on bit-lists, and convert back. In these cases, permeative book-keeping of list lengths complicates reasoning. Integers are used throughout even when known to be positive, despite the comparative ease-of-use of natural numbers.

Consider the L3-derived (left) and ASL-derived (right) HOL4 definitions in Example 1, which determine the index of the highest set bit of input bit vector x. The L3-derived definition uses HOL4 library definitions to convert the base-2 logarithm of the input word to an integer (w2i). The ASL-derived definition is more computational: it examines each bit from high to low, returning early if any is set. However, this intended definition is obfuscated:

▶ **Definition 2.**

```
l3_models_asl opcode ≝
  Decode opcode ≠ Unallocated ∧
  ∀ l3 asl l3'.
    state_rel l3 asl ∧ asl_sys_regs_ok asl ∧ Run (Decode opcode) l3 = l3' ∧
    l3'.exception = NoException ⇒
    ∃ v asl'.
      ExecA64 opcode asl = (Value v,asl') ∧ state_rel l3' asl' ∧
      asl_sys_regs_ok asl'

l3_models_asl_instr instr ≝
  ∃ opcode. Encode instr = ARM8 opcode ∧ l3_models_asl opcode
```

the early return necessitates embedding within the Sail monad and use of early-return monadic operations (catch_early_returnS, early_returnS); explicit monadic operations are used for looping and sequencing (made more palatable here by do notation); integer loop variables are used over natural numbers (n2i casts from natural to integer); custom bit vector operations from Sail libraries are used (vec_of_bits and access_vec_dec). The definitions of these last operations involve a large number of auxiliary functions: they convert x to a list of booleans, access the boolean at index i, and use it to create a 1-bit bit vector. This re-implements HOL4's well-supported bit vector access operation. This small example exemplifies the increased complexity which pervades the ASL-derived specification.

4.4 Simulation between L3 and ASL-derived specifications

Definition 2 expresses our simulation relation, l3_models_asl, a predicate on the AST for instructions defined by the L3 specification. The instruction should successfully encode (using the L3-specified Encode) to produce a 32-bit *opcode*. Given L3 and ASL-derived machine states related by state_rel, if the L3 specification can decode and run (Run (Decode *opcode*)) the opcode without failure, the ASL-derived specification should also run (ExecA64) it successfully, both producing resultant states that remain related by state_rel. In addition, the predicate asl_sys_regs_ok should hold of the ASL-derived state throughout.

Note that we rely on L3 machinery: its AST for instructions and its encoder. These are orthogonal to the semantics of instructions, but use of an AST is well-suited to interactive proof and makes it simpler to carve out classes of opcodes. We have already noted that the ASL-derived specification does not provide such an AST or encoder.

The state equality relation state_rel is effectively a simple inclusion: the ASL-derived specification models strictly more registers and processor state than the L3 one, so state_rel asserts that the specifications agree on the parts modelled by both. We omit the full definition, which is verbose due to differences between the two specifications. In particular, L3 machine states (*l3*) are concise records, with clean access to registers, e.g. *l3*.PC is the program counter. Instead, ASL-derived states (*asl*) group registers together by their types, e.g. *asl*.regstate.bitvector_64_dec_reg of type string → word64 models all simple 64-bit system registers. The sheer number of registers forces this unusual approach: HOL4 struggles to cope with very large records in which each register is declared separately, so they must be grouped. Register references (*reg*) are used to index into these groups: these records contain

a canonical name (*reg*.name), and read-from/write-to functions (*reg*.read_from, *reg*.write_to). For example, PC_ref.read_from *asl*.regstate reads the program counter. This is equivalent to *asl*.regstate.bitvector_64_dec_reg "_PC".

We also explore a few subtleties that arise.

The L3 specification models Armv8.0, whereas the ASL-derived specification models Armv8.6. Though this is a "minor" version difference, there are observable effects, e.g. certain system control registers are 32-bit in L3 but 64-bit in ASL. In Armv8.0, these registers were 64-bit with their upper 32 bits reserved, so only their lower 32 bits required modelling. However, ASL faithfully models all 64 bits regardless.

The L3 specification represents memory cleanly as a total function from addresses to bytes (word64 $\rightarrow$ word8). However, the ASL-derived specification models memory as a finite mapping from natural numbers to trit-lists (num $\mapsto$ trit list), each intended to represent a byte. It also models per-address validity tags (num $\mapsto$ trit). We impose restrictions on ASL-derived memory to ensure we can equate it to L3-derived memory: its domain must be exactly the natural numbers representable by 64-bit words; its range must contain only *bit*-lists (i.e. no "unknown" trits) of length 8; all addresses must have a valid tag.

To model Arm's 31 general-purpose registers and zero register, the L3 specification uses a total mapping from 5-bit words (word5 $\rightarrow$ word64). The ASL specification instead uses a list of registers (word64 list). To access the register n, it takes the nth index of the list. We must require the list to have length of exactly 32.

The predicate asl_sys_regs_ok is necessary as L3 specifications model mostly user-level operation, omitting most system registers. We fix certain bits of these registers in the ASL-derived specification to ensure it models a processor in a similar mode of operation. We omit its definition here, but refer readers to our proofs and the Arm Architecture Reference Manual for a full account. Overall, we fix 11 bits of various system registers, and clear one memory-mapped register. This disables optional features not modelled in L3, such as secure modes for low exception levels and hypervisors.

Due to a versioning difference, four of these bits are in the TCR_EL{1,2,3} registers which are also modelled in L3. A feature implementing pointer authentication codes (PACs) was made compulsory in Armv8.3 onward, supporting authentication of addresses stored in registers before targeting them for a branch or load. Fixing these bits in the Armv8.6 modelled by ASL aligns behaviour more to the Armv8.0 modelled by L3.

Proving the simulation

Establishing simulation for a particular instruction effectively requires execution of the instruction on both specifications. We leverage pre-existing automation to execute the L3 specification effectively. However, the difficulties detailed in Section 4.3 complicate execution of the ASL-derived specification.

We adopt a partial, symbolic evaluation strategy to bypass decoding, which examines only known bits of opcodes (i.e. those that distinguish it from other classes of opcode). More precisely, we use HOL4's customisable call-by-value computation library to bypass the large, monolithic ExecA64 (a wrapper around the overall instruction decoding/execution function, DecodeA64). Once this has advanced the proof to the instruction-specific execution functions to which ExecA64 calls, we proceed by interactive proof.

We use the L3 specification heavily to provide abstract representations of auxiliary functions, avoiding unwrapping definitions and making case splits. For example, for the definitions in Example 1, we prove the following theorem for any 7-bit bit vector w:

$\vdash$ asl_HighestSetBit w = returnS (l3_HighestSetBit w).

A notable sub-proof relates the L3 and ASL-derived implementations of DecodeBitMasks, used in decoding to resolve immediate fields. Its 90 LoC ASL implementation is obfuscated by its optimised implementation, but a comment block asserts an equivalent 7 LoC definition. The L3 specification uses the shorter definition, so we must prove the asserted equivalence: we split up the large function into more manageable chunks, proving more presentable, manually-defined specifications for each by brute force enumeration of inputs.

In total, we prove l3_models_asl for the following instruction classes, requiring 7.5 kLoC of proof and 0.5 kLoC of simple automation (∼40 mins to build, limited by symbolic evaluation).

Instruction class description	Assembly shorthands
move wide operations	MOVK, MOVN, MOVZ
bit field moves	BFM, SBFM, UBFM
logical operations*†	AND[S], BIC[S], EON, EOR, ORN, ORR
addition/subtraction*†	ADD[S], SUB[S]
addition/subtraction with carry	ADC[S], SBC[S]
division	SDIV, UDIV
multiply with addition/subtraction	MADD, MSUB
multiply high	SMULH, UMULH
conditional compare*	CCMN, CCMP
conditional select	CSEL, CSINC, CSINV, CSNEG
branch immediate (call/jump)	B, BL
conditional branches	B.COND
branch register (jump)	BR
register extract	EXTR
address calculation	ADR, ADRP
byte/register loads/stores*‡	LD[U]R, LD[U]R[S]B, ST[U]R, ST[U]RB

Our reliance on the existing L3 specification keeps the proofs tractable, and no unexpected discrepancies were found in its semantics or encoder (we do encounter a known issue, see Section 8). All of our definitions and proofs thus far are self-contained, and not tied to any particular usage of our ASL-derived specification. To aid re-use in future work, they have been integrated into the HOL4 public repository.

5 Case study: compiler correctness in CakeML

Verification efforts such as CakeML and seL4 build correctness guarantees down to hardware by using unofficial L3 specifications of ISAs as a reference semantics. Equipped with our ASL-derived HOL4 specification (Section 3) and simulation proofs (Section 4.4), we can strengthen assurances in these results. We demonstrate our approach by proving a new compiler correctness result for CakeML (Section 5.2), targeting our ASL-derived specification.

What is CakeML? CakeML [27, 35] is a formally-specified language, end-to-end verified compiler, and proof ecosystem built using HOL4. The compiler is proved to preserve semantics: any valid input program is compiled to machine code with the same behaviour. The core of the project is a proof-producing translation from HOL4 to CakeML: input HOL4 functions are translated to output CakeML abstract syntax, and an accompanying proof that

* For immediate operands.
† For shifted register operands.
‡ Scaled 12-bit unsigned immediate offset and unscaled 9-bit signed immediate offset addressing modes.

the output semantics models the input HOL4 function [38]. The compiler is bootstrapped by first translating the HOL4 compiler function to CakeML, then evaluating the HOL4 compiler on the output CakeML within the HOL4 logic. This produces a binary which is verified to implement the original HOL4 compiler algorithm.

5.1 Target correctness proofs in CakeML

CakeML targets x86-64, Armv7, Armv8 (AArch64), RISC-V, MIPS, and Silver (a custom ISA for a verified processor [32]), proving compiler correctness with respect to specifications for each. These are all L3-derived, but we are concerned with Armv8 AArch64.

Both the compiler and its proofs are carefully structured to remain as target-agnostic as possible, reducing the implementation burden and proof obligation of supporting a new target. The following design decisions are made [18].

- The compiler produces generic assembly instructions known as Asm, which are embedded in its final intermediate language (LabLang). Defining an encoding to a new target requires a simple translation from this generic assembly.
- The compiler implementation is parametric over a target-specific configuration record, known as a "compiler configuration". For example, this record defines: an encoder from Asm, the registers useable in register allocation, classes of supported operations, and restrictions on valid immediate values. Compiling to a new target requires definition of an appropriate compiler configuration.
- Compiler correctness proofs use a generic form of target semantics (machine_semantics), parametrised by another kind of target-specific record known as a "machine configuration". All target-specific requirements of the proof are factored out into a precondition (target_configs_ok). Proving correctness for a new target requires definition of its machine configuration and discharging of the precondition.

We re-use the L3 encoder from the existing Armv8 backend. Therefore, to instantiate the generic compiler correctness theorem we must define a *compiler_config* and *machine_config*, and establish the precondition target_configs_ok *compiler_config machine_config*.

Machine configurations (*mc*) define exactly the features of a target necessary to define machine_semantics. This generic semantics models interference from the surrounding execution environment by allowing the environment to change a subset of target state arbitrarily between instructions and on FFI calls. Important fields include: a function to execute a single step (*mc*.target.next); accessors of registers, the program counter, and memory (*mc*.target.get_{reg,pc,byte}); information about the calling convention (*mc*.callee_saved_regs); a well-formedness predicate on target state (*mc*.target.state_ok); and a projection out of target state (*mc*.target.proj). This projection is the subset of target state which must *not* be modified by the surrounding execution environment, i.e. *mc*.target.proj is always the same both before and after external interference.

A proof of target_configs_ok establishes some simple initialisation conditions, and the key property encoder_correct *mc*.target. We omit a full definition, but intuitively: given an Asm state, an Asm instruction which will successfully retire, and an equivalent target state with the encoded Asm instruction in memory (using *mc*.config.encode), the target should be able to execute some number of steps successfully (according to *mc*.target.next) and end up in a state equivalent to the resulting Asm state. Note that a single Asm instruction may be encoded to a sequence of target opcodes, so multiple execution steps may be necessary. The property accounts for the surrounding environment by permitting interference by a function satisfying

▶ **Theorem 3.**

$\vdash$ mem *instr* (asm_to_arm8 *prog*) $\wedge$
($\forall$ *s*. Encode *instr* $\neq$ BadCode *s*) $\Rightarrow$
l3_models_asl_instr *instr*

▶ **Definition 4.**

```
NextASL ≝
 do
  write_regS BranchTaken_ref F;
  pc ← PC_read ();
  instr ← Mem_read0 pc 4 AccType_IFETCH;
  ExecA64 instr;
  branch_taken ← read_regS BranchTaken_ref;
  if branch_taken then returnS ()
  else
   do pc ← PC_read (); PC_set (pc + 4w) od
 od
```

interference_ok (i.e. the interference must preserve *mc*.target.proj). Target states must further satisfy *mc*.target.state_ok throughout. We omit details concerning program counters of the encoded instructions in memory, which must not be overwritten during execution.

Proving encoder_correct effectively requires running Asm and target state machines side-by-side. Therefore, proofs rely heavily on symbolic evaluation within HOL4, using L3-enabled automation to repeatedly apply *mc*.target.next and re-establish *mc*.target.state_ok after each step and its associated interference.

5.2 Lifting simulation to compiler correctness

We first prove Theorem 3: l3_models_asl (Definition 2) holds for any encodeable instruction produced by the CakeML compilation to Armv8 via Asm.

We then define a machine configuration (Section 5.1) for the ASL-derived specification, which largely mirrors the existing one for the L3 specification. It additionally enforces asl_sys_regs_ok (extending the state projection *mc*.target.proj to be sure that interference cannot break it), and ensures that memory and registers are well-formed (Section 4.4).

We must also define a next-step function (*mc*.target.next). Though the ASL specification provides a function TopLevel, this covers unwanted extraneous details, such as processor interrupts and memory-mapped devices. Instead we define NextASL (Definition 4), essentially TopLevel with the complexity stripped away. NextASL clears the branch-taken flag, reads the program counter, fetches the next opcode, and executes the opcode. The program counter is then updated only if no branch has been taken.

The bulk of the proof-effort is establishing encoder_correct. Interference from the surrounding execution environment prevents us from re-using the theorem proved for the L3 specification. We would require once and for all a transformation from interference with ASL-derived target states (satisfying interference_ok) to interference with L3 target states, such that the transformation preserves state_rel (to allow use of our simulation proofs). We cannot express this transformation: interference on ASL-derived specifications has a larger input space than that on L3 specifications, due to processor state not modelled in L3.

Instead, we run *three* state machines side-by-side: Asm, L3, and ASL-derived. We re-use pre-existing automation for the L3 specification to symbolically evaluate each opcode, giving a resultant L3 state, and then use Theorem 3 to produce a resultant ASL-derived state. This undergoes interference, breaking state_rel. We then interfere with the L3 state in a way which satisfies interference_ok and re-establishes state_rel between the resultant states. We repeat the process for as many opcodes as necessary for each Asm instruction encoding.

▶ **Theorem 5.**

⊢ Fail ∉ source_semantics *ffi program* ∧
compile arm8_compiler_config *program* = Some *compiled* ∧
asl_machine_config_ok *machine_config* ∧
program_in_memory arm8_compiler_config *machine_config compiled asl_state* ⇒
machine_semantics *machine_config ffi asl_state* ⊆
extend_with_resource_limit (source_semantics *ffi program*)

Overall, we produce Theorem 5, our top-level compiler correctness proof (mostly mechanical, 3.2 kLoC, ∼15 mins to build). A program with non-failing source semantics is compiled and installed in the memory of an appropriate *asl_state*. The resulting machine will have a machine_semantics which is a subset of the original source_semantics. We omit irrelevant details concerning CakeML's FFI model and extend_with_resource_limit. The latter extends source behaviours with out-of-memory errors: unlike source semantics, machine semantics is bounded by finite memory. This proof has been integrated into the CakeML public repository.

6 Discussion

We now consider our findings: what can we learn from our verification?

Trustworthiness of our specifications. The ASL specification cannot cover all aspects of its ISA, instead providing a working reference. When considering more complex features, such as concurrency and interrupts, it remains an abstraction of the authoritative detail of the Arm Reference Manual. However, even if a complete, machine-readable specification existed, proofs of semantics preservation with respect to it would be intractable without the abstractions. The ASL specifications and our work are a best-effort, modelling as much detail as currently feasible. Other proof goals (such as architecture security properties) may require different levels of detail.

Our root of trust is the extensive Arm-internal evaluation of the ASL specification, but extraction via Sail could introduce unintended semantic changes. Validating the generated HOL4 against Arm test suites or real hardware could improve trust (Sail's C backend has been tested in this way), but proving that the extraction preserves semantics is better still. This would be a significant undertaking, and require formal models of (at least) ASL and Sail. There has been some work into such models for both ASL (from personal communication) and Sail [4]. Note that both Sail and Lem have been validated through heavy usage. For example, Sail-extracted ASL models have successfully been used to simulate a Linux boot, and the Lem ecosystem is well-exercised. However, Sail's HOL4 backend has received limited prior usage, so our work better validates this pathway too.

Our simulation proofs (Section 4.4) allow each specification to improve trust in the other. The L3 and ASL-derived specifications differ in their derivations considerably, yet are formally connected by our work. Therefore, any bugs found in one must be found in the other, and the low likelihood of this strengthens assurances in both. Previously, the L3 Armv8 specification had not been rigorously validated due to the scarcity of Armv8 hardware when it was written. By contrast, the L3 Armv7 specification was tested extensively against real hardware.

Even so, we strengthen trust in a single L3 specification, and a single extraction pathway via Sail. There are many such specifications that can be generated, via different extraction options, versioning differences, choices in manual modification, and so on.

Absence of bugs. We discovered no new bugs in the L3 specification semantics or encoder (we encountered one known issue, see Section 8), despite the differing provenances of the L3 and ASL-derived specifications. This validates the approach of formalising a specification by using a DSL (L3) which can closely mirror it. However, the absence of bugs is surprising given that the ASL-derived specification covers implementation-defined behaviour and architecturally unknown values.

A partial explanation is our restricted domain of proofs of semantics preservation: we verify general-purpose instructions targeted by compilers, which avoid ambiguity to ensure portability. As a verified compiler, CakeML targets an even smaller subset of instructions.

Removal of address translation and interrupts (Sections 4.1 and 4.4 respectively) further reduces ambiguity significantly. We also do not tackle exception-handling, assuming that instructions retire without failure in the L3 specification (Definition 2, Section 4.4).

Furthermore, Arm intentionally reduced underspecification in Armv8 (compared to Armv7). For example: in Armv7 the program counter is a general-purpose register (R15) which can be modified unexpectedly by programmers; in Armv8 there is no direct access to the program counter. Architecturally unknown values are also mostly used as placeholders for variables which are declared and only later initialised.

The need for L3. We build our results on existing, unmodified tools: an ASL specification and the Sail ecosystem. Neither is designed for interactive theorem proving, so we use a purpose-built L3 specification as a stepping stone in proof. Instead, could we obviate the need for this indirection by adapting the tools to our domain?

One approach is to change the official ASL specification, though this would require support from Arm. Stylistic refactoring could reduce overly imperative code (e.g. Example 1). Logical refactoring has recently streamlined address translation, and could be applied to other parts of the specification. Forbidding early-return statements could reduce embedding of otherwise pure functions into the monad, but this is a significant language change. Alternatively ASL-to-ASL transformations before translation to Sail could achieve similar effects, without compromising our root of trust: the resultant ASL can be subjected to the Arm-internal test suite. Semantics-preserving transformations are already used in Arm to produce model-checking-friendly Verilog (from personal communication).

Another approach is to strengthen Sail's extraction to HOL4, taking inspiration from L3. Streamlining of Sail libraries for HOL4 is a first step. Extraction would also need to produce an AST for instructions. The challenge here is reproducing L3's ease of use: its AST is handcrafted for theorem proving (i.e. split into instruction classes for convenience and to avoid scaling issues with HOL4 types). However, Sail's extraction must be automatic and target-agnostic. Design choices made here will suit different domains, for example the AST could mirror assembly syntax or reference manual structure. A direct translation from ASL or Sail to HOL4 (cutting out Lem) could also streamline the extraction process and more closely target HOL4, but this is a significant undertaking.

Direct proof without L3 would also require new proof automation. This is because verification with respect to the L3 Armv8 specification relies heavily on its *step library*: concise Hoare triples which provide tractable, rule-based instruction semantics in interactive proof by explicitly stating conditions for well-definedness. No such library exists for the ASL-derived specification, but several have been painstakingly handcrafted for various other specifications. Automatic generation of certified step libraries using symbolic evaluation and symbolic execution [7] is a promising approach. A key challenge will be navigating the large, monolithic decoding/execution functions.

Though we have identified some promising avenues to verification without L3, none provides a magic bullet and taken together they represent a significant body of work. As industrial specifications of the scale of Armv8 become more prevalent, such engineering issues will be critical. We note that despite HOL4's veteran status amongst interactive theorem provers, its customisable simplifier, call-by-value evaluation, and highly accessible Standard ML metaprogramming make it an ideal candidate for work on such substantial specifications.

7 Future work

Removing the precondition to Theorem 3 is a clear next step, perhaps complicated by further incompatibilities between the L3 specification's Armv8.0 and the ASL-derived specification's Armv8.6. Upgrading the L3 to Armv8.6 and modelling a greater subset of the ISA could help. Notably, AArch32 and floating point instructions are not modelled and so remain unsupported by CakeML on Armv8. Connecting ASL floating point definitions with HOL4 libraries would be a significant challenge here.

The seL4 [25, 26] verified operating system microkernel uses a translation validation [57] phase to extend its guarantees to the binary on Armv7. An L3 specification provides the semantics for Armv7, but using an ASL-derived specification instead would strengthen the result. However, seL4 would need to be updated to target Armv8 first.

A more faithful account of address translation would improve trust in our work. We could take inspiration from the Sail developers, who prove in Isabelle/HOL that address translation adheres to a hand-written specification under certain conditions [2]. Recent simplifications to Arm's ASL specification of address translation may help here.

8 Related work

Applications of the Sail toolchain. Isla [3] is an SMT solver-based symbolic execution engine for Sail, producing simplified, per-opcode instruction execution traces derived from user-supplied constraints on machine state. Islaris [56] builds on this by formalising the traces in Coq and building a separation logic for reasoning about their semantics, using automation to simplify proofs considerably. Our clear common aim is to derive simplified machine code semantics from the complex ASL-derived specifications, using the desired domain to constrain semantics appropriately. Islaris' domain is machine code verification, so it relies on an external SMT solver on a per-opcode basis to provide sufficient automation. Our domain of semantics preservation instead allows us to abstract once and for all within the prover.

Sail is integral in key security proofs with respect to Morello, the Arm implementation of the Capability Hardware Enhanced RISC Instructions (CHERI) ISA [5, 40]. CHERI extends conventional ISAs with new features enabling memory protection, defending against many memory-based security exploits [58, 59]. The Isabelle/HOL proof (with some SMT solver oracles for bit vector operations) is complex, requiring hours to build with considerable computing power. For such security properties, monadic representation of specifications may be useful. In our case it obfuscates the high-level semantics of the instruction.

Validation of ISA specifications. The Provably Secure Execution Platforms for Embedded Systems (PROSPER) project has created Scam-V [39], an automatic validator for ISA specifications. It searches for pairs of executions which behave identically according to the specification, but are distinguishable on hardware via some observation. If such a pair exists, the specification has failed to model an observable side-channel correctly.

Scam-V has uncovered a bug in the L3 specification of Armv8: the compare and branch on non-zero instruction (CBNZ) incorrectly behaves like the compare and branch on zero instruction (CBZ). We have replicated this finding in a failed attempt to prove l3_models_asl for CBNZ. Furthermore, we have validated a corrected specification of CBNZ by successfully establishing l3_models_asl for it. Note that the CakeML compiler does not produce CBNZ instructions, so our CakeML proofs are unaffected.

Verified compilation to hardware. Erbsen et al. [11] have verified an application on a realistic embedded stack which includes a C-based language, verified compiler, and verified processor (the Kami [9] verified implementation of RISC-V). Their goals contrast with ours: they focus on a complete result down to a specific processor, prioritising clean verification across interfaces and realistic I/O. However, we must remain processor-agnostic and so are forced to verify against an ISA specification only.

Verification with respect to the Arm architecture. Official Arm ISA specifications have been used within Arm to establish architectural properties below the ISA abstraction boundary. These efforts use automated techniques (SMT solving and bounded model-checking) to suit their problem domain and their setting in industry.

ISA-Formal [49,55] uses bounded model-checking to verify that Arm ISA implementations adhere to their intended specification. The project translates ASL specifications to reference Verilog implementations [51], comparing these to the actual implementation using an off-the-shelf bounded model-checker. Failures are output as counterexamples.

Secure-M [54] produces key security properties for the Arm M-class specification, using an automated SMT solver to verify they hold. It provides a more rigorous alternative to code review and testing for architecture modifications, focussing on whole-specification properties for continuous integration testing. There is also some support for developer-stated assertion-checking. A future goal is to prove that successive versions of specifications preserve necessary backwards-compatibility.

9 Conclusion

To the best of our knowledge, we have produced the first compiler correctness proof which is backed by an official specification of an Arm ISA. Our proof is made tractable by the use of an existing L3 specification to abstract away the complexity of the official ASL. This approach validates the L3 specification more rigorously, and we report no new bugs uncovered. Our work remains decoupled from CakeML for re-use in future verification efforts, and has been incorporated in the HOL4 and CakeML public repositories.

Throughout we have kept in mind the issue of trustworthiness. Our work strengthens assurances not only in CakeML, but also in other projects relying on the L3 specification of Armv8. We have reproduced results from the Sail ecosystem and further validated its extraction process. Furthermore, we have demonstrated a novel application of the Sail ecosystem to proofs of semantic preservation.

References

1 Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Shaked Flur, Kathryn E. Gray, Prashanth Mundkur, Robert M. Norton, Christopher Pulte, Alastair Reid, Peter Sewell, Ian Stark, and Mark Wassell. Detailed models of instruction set architectures: From pseudocode to formal semantics. In *Proceedings of the Automated Reasoning Workshop*, 2018. Two-page abstract. URL: `http://www.cl.cam.ac.uk/~pes20/sail/2018-04-12-arw-paper.pdf`.

2 Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. *Proceedings of the ACM on Programming Languages (PACMPL)*, 3(POPL), 2019. doi:10.1145/3290384.

3 Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. Isla: Integrating full-scale ISA semantics and axiomatic concurrency models. In *Computer Aided Verification (CAV)*, volume 12759 of *Lecture Notes in Computer Science*. Springer, 2021. doi:10.1007/978-3-030-81685-8_14.

4 Alasdair Armstrong, Neel Krishnaswami, Peter Sewell, and Mark Wassell. Formalisation of MiniSail in the Isabelle theorem prover. In *Proceedings of the Automated Reasoning Workshop*, 2018. Two-page abstract. URL: https://www.cl.cam.ac.uk/~pes20/sail/arw18_mpew2.pdf.

5 Thomas Bauereiss, Brian Campbell, Thomas Sewell, Alasdair Armstrong, Lawrence Esswood, Ian Stark, Graeme Barnes, Robert N. M. Watson, and Peter Sewell. Verified security for the Morello capability-enhanced prototype Arm architecture. In *European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science. Springer, 2022. To appear. URL: http://www.cl.cam.ac.uk/~pes20/morello-proofs-esop2022.pdf.

6 Christoph Baumann, Mats Näslund, Christian Gehrmann, Oliver Schwarz, and Hans Thorsen. A high assurance virtualization platform for ARMv8. In *European Conference on Networks and Communications (EuCNC)*. IEEE, 2016. doi:10.1109/EuCNC.2016.7561034.

7 Brian Campbell and Ian Stark. Extracting behaviour from an executable instruction set model. In *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2016. doi:10.1109/FMCAD.2016.7886658.

8 Adam Chlipala. Algorithmic checking of security arguments for microprocessors. In *GOMACTech Conference*, 2019. URL: https://apps.dtic.mil/sti/citations/AD1075652.

9 Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: a platform for high-level parametric hardware specification and its modular verification. *Proceedings of the ACM on Programming Languages*, 1(ICFP), 2017. doi:10.1145/3110268.

10 Alessandro Coglio and Shilpi Goel. Adding 32-bit mode to the ACL2 model of the x86 ISA. In *Workshop on the ACL2 Theorem Prover and Its Applications*, volume 280 of *EPTCS*, 2018. doi:10.4204/EPTCS.280.6.

11 Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. Integration verification across software and hardware for a simple embedded system. In *Programming Language Design and Implementation (PLDI)*. ACM, 2021. doi:10.1145/3453483.3454065.

12 Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *Principles of Programming Languages (POPL)*. ACM, 2016. doi:10.1145/2837614.2837615.

13 Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. Mixed-size concurrency: ARM, POWER, C/C++11, and SC. In *Principles of Programming Languages (POPL)*. ACM, 2017. doi:10.1145/3009837.3009839.

14 Anthony C. J. Fox. Formal specification and verification of ARM6. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 2758 of *Lecture Notes in Computer Science*. Springer, 2003. doi:10.1007/10930755_2.

15 Anthony C. J. Fox. Directions in ISA specification. In *Interactive Theorem Proving (ITP)*, volume 7406 of *Lecture Notes in Computer Science*. Springer, 2012. doi:10.1007/978-3-642-32347-8_23.

16 Anthony C. J. Fox. Improved tool support for machine-code decompilation in HOL4. In *Interactive Theorem Proving (ITP)*, volume 9236 of *Lecture Notes in Computer Science*. Springer, 2015. doi:10.1007/978-3-319-22102-1_12.

17 Anthony C. J. Fox and Magnus O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Interactive Theorem Proving (ITP)*, volume 6172 of *Lecture Notes in Computer Science*. Springer, 2010. doi:10.1007/978-3-642-14052-5_18.

18 Anthony C. J. Fox, Magnus O. Myreen, Yong Kiam Tan, and Ramana Kumar. Verified compilation of CakeML to multiple machine-code targets. In *Certified Programs and Proofs (CPP)*. ACM, 2017. doi:10.1145/3018610.3018621.

19 Shilpi Goel and Warren A. Hunt Jr. Automated code proofs on a formal model of the X86. In *Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 8164 of *Lecture Notes in Computer Science*. Springer, 2013. doi:10.1007/978-3-642-54108-7_12.

20 Shilpi Goel, Warren A. Hunt Jr., and Matt Kaufmann. Engineering a formal, executable x86 ISA simulator for software verification. In *Provably Correct Systems*, NASA Monographs in Systems and Software Engineering. Springer, 2017. doi:10.1007/978-3-319-48628-4_8.

21 Shilpi Goel, Anna Slobodová, Rob Sumners, and Sol Swords. Verifying x86 instruction implementations. In *Certified Programs and Proofs (CPP)*. ACM, 2020. doi:10.1145/3372885.3373811.

22 Kathryn E. Gray, Gabriel Kerneis, Dominic P. Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Microarchitecture (MICRO)*. ACM, 2015. doi:10.1145/2830772.2830775.

23 Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2016. doi:10.5555/3026877.3026928.

24 Roberto Guanciale, Hamed Nemati, Mads Dam, and Christoph Baumann. Provably secure memory isolation for linux on ARM. *Journal of Computer Security*, 24(6), 2016. doi:10.3233/JCS-160558.

25 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 OS kernel. In *Operating Systems Principles (SOSP)*. ACM, 2009. doi:10.1145/1629575.1629596.

26 Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1), 2014. doi:10.1145/2560537.

27 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In *Principles of Programming Languages (POPL)*. ACM, 2014. doi:10.1145/2535838.2535841.

28 Dirk Leinenbach and Thomas Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *Formal Methods (FM)*, volume 5850 of *Lecture Notes in Computer Science*. Springer, 2009. doi:10.1007/978-3-642-05089-3_51.

29 Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7), 2009. doi:10.1145/1538788.1538814.

30 Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. A secure and formally verified Linux KVM hypervisor. In *Security and Privacy (SP)*. IEEE, 2021. doi:10.1109/SP40001.2021.00049.

31 Andreas Lindner, Roberto Guanciale, and Roberto Metere. TrABin: Trustworthy analyses of binaries. *Science of Computer Programming*, 174, 2019. doi:10.1016/j.scico.2019.01.001.

32 Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony C. J. Fox. Verified compilation on a verified processor. In *Programming Language Design and Implementation (PLDI)*. ACM, 2019. doi:10.1145/3314221.3314622.

33 Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering of real-world semantics. In *International Conference on Functional Programming (ICFP)*. ACM, 2014. doi:10.1145/2628136.2628143.

34 Magnus O. Myreen. Verified just-in-time compiler on x86. In *Principles of Programming Languages (POPL)*. ACM, 2010. doi:10.1145/1706299.1706313.

35 Magnus O. Myreen. The CakeML project's quest for ever stronger correctness theorems (invited paper). In *Interactive Theorem Proving (ITP)*, volume 193 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ITP.2021.1.

36 Magnus O. Myreen and Jared Davis. A verified runtime for a verified theorem prover. In *Interactive Theorem Proving (ITP)*, volume 6898 of *Lecture Notes in Computer Science*. Springer, 2011. doi:10.1007/978-3-642-22863-6_20.

37 Magnus O. Myreen and Michael J. C. Gordon. Verified LISP implementations on ARM, x86 and PowerPC. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *Lecture Notes in Computer Science*. Springer, 2009. doi:10.1007/978-3-642-03359-9_25.

38 Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *Journal of Functional Programming (JFP)*, 24(2-3), 2014. doi:10.1017/S0956796813000282.

39 Hamed Nemati, Pablo Buiras, Andreas Lindner, Roberto Guanciale, and Swen Jacobs. Validation of abstract side-channel models for computer architectures. In *Computer Aided Verification (CAV)*, volume 12224 of *Lecture Notes in Computer Science*. Springer, 2020. doi:10.1007/978-3-030-53288-8_12.

40 Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony C. J. Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *Security and Privacy (SP)*. IEEE, 2020. doi:10.1109/SP40000.2020.00055.

41 Scott Owens, Peter Böhm, Francesco Zappa Nardelli, and Peter Sewell. Lem: A lightweight tool for heavyweight semantics. In *Interactive Theorem Proving (ITP)*, volume 6898 of *Lecture Notes in Computer Science*. Springer, 2011. doi:10.1007/978-3-642-22863-6_27.

42 Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proceedings of the ACM on Programming Languages*, 2(POPL), 2018. doi:10.1145/3158107.

43 Alastair Reid. Arm releases machine readable architecture specification. URL: https://alastairreid.github.io/ARM-v8a-xml-release/.

44 Alastair Reid. Arm v8.3 machine readable specification released. URL: https://alastairreid.github.io/arm-v8_3/.

45 Alastair Reid. Arm's architecture specification language. URL: https://alastairreid.github.io/specification_languages/.

46 Alastair Reid. ASL lexical syntax. URL: https://alastairreid.github.io/using-asli/.

47 Alastair Reid. Dissecting Arm's machine readable specification. URL: https://alastairreid.github.io/dissecting-Arm-MRA/.

48 Alastair Reid. Formal validation of the Arm v8-M specification. URL: https://alastairreid.github.io/validating-specs/.

49 Alastair Reid. Limitations of ISA-Formal. URL: https://alastairreid.github.io/isa-formal-limitations/.

50 Alastair Reid. Using ASLi with Arm's v8.6-A ISA specification. URL: https://alastairreid.github.io/asl-lexical-syntax/.

51 Alastair Reid. Verifying against the official Arm specification. URL: https://alastairreid.github.io/using-armarm/.

52 Alastair Reid. What can you do with an ISA specification? URL: https://alastairreid.github.io/uses-for-isa-specs/.

53 Alastair Reid. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2016. doi:10.1109/FMCAD.2016.7886675.

54 Alastair Reid. Who guards the guards? Formal validation of the Arm v8-M architecture specification. *Proceedings of the ACM on Programming Languages (PACMPL)*, 1(OOPSLA), 2017. doi:10.1145/3133912.

55 Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end verification of processors with ISA-Formal. In *Computer Aided Verification (CAV)*, volume 9780 of *Lecture Notes in Computer Science*. Springer, 2016. doi:10.1007/978-3-319-41540-6_3.

56 Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. Islaris: verification of machine code against authoritative ISA semantics. In *Programming Language Design and Implementation (PLDI)*. ACM, 2022. doi:10.1145/3519939.3523434.

57 Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *Programming Language Design and Implementation (PLDI)*. ACM, 2013. doi:10.1145/2491956.2462183.

58 Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Peter Neumann. An introduction to CHERI. Technical Report UCAM-CL-TR-941, University of Cambridge, Computer Laboratory, September 2019. doi:10.48456/tr-941.

59 Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Technical Report UCAM-CL-TR-951, University of Cambridge, Computer Laboratory, October 2020. doi:10.48456/tr-951.

Formalization of Randomized Approximation Algorithms for Frequency Moments

Emin Karayel
Department of Computer Science, Technische Universität München, Germany

Abstract

In 1999 Alon et al. introduced the still active research topic of approximating the frequency moments of a data stream using randomized algorithms with minimal space usage. This includes the problem of estimating the cardinality of the stream elements – the zeroth frequency moment. Higher-order frequency moments provide information about the skew of the data stream which is, for example, critical information for parallel processing. (The k-th frequency moment of a data stream is the sum of the k-th powers of the occurrence counts of each element in the stream.) They introduce both lower bounds and upper bounds on the space complexity of the problems, which were later improved by newer publications. The algorithms have guaranteed success probabilities and accuracies without making any assumptions on the input distribution. They are an interesting use case for formal verification because their correctness proofs require a large body of deep results from algebra, analysis and probability theory. This work reports on the formal verification of three algorithms for the approximation of F_0, F_2 and F_k for $k \geq 3$. The results include the identification of significantly simpler algorithms with the same runtime and space complexities as the previously known ones as well as the development of several reusable components, such as a formalization of universal hash families, amplification methods for randomized algorithms, a model for one-pass data stream algorithms or a generic flexible encoding library for the verification of space complexities.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Higher order logic; Mathematics of computing → Probabilistic algorithms; Theory of computation → Pseudorandomness and derandomization

Keywords and phrases Formal Verification, Isabelle/HOL, Randomized Algorithms, Frequency Moments

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.21

Supplementary Material *Software (Isabelle/HOL Formalization)*: `https://isa-afp.org/entries/Frequency_Moments.html` [33]

Acknowledgements Special thanks to Tobias Nipkow for all his support, guidance and feedback on this work, to Manuel Eberl for advice and simplifications on the Isabelle/HOL formalization and to the anonymous reviewers for their careful feedback and many helpful comments and suggestions.

1 Introduction

Flajolet and Martin [20] introduced one of the first modern big data algorithms to approximate the number of distinct elements in a stream using a randomized algorithm with logarithmic space usage. In 1999 Alon et al. [3] realize that the estimation of the number of distinct elements is a special case of a more generic problem. They define frequency moments of an input stream $a_1, a_2, \ldots, a_m \in U$ with length m by:

$$F_k := \sum_{u \in U} C(u, a)^k \tag{1}$$

where $C(u, a)$ is the count of occurrences of u in the stream a, i.e., $C(u, a) := |\{i \mid a_i = u\}|$. Then they provide randomized space-efficient algorithms for the estimation of all frequency moments. They also overcome the need for idealized model assumptions about hash functions,

which were used in previous work. Instead they provide concrete solutions using universal hash families. Moreover, they establish lower bounds for the problems, which were later improved and matched [7, 9, 12, 28, 30].

Note that F_0 coincides with the number of distinct elements in the stream. Higher frequency moments are useful to derive information about the *skewness* of the rank-size distribution of the data stream. This is, for example, useful during query planning in database applications [14] and predicting the speed-up factor in parallel data processing [25]. Using estimates of both F_2 and F_0 it is possible to compute several statistical dispersion measures, such as variance, standard deviation or Gini's index of homogeneity [3].

The verification of these algorithms is a distinct challenge that requires a large body of mathematical results from probability theory, such as the Hoeffding, Chebyshev and Hölder[1] inequalities as well as algebraic results about finite fields and polynomials. However, the body of existing strong theoretical results in Isabelle/HOL [40] is growing, both in the Archive of Formal Proofs (AFP) [2] as well as Isabelle's own libraries, such that showing the correctness of such theory-heavy algorithms has become feasible.

On the other hand, it is very hard to gain confidence on the correctness of these algorithms empirically and/or using traditional unit tests, because:

- the correctness properties are probabilistic, and
- the correctness properties are independent of the statistical properties of the input data – however properties established using statistical test can only provide confidence for a specified input distribution.

In the accompanying formalization [33], I verify the correctness of three distinct algorithms for the estimation of frequency moments using the Isabelle/HOL theorem prover. Table 1 summarizes them together with their asymptotic space complexity in bits and the source material they were based on. I have not been able to find any previous publications on the formalization of these algorithms. They all return an approximation F_k^* of F_k with relative error $\delta > 0$ with a probability of at least $1 - \varepsilon$, i.e.,

$$P(|F_k^* - F_k| \leq \delta F_k) \geq 1 - \varepsilon, \tag{2}$$

and require only one-pass over the stream of elements. They are all Monte-Carlo algorithms, i.e., the established probability bounds hold for every input. The fact that the result has a probability distribution stems only from the internal random choices of the algorithms.

Table 1 Formally verified algorithms: n denotes the size of the universe of the elements of the stream, m is the length of the stream, $\varepsilon \in (0, 1)$ is the maximum failure probability, $\delta \in (0, 1)$ is the required relative accuracy. See also Equation 2.

Approximation of	Asymptotic Space Complexity for $n, k, m \to \infty$ and $\varepsilon, \delta \to 0^+$	Based on
F_0	$O\left(\ln(\varepsilon^{-1})\left(\ln n + \delta^{-2}(\ln(\delta^{-1}) + \ln\ln n)\right)\right)$	[5]
F_2	$O\left(\ln(\varepsilon^{-1})\delta^{-2}(\ln n + \ln m)\right)$	[3, §2.2]
F_k for[2] $k \geq 3$	$O\left(\ln(\varepsilon^{-1})\delta^{-2}(\ln n + \ln m)kn^{1-\frac{1}{k}}\right)$	[3, §2.1]

[1] The Hölder inequality is part of the formalization of L_p spaces by Gouezel [24].

[2] The algorithm is actually correct even for $k \geq 1$ but the specialized algorithms for $k \leq 2$ are better in terms of space complexity.

A note about the case F_1: Since $F_1 = m$ an exact solution for the problem only requires $O(\ln m)$ bits of memory (which is just a counter). Alon et al. also discuss a randomized algorithm requiring $O(\ln(\ln m))$ bits using approximate counting [37]. Because it deviates a lot from the more interesting cases $k \neq 1$, I did not formalize that case.

While the algorithm for F_k identically matches the source material, I made some improvements to the F_0, F_2 algorithms, but match the space and runtime complexity bounds of the source material.

In particular, contrary to previous work in this field, the results are established using simple prime fields $GF(p)$ for p prime, instead of fields with a two power order: $GF(2^n)$. This is significant because on common machines computations in simple prime fields can be implemented easily. On the other hand, especially fast multiplications in prime power fields require advanced algorithms [38, 43]. As far as I can tell, the fact that simple prime fields are sufficient to achieve the space complexity bounds in Table 1 has not been observed by previous publications.

In the case of the approximation algorithm for F_0, I could derive a new KMV-type (k-minimum value) algorithm with a rounding component that matches the complexities of Algorithm 3 by Bar-Yossef et al. [5], but its implementation (and hence verification) is simpler than their solution. While there is considerable research on KMV-type algorithms [6, 22, 50], I could not find any previous publications that verify the correctness of this variant.

Some of the results required for the formal verification are reusable general results, which I have contributed as separate entries into the AFP [31, 32, 34, 35, 36]. The formalization of the algorithms for the frequency moments are in the AFP entry: Formalization of Randomized Approximation Algorithms for Frequency Moments [33]. I needed 7349 lines (not including text or empty lines) overall, of which 4779 lines are reusable more general results.

The next section contains theoretical background used in these algorithms, a description of how the randomized algorithms are modeled using the Giry-monad as a shallow-embedding in Isabelle/HOL and how the space complexity is verified. The details about the algorithms follow in Section 3. Related work is presented in Section 4 and Section 5 concludes with future research opportunities.

2 Background

2.1 Universal Hash Families

Universal hash families are a critical component in the algorithms for F_0 and F_2. They are used to randomize the input data such that statistical methods can be employed even without any assumptions about the input distribution. To give a rough idea why randomization is useful, Figure 1 depicts the probability density function of the third smallest element of independent uniformly distributed random variables. It is visible that information about the count of random variables can be deduced by the value of order statistics of the random variables. A first idea would be applying a fully random function $h : U \to \{0, \ldots, n-1\}$ on the values of the input stream before processing it.[3] However, choosing and storing such a function requires $O(n \ln n)$ bits of memory, which is far above the space complexity of the algorithms.

[3] The application of such a function to the input stream may obviously with some extent and probability affect the frequency moment itself, which needs to be taken into account in the design of the algorithms.

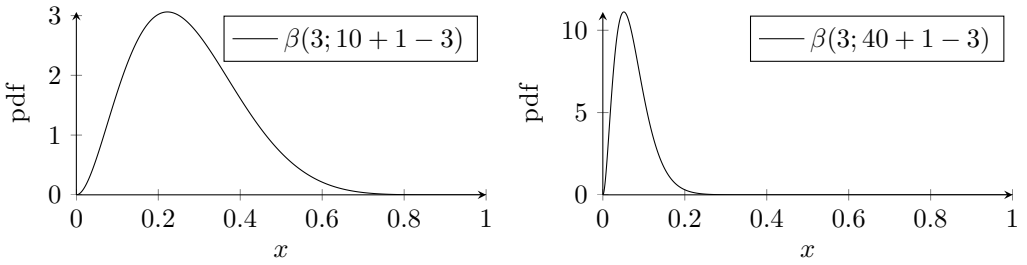


Figure 1 The probability density function for the distribution of the third-smallest element of 10 [left] (resp. 40 [right]) independent uniformly distributed random variables with range $[0, 1]$. The distributions are instances of the β-distribution.

The key insight from Alon et al. [3] was that it is possible to make headway even if the family of functions h is chosen from is only k-*universal* (where $k = 2$ in their F_0 approximation algorithm and $k = 4$ in their algorithm for the approximation of F_2). If a function h is chosen from a k-universal hash family than the restriction of h to any k elements of its domain is like a random function. More precisely:

▶ **Definition 1.** *If we regard a finite family $\mathcal{H}$ of hash functions $U \to V$ as a uniform discrete probability space, then $\mathcal{H}$ is k-universal, if*

- *for each $u \in U$ the evaluation function $h \mapsto h(u)$ is a random variable with uniform distribution on V, i.e., $P(\{h \in \mathcal{H} | h(u) = v\}) = |V|^{-1}$ and*
- *for any k or fewer distinct domain elements $u_1, \ldots, u_l$ the evaluation functions $h \mapsto h(u_1), \ldots, h \mapsto h(u_l)$ are independent random variables.*[4]

A different way of stating the second requirement in the definition is that the evaluation functions must be k-wise independent random variables. Bienaymé's identity is an example where pairwise independence is useful. With it the variance of a sum of pairwise independent variables $X_1, X_2, \ldots, X_n$ can be computed as the sum of the variances of the summands, i.e.

$$\operatorname{Var}\left(\sum_{k=1}^{n} X_i\right) = \sum_{k=1}^{n} \operatorname{Var}(X_i).$$

A generic construction for k-universal hash families was described by Wegman and Carter [49, §1] for the case where the domain and range of the hash family is a finite field $GF(q)$. (In the following I refer to this hash family as the *Carter–Wegman hash-family.*) The result essentially follows from the fact that given k key-value pairs there exists exactly one polynomial with degree strictly less than k interpolating those points. The k-universal hash family consists of the polynomials with coefficients in $GF(q)$ with degree less than k, where the evaluation of the polynomial is the hash function. Let us see how the results are stated in the formalization [36]:

```
definition (in prob-space) k-wise-indep-vars where
  k-wise-indep-vars k M' X' I = (∀ J ⊆ I. card J ≤ k ⟶ finite J ⟶ indep-vars M' X' J)
```

[4] This definition closely follows the definition from [48, §3.5.5], with the minor modification that independence is required not only for exactly k, but also for *fewer* than k distinct domain elements. The modification only has an effect in the corner case, where $|U| < k$ and helps avoid unnecessary special cases in the formalization.

This statement introduces a new definition for indexed sets of random variables, where any subset with no more than k elements is independent. The notation (**in** *prob-space*) means that the definition is in the context of all probability spaces. The function *card* denotes the number elements of a finite set. The predicate *indep-vars* M' X' J is true if X' is a J-indexed set of independent random variables from the probability space to the J-indexed set of measurable spaces M'.

definition *hash* **where** *hash* F x ω = *ring.eval* F ω x

This definition introduces the abbreviation *hash* for the evaluation of a polynomial over a ring F.[5]

lemma *hash-prob-single*:[6]
assumes *field* $F \wedge$ *finite* (*carrier* F)
assumes $\{x, y\} \subseteq$ *carrier* F
assumes $1 \leq n$
shows $\mathcal{P}(\omega$ *in pmf-of-set* (*bounded-degree-polynomials* F n)*. hash* F x $\omega = y)$
$= 1/($*card* (*carrier* F))

This lemma implies that the Carter–Wegman hash-family fulfills the first condition of Definition 1: Assuming F is a finite field then the *hash* function is a random variable with uniform distribution on the probability space *pmf-of-set* (*bounded-degree-polynomials* F n), i.e., the set of polynomials with coefficients in F and degree less than n. The expression *carrier* F denotes the underlying set of the field and the notation in the last line, $\mathcal{P}(\omega$ *in* M*.* P $\omega)$, denotes the probability of the event $\{\omega \mid P\,\omega\}$ in the probability space M.

lemma *hash-k-wise-indep*:
assumes *field* $F \wedge$ *finite* (*carrier* F)
assumes $1 \leq n$
shows *prob-space.k-wise-indep-vars* (*pmf-of-set* (*bounded-degree-polynomials* F n)) n
(λ-. *pmf-of-set* (*carrier* F)) (*hash* F) (*carrier* F)

This lemma implies that the Carter–Wegman hash-family fulfills the second condition of Definition 1, i.e., if F is a finite field, then the *hash* functions are k-wise independent random variables.

Besides polynomials there is also a method called tabulation hashing to construct k-universal hash families for $k \leq 5$ [47]. In the case where V has only two values, orthogonal arrays of strength k [11] can also be interpreted as k-universal hash families. However, because of the generic nature of Carter and Wegman's solution, I formalized their construction.

As noted in the introduction, I use the simple prime fields $GF(p)$ where p is prime. In particular, if the universe size is n, then the smallest prime $p \geq n$ is chosen to construct a k-universal hash-family for the stream elements. Because of Bertrand's postulate[7] $p \leq 2n+2$, which is used to bound the space used for the coefficients of the polynomial as well as the hashed values.

[5] Some of the results in the theory file require only that F is a ring. Currently the lemmas are only applied in the context of finite fields, but there is a good reason to avoid the field assumption when it is not necessary. An example is the use of approximate primality tests where F would be a field only with high probability but it would be unconditionally a ring.

[6] To improve readability embeddings between natural numbers, integers, rational, floating point and (extended) real numbers are omitted.

[7] Bertrand's postulate was formalized in Isabelle/HOL by Biendarra and Eberl [8].

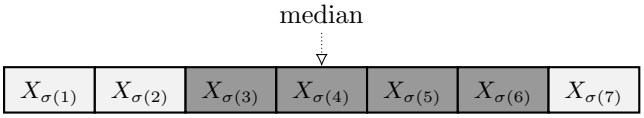


Figure 2 An example for 7 random variables (sorted via the permutation σ). The variables that are inside the desired interval $[a, b]$ are shaded gray.

In some publications in this field standard (or cryptographic) hash functions are assumed to be independent random variables. See for example in the context of F_0-estimation: [19, 20, 22, 26]. While there is also empirical evidence, that practically useful conclusions can be drawn under such model assumptions, in this work I have followed the approach by Alon et al. [3] and use universal hash families to avoid unjustified statistical model assumptions.

2.2 The Median Method

The reader may have observed that a common factor of the space complexities of the algorithms is $\ln(\varepsilon^{-1})$. The factor stems from the application of the median method to amplify the success probability [3]. To understand the method let us consider $2n + 1$ independent random variables $X_1, X_2, \ldots, X_{2n+1}$, that are in an interval $[a, b]$ with a probability of $2/3$. In the case of the algorithms a (resp. b) denote the minimal (respectively maximal) value the algorithm may return given the desired accuracy parameter. The interesting result is that the median of the random variables will be in the interval $[a, b]$ with a considerably higher probability of $1 - \exp\left(\frac{-(n+2)}{9}\right)$.

To see why this works, let us make a preliminary observation:

▶ **Observation 2.** *If at least $n + 1$ of the random variables are in the desired interval, then the median will be as well. This is because if we sort the random variables X_i the random variables that are in the interval form a consecutive subsequence in the sorted sequence. If its length is at least $n + 1$, it will necessarily contain the median. See also Figure 2 for an example.*

Hence we are left with estimating the probability that at least $n + 1$ of the X_i are in the range $[a, b]$. Let us introduce a second set of random variables, indicating whether each of the above random variables X_i are within the desired interval:

$$Y_i := \begin{cases} 1 & \text{if } X_i \in [a, b] \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

Note that the values of random variables Y_i are either 0 or 1 and the expectation of each is bounded from below by $2/3$ because of our assumption about the probability that each X_i is in $[a, b]$ is $2/3$, i.e., $\mathbb{E}(Y_i) \geq 2/3$. Now, Hoeffding's inequality [27] implies for the sum $S = Y_1 + \ldots + Y_{2n+1}$ – the number of random variables X_i whose values lie within the interval $[a, b]$:

$$P(S \leq \mathbb{E}(S) - t) \leq \exp\left(\frac{-2t^2}{2n+1}\right).$$

The case we are interested in is $t = \frac{n}{3} + \frac{2}{3}$. Thus

$$P(S \leq n) \leq \exp\left(\frac{-2(n+2)^2}{9(2n+1)}\right) \leq \exp\left(\frac{-(n+2)}{9}\right).$$

I.e. the probability that n or fewer of the X_i are inside of $[a, b]$ decreases exponentially with n. With Observation 2 this implies, that the probability that the median of the random variables X_i is outside the interval $[a, b]$ decreases exponentially with n.

In the design of the algorithms for F_0, F_2 and F_k, this result is used to amplify the probabilities of the algorithms. To achieve this, the algorithms run $18 \ln(\varepsilon^{-1})$ copies of the same estimation algorithm in parallel (with fully independent random choices). Each copy obtains a result which approximates F_k with a relative error of δ with a probability of $2/3$. In the final step the algorithm returns the median of the results of the copies, which then is in the desired interval with probability $1 - \varepsilon$. The formalization [35] includes a generalized version of this result:

```
lemma (in prob-space) median-bound-1:
  assumes α > 0
  assumes ε ∈ {0<..<1}
  assumes indep-vars (λ-. borel) X {0..<n}
  assumes n ≥ - ln ε / (2 * α²)
  assumes ∀ i ∈ {0..<n}. P(ω in M. X i ω ∈ ({a..b} :: real set)) ≥ 1/2+α
  shows P(ω in M. median n (λi. X i ω) ∈ {a..b}) ≥ 1−ε
```

Assuming M is a probability space, $\alpha > 0$, $0 < \varepsilon < 1$, $n \geq \frac{1}{2} \ln(\varepsilon^{-1})\alpha^{-2}$ and $X_0, \ldots, X_{n-1}$ are independent Borel-measurable random variables: If the probability for each X_i to be in the interval $[a, b] \subset \mathbb{R}$ is $\frac{1}{2} + \alpha$ then the probability that the median is in the same interval is at least $1 - \varepsilon$.

▶ Note. The formalization [35] contains an even more general lemma *median-bound* where the above result is shown for all convex sets in second-countable linearly ordered Borel spaces instead of just finite closed intervals of the form $[a, b] \subset \mathbb{R}$. The generalized version might be useful, if the codomain of the random variables is of another type than the real numbers, such as the rational numbers, or if the interval is not finite and closed. The lemma *median-bound-1* above is a specialization of *median-bound*.

▶ Note. Observation 2 is missing in previous publications. That it is necessary for a rigorous proof became apparent during the formalization. The argument starting from Eq. 3 can be found in [3, §2.1]; it is included here for completeness.

▶ Note. The requirement that the variables are *Borel* measurable in the lemma is essential. Without the assumption, the median of the random variables would not necessarily be measurable. For interested readers: The measurability proof in the formalization for the median relies on the existence of branch-free comparison-sort algorithms.[8] Given the number of elements in the sequence, such an algorithm performs compare-swap operations in a pre-defined order on pre-defined indices. The results of each compare swap operation can be represented as the pair of functions: $\min(X_i, X_j)$, $\max(X_i, X_j)$, which are Borel-measurable if X_i, X_j are. And the entire sorting operation can, using a branch-free comparison sort algorithm, be represented as a repeated application of such compare-swap operations. Thus the median – and any other order statistic – of the random variables is still measurable. [9]

[8] Sometimes these algorithms are also called sorting networks or oblivious comparison sort algorithms [41].

[9] It may be possible to proof the measurability of order statistics directly by verifying that the existing *sort* operation in Isabelle is measurable. However the approach using a branch-free sorting algorithm is more concise, in particular, it circumvents the need for introducing a σ-algebra on lists.

2.3 Formalization of Randomized Algorithms

Eberl et al. [16] built a library in Isabelle/HOL for the formalization of randomized algorithms, in particular a formalization of the Giry monad. To introduce the notation, let us first consider a few minimal examples:

```
example[10] =
  do {
    a ← pmf-of-set {0, 1};
    return-pmf (a+1)
  }
```

This example represents an algorithm, where a is uniformly chosen from the set $\{0, 1\}$ and the successor of a is then returned. On the other hand from the probabilistic perspective, it makes sense to think of $a \mapsto a + 1$ as a random variable on the probability space $\{0, 1\}$ and the entire expression represents the distribution of that random variable. Indeed it is easy to show that:

```
lemma[10] example = pmf-of-set {1, 2}
```

Note in general, the term *pmf-of-set A* is a probability space, which assigns the same probability to each element of A if it is a finite, non-empty set. The abbreviation PMF stands for *probability mass function*, which are a subtype of probability spaces in Isabelle with the condition that the σ-algebra is *discrete*, more precisely, the σ-algebra must be the universe of the type forming the events. They have the advantage that all functions defined on these probability spaces are automatically measurable. The disadvantage is that the support of a measure defined on a discrete σ-algebra must necessarily be a countable set. Hence, probability mass functions are a well-suited model for randomized algorithms, where the probability spaces will be discrete anyway.

Let us investigate the case when multiple random operations are composed:

```
do[10] {
  a ← pmf-of-set {0, 1};
  b ← pmf-of-set {2, 3};
  return-pmf (a, b)
}
```

The resulting probability space is the product space $\{0, 1\} \times \{2, 3\}$, again with uniform probability. This means independent sequential composition can be though of as the construction of the product probability space. However, things can become more complex, when earlier random variables influence later random operations:

```
do[10] {
  a ← pmf-of-set {1, 2};
  b ← pmf-of-set {0..<a};
  return-pmf (a, b)
}
```

Here, the resulting probability space is a dependent sum: $\bigcup_{a \in \{1,2\}} \{a\} \times \{0, .., a - 1\}$. The probability of the pair $(0, 1)$ is $\frac{1}{2}$ while the probability of the pairs $\{(0, 2), (1, 2)\}$ is $\frac{1}{4}$. Note that the components are not independent random variables any more. On a deeper level, these probability spaces are expressions with two combinators:

[10] This is example code. It is not part of the accompanying formalization [33].

- *return-pmf*: This operation returns the Dirac measure, the probability of an event is 1 exactly if it contains the argument of *return-pmf*.
- *bind-pmf*: This operation builds a new probability space, using a first probability space Ω_1 and a function that maps each element of $x \in \Omega_1$ to a new probability space $\Omega_2(x)$. We can write this as $\Omega_1 \gg\!= \Omega_2$, where the probability of an event E in $\Omega_1 \gg\!= \Omega_2$ is: $\int_{\Omega_1} P_{\Omega_2(\omega)}(E)\, d\omega$.[11] Note: In the *do*-notation the bind operator is implicitly inserted, whenever there is a semicolon.

Because the algorithms for the frequency moments are one-pass streaming algorithms, they are represented using three functions over the Giry monad. First, an initialization function that sets up the initial state based on the parameters: the desired relative accuracy δ, the desired success probability ε, the size of the universe of the stream elements n. For simplicity, we assume the stream elements are represented as natural numbers in $\{0, .., n-1\}$. Note: A state of these algorithms is also called *sketch* or synopsis in the context of frequency moments. Second an update function that processes a single stream element and updates the state. And finally a result function that computes an approximation of the frequency moment. We can then describe the distribution of the algorithm for the stream elements $a_1, \ldots, a_m$ like:

```
do[10] {
  s_0 ←init δ ε n;
  s_1 ←update a_0 s_0;
  s_2 ←update a_1 s_1;
  ...
  s_m ←update a_{m-1} s_{m-1};
  result s_m
}
```

which can be written more succinctly as:

$$\mathit{fold}\ (\lambda a\ s.\ s \gg\!= \mathit{update}\ a)\ \mathit{as}\ (\mathit{init}\ \delta\ \varepsilon\ n) \gg\!= \mathit{result} \tag{4}$$

The following snippet is the theorem in the formalization that establishes the correctness property for the F_0 estimation algorithm. The theorems for the correctness of the F_2 and F_k estimation algorithms are analogous:

```
theorem f0-alg-correct:[6]
  assumes ε ∈ {0<..<1}
  assumes δ ∈ {0<..<1}
  assumes set as ⊆ {0..<n}
  defines M ≡ fold (λa state. state ⤜ f0-update a) as (f0-init δ ε n) ⤜ f0-result
  shows P(ω in measure-pmf M. |ω − F 0 as| ≤ δ * F 0 as) ≥ 1 − ε
```

The first two assumptions establish that ε and δ are strictly between 0 and 1. The next assumption is the requirement that the stream elements are elements in $\{0, \ldots, n-1\}$. M is defined – as discussed above (Equation 4) – as the distribution of the estimation algorithm, described by the three functions *f0-init*, *f0-update* and *f0-result* after processing the stream elements *as*. The final line establishes that the relative error of the estimate is less than δ with probability $1 - \varepsilon$. The term *F k as* refers to the actual k-th frequency moment of the stream *as* as defined in Equation 1. While this is exciting, it is also necessary to verify the space usage of the algorithms, which is going to be discussed in the next section.

[11] Especially, in the case, where the probability spaces are not countable, the construction of the resulting probability space is non-trivial. See also Eberl et al. [16] for more details on this and the Giry Monad.

2.4 Verification of the space complexity

Because the algorithms are shallowly embedded as functions in the logic, it is not directly possible to verify the memory-complexity of the algorithms. A possible solution would be to represent the algorithm within a formalized machine model and show its equivalence to the high-level representations in the logic. This is for example discussed by Myreen [39, §1].

However – because of the representation of the streaming algorithms using the three functions as described above – it is still possible to rigorously establish a bound on the memory requirements of the states the algorithms reach before and after processing each element, i.e., the sketch size.[12] For this, I decided to build an encoding of the states into bit sequences and use the length of it as a measure of the size of the data structure. In general any injective function from the state space to lists of booleans would form such an encoding and thus would provide an upper bound on the space usage, i.e., an idea would be to show a statement of the form:

$$\exists f.\ \mathit{inj}\ f \wedge \forall s \in S\ \varepsilon\ \delta\ n.\ \mathit{length}\ (f\ s) \leq F\ \varepsilon\ \delta\ n \qquad *^{10}$$

where $S\ \varepsilon\ \delta\ n$ denotes the set of states the algorithm may reach for all possible inputs and internal random operations with the provided parameters ε, δ, n and F denotes the upper bound on the space usage in bits to be shown.[13]

It turns out that this is too restrictive. The condition *inj f* requires the function to be injective on the entire universe, i.e., all possible elements of the type of the state space, even though the reachable states might be a smaller set. An example where this is an issue is when the type of the state is not a countable set. For example coefficients of the finite field $GF(p)$ can be encoded using $\ln p$ bits, but their type is: *int set*, representing each element of the field as a congruence class. A fix for this problem is allowing the encoding to be a partial function, which still needs to be injective on its domain, and requiring that the reachable states are in the domain of the function:

$$\exists f.\ \mathit{inj\text{-}on}\ f\ (\mathit{dom}\ f) \wedge \forall s \in S\ \varepsilon\ \delta\ n.\ s \in \mathit{dom}\ f \wedge \mathit{length}\ (\mathit{the}\ (f\ s)) \leq F\ \varepsilon\ \delta\ n \qquad *^{10} \qquad (5)$$

Note: Partial functions are represented using the option type: *'a ⇒ 'b option*. If x is outside of the domain of f then $f\ x = \mathit{None}$. If $f\ x = \mathit{Some}\ y$ then x is in the domain and the value of f at x is y. Moreover, *dom f* denotes the domain and *the* $(f\ x)$ is the value of the partial function if $x \in \mathit{dom}\ f$. With the introduction of the new function *bit-count*:

```
fun bit-count :: bool list option ⇒ ereal where[6]
  bit-count None = ∞ |
  bit-count (Some x) = length x
```

the conjunction on the right hand side of Equation 5 can be expressed more concisely as *bit-count* $(f\ s) \leq F\ \varepsilon\ \delta\ n$, i.e., a finite upper bound on bit-count automatically implies that s must have been in the domain of f.[14] The following snippet is the actual result for the space-complexity of the F_0-Algorithm in the formalization [33]:

[12] It is informally easy to see that the capacity bounds are not exceeded even during an update operation.
[13] Note that the order of the quantifiers is important.
[14] Subsection 2.5 contains a second reason for setting the *bit-count* of unencodable values to ∞.

```
definition encode-f0-state :: f0-state ⇒ bool list option where [omitted ...]
fun f0-space-usage :: (nat × rat × rat) ⇒ real where [omitted ...]

lemma inj-on encode-f0-state (dom encode-f0-state)

theorem f0-exact-space-usage:
  assumes ε ∈ {0<..<1}
  assumes δ ∈ {0<..<1}
  assumes set as ⊆ {0..<n}
  defines M ≡ fold (λa state. state ⤜ f0-update a) as (f0-init δ ε n)
  shows AE ω in M. bit-count (encode-f0-state ω) ≤ f0-space-usage (n, ε, δ)
```

Instead of showing an existence result like in Equation 5, the encoding function *encode-f0-state* is defined explicitly and that it is injective on its domain is verified in a separate lemma. The function *f0-space-usage* is a pure arithmetic expression in terms of n, ε and δ. The theorem establishes that the reachable states are encodable using the encoding function and meet the memory bounds of *f0-space-usage*. The syntax *AE ω in M* stands for *almost everywhere*, this means the predicate must be true up to a set of probability 0. Since the states of the algorithms are probability distributions, the reachable states constitute the elements of M that have a non-zero probability. Besides providing an explicit bound using the function *f0-space-usage*, the next theorem concludes with the asymptotic space complexity.

```
theorem f0-asymptotic-space-complexity:[6]
  f0-space-usage ∈ O[at-top ×_F at-right 0 ×_F at-right 0](λ (n, ε, δ). ln (1 / ε) *
  (ln n + 1 / δ^2 * (ln (ln n) + ln (1 / δ))))
```

This means that the space usage is bounded by $C \ln(\varepsilon^{-1}) \left(\ln n + \delta^{-2}(\ln(\ln n)) + \ln(\delta^{-1})\right)$ for some constant C for sufficiently large n, ε^{-1} and δ^{-1}.

2.5 A flexible encoding library

As noted in the previous section, the memory requirement of the states the algorithms reach is measured by encoding them into bit strings. To achieve this, I have built a small flexible encoding library [31] for Isabelle data structures comprised of encoding functions for primitive types and a set of combinators to handle structured data types. The encoding functions are *prefix-free*, i.e., if $f\ x$ is a (not necessarily strict) prefix of $f\ y$ then $x{=}y$. This implies in particular that they are injective, which implies that the resulting bit strings can be decoded. Moreover prefix-freeness has the useful property that it is preserved under concatenation, i.e., if f, g are prefix-free, then $\lambda(x,\ y).\ f\ x\ @\ g\ y$ is also.[15] The symbol @ stands for the concatenation of two bit strings. To see why this works, let us observe another characterization of prefix-free functions:

$$f\ x_1\ @\ y_1 = f\ x_2\ @\ y_2 \Longrightarrow x_1 = x_2 \wedge y_1 = y_2 \qquad *^{10}$$

with which it is easy to conclude, for example if f, g are prefix-free:

$$\begin{aligned} &f\ x_1\ @\ g\ y_1\ @\ z_1 = f\ x_2\ @\ g\ y_2\ @\ z_2 \Longrightarrow \\ &x_1 = x_2 \wedge g\ y_1\ @\ z_1 = g\ y_2\ @\ z_2 \Longrightarrow \\ &x_1 = x_2 \wedge y_1 = y_2 \wedge z_1 = z_2 \end{aligned} \qquad *^{10}$$

[15] The approach of using prefix-free codes [on the byte-level] is commonly utilized in many serialization libraries. See for example [10, §4.2.1].

Table 2 Encoding functions for primitive types and combinators.

Sym	Description	Bit Count
N_e	Natural numbers[16]	*bit-count* $(N_e\ n) \leq 2 * log\ 2\ (real\ n{+}1) + 1$
I_e	Integers	*bit-count* $(I_e\ x) \leq 2 * log\ 2\ (\lvert x\rvert{+}1) + 3$
F_e	Floating point numbers	*bit-count* $(F_e\ (float\text{-}of\ (m * 2\ powr\ e)) \leq$ $6 + 2 * (log\ 2\ (\lvert m\rvert + 2) + log\ 2\ (\lvert e\rvert + 1))$
$\times_e$	Tuples	*bit-count* $((e \times_e f)\ (x,y)) =$ *bit-count* $(e\ x) +$ *bit-count* $(f\ y)$
$\bowtie_e$	Dependent tuples	*bit-count* $((e \bowtie_e f)\ (x,y)) =$ *bit-count* $(e\ x) +$ *bit-count* $(f\ x\ y)$
L_e	Lists	*bit-count* $(L_e\ f\ xs) = (\sum x \leftarrow xs.$ *bit-count* $(f\ x){+}1) + 1$
S_e	Finite sets	*finite* $S \Longrightarrow$ *bit-count* $(S_e\ e\ S) = (\sum x \in S.$ *bit-count* $(e\ x){+}1){+}1$
$\rightarrow_e$	Functions defined on *set xs*	$f \in$ *extensional* $(set\ xs) \Longrightarrow$ *bit-count* $((xs \rightarrow_e e)\ f) = (\sum x \leftarrow xs.$ *bit-count* $(e\ (f\ x)))$

Table 2 summarizes the encoding functions and combinators that are used to build encodings for the states. An interesting property about the definition of *bit-count* is that the equation:

$$bit\text{-}count\ ((e_1 \times_e e_2)\ (x_1,x_2)) = bit\text{-}count\ (e_1\ x_1) + bit\text{-}count\ (e_2\ x_2)$$

holds unconditionally, e.g., even if x_1 and/or x_2 are not in the domain of e_1/e_2. This is because of the facts: $\infty + x = \infty$, $x + \infty = \infty$ if $x \geq 0$ in the extended reals, and that a pair is in the domain of $e_1 \times_e e_2$ if and only if its first component is in the domain of e_1 and its second component in the domain of e_2. Similar equations hold unconditionally for the other combinators, which means that in the proof of the theorems for space usage it was possible to show the property that the state is part of the domain of the encoding function, as well as the actual space bound using the same reasoning step.

A tempting design question is whether it would be possible to derive such an encoding fully automatically. However, in this use case, it is important to *choose* an efficient encoding for the reachable states of the algorithms.

A key benefit of this approach is that it allows verification of the space complexity of data structures used in high-level specifications of algorithms close to the mathematical representation, for example, using cosets and indexed products.

3 The Algorithms

3.1 Frequency Moment 0

The original plan I had was to formalize Algorithm 3 from Bar-Yossef et al. [5]. It is the solution with the best space-time trade-off in the paper. The authors describe it as a modification of the Gibbons-Tirthapura algorithm. Briefly, it stores only the elements of the stream that have a given count l of leading zeros (or more) in the binary representation of their hash values. The value l is initialized to 0 at the beginning and is incremented during the run of the algorithm, such that the set of filtered elements fit in the allocated space. The modification by Bar-Yossef is to avoid storing the elements themselves in the state, but only a hash of them, for that purpose they introduce a second hash function.

[16] Prefix free codes for natural numbers are also called universal codes. See for example Elias [17].

During the formalization, it became apparent that a simpler algorithm is possible with the same space and amortized runtime cost as that one and an improved worst-case runtime. It uses only a single hash function and does not require its range to be a 2 power. It is based on the first algorithm described in the same paper but with an added rounding operation to the hashed values. Since that kind of algorithm is called KMV synopsis or sketch in newer publications [6, 42, 44], where the abbreviation KMV stands for *k-minimum value*[17], it makes sense to call the new algorithm Rounding-KMV. The general principle of KMV algorithms is to use the t-th smallest element of the hashed stream elements, where t is chosen according to the required accuracy parameter. This can be done by keeping track of the smallest t hashed stream elements during the course of the algorithm and using the maximum for the estimation step. Table 3 summarizes the algorithms discussed in this subsection.

Table 3 Algorithms mentioned in this subsection.

Algorithm	Space usage with respect to δ, n[18]	Hash space	Based on
Gibbons–Tirthapura [21]	$O\left(\delta^{-2}\ln n\right)$	$GF(2^e)$	-
Algorithm 3 [5]	$O\left(\delta^{-2}(\ln\ln n + \ln\delta^{-1}) + \ln n\right)$	$GF(2^e)$	Gibbons–T.
Algorithm 1 [5]	-	$[0,1] \subset \mathbb{R}$	-
Standard KMV (below)	$O\left(\delta^{-2}\ln n\right)$	$GF(p)$	Algorithm 1 [5]
Rounding KMV (below)	$O\left(\delta^{-2}(\ln\ln n + \ln(\delta^{-1})) + \ln n\right)$	$GF(p)$	Standard KMV

Let us first review the correctness proof for the standard KMV algorithm[19] (without rounding operation) with the Carter–Wegman hash family. This means we need to investigate the distribution of the t-th smallest element. For that let $a_1, \ldots, a_m$ be the elements of the stream and $a_i \in \{0, .., n-1\}$, let $A = \{a_1, \ldots, a_m\}$ be the set of distinct elements in the stream. Note that: $F_0 = |A|$, but $m \geq F_0$, since the a_i are not necessarily distinct. Let $p \geq \max(n, 11)$ be a prime and let h be uniformly chosen from the 2-universal Carter–Wegman hash family.

It makes sense to investigate the two closely related random variables X_t and $X_t^{\#}$ denoting the t-th smallest element of the hashed stream elements $H = \{h(a) \mid a \in A\}$, where the second one treats distinct elements of A mapped to the same value by the hash function as separate elements, while the first one does not. See also Figure 3 for an example, where X_t and $X_t^{\#}$ differ. More precisely X_t and $X_t^{\#}$ are the unique random variables fulfilling the following conditions:

$$X_t \in H \qquad\qquad X_t^{\#} \in H$$
$$|\{x \in H \mid x < X_t\}| = k-1 \qquad |\{a \in A \mid h(x) < X_t^{\#}\}| = k-1$$

While it is more space-efficient to compute X_t; it is mathematically easier to investigate the distribution of $X_t^{\#}$, in particular, if [and only if] there are at least t elements $a \in A$ such that $h(a) < u$, then $X_t^{\#}$ must be strictly smaller than u as well.

[17] Since k is being used to denote the order of the frequency moment in this work; the letter t is for the index of the rank instead.
[18] If ε is not assumed to be constant all complexities need to be multiplied by another factor of $ln(\varepsilon^{-1})$
[19] Bar-Yossef et al. provide a proof for the idealized case, where the hash family maps into $[0, 1]$. The proof here differs from theirs to cover the more realistic case with the Carter–Wegman hash family.

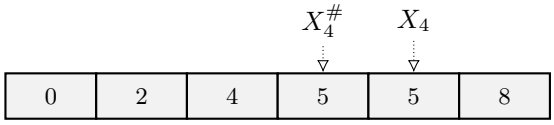


Figure 3 An example for a collision in the application of the hash function, leading to a difference between X_t and $X_t^\#$. The numbers in the boxes denote hash values of distinct elements of A.

The expectation and variance of the random variable Q_u that counts the number of elements $a \in A$ hashed to a value strictly less than u is easy to determine:

$$Z_{u,a} := \begin{cases} 1 & \text{if } h(a) < u \\ 0 & \text{otherwise.} \end{cases} \qquad Q_u := \sum_{a \in A} Z_{u,a}$$

Note that $\mathrm{Var}(Z_{u,a}) \leq \frac{u}{p}$ and $\mathbb{E}(Z_{u,a}) = \frac{u}{p}$. Because the $Z_{u,a}$ are pairwise independent, we can apply Bienaymé's identity to conclude that $\mathrm{Var}(Q_u) \leq \frac{|F_0|u}{p}$ and $\mathbb{E}(Q_u) = \frac{|F_0|u}{p}$.

Now, using our observation from above, we can estimate the probability that $X_t^\#$ is less than u by the probability that $Q_u \geq t$. Since we know expectation and variance of Q_u, Chebyshev's inequality bounds the probability of $Q_u \geq t$. A similar reasoning also works for a lower bound on X_t.

Another observation, we can make is that $X_t^\# = X_t$ if h is injective. The probability that h is not injective is $\frac{1}{p} \leq \frac{1}{9}$ with the 2-universal Carter–Wegman hash family and the lower limit for p. The overall proof works by estimating the probabilities of the following events:
Case 1 The function h is not injective.
Case 2 Less than t elements are hashed to values below $v = \lfloor tp(1-\delta)^{-1}F_0^{-1} \rfloor$, i.e. $Q_v < t$.
Case 3 At least t elements are hashed to values below $u = \lceil tp(1+\delta)^{-1}F_0^{-1} \rceil$, i.e. $Q_u \geq t$.
and showing that the probability of each of these are strictly less than $\frac{1}{9}$. On the other hand if neither of these events occur, i.e., with probability at least $\frac{2}{3}$: h is injective and tpX_t^{-1} is an approximation of F_0 with a relative error of δ.

For the curious reader: This works for $t \geq 6\delta^{-2}$. The case, where $|A| < t$ – i.e., if there is no t-th smallest element – needs to be handled separately. But in that case, an implementation would simply return the count of distinct elements observed so far.

The above strategy requires $O(\delta^{-2} \ln n)$ bits of memory to store the state. Using a rounding operation, it is possible to improve the space complexity to $O(\delta^{-2}(\ln \delta^{-1} + \ln(\ln n)) + \ln n)$ bits of memory[20], which is the new rounding-KMV variant. Let us denote this rounding operation by $\lfloor \cdot \rfloor_r$, which can be defined by:

$$\lfloor x \rfloor_r := \tau(r - \lfloor \log_2 |x| \rfloor, x) \text{ where } \tau(e, x) = 2^{-e} \lfloor 2^e x \rfloor$$

if $x \neq 0$, otherwise: $\lfloor 0 \rfloor_r := 0$. In particular: $\lfloor x \rfloor_r \leq x < \lfloor x \rfloor_r + |x|(1 + 2^{-r})$. Expressed differently $\lfloor x \rfloor_r$ is the largest binary floating point number with a mantissa of at most r bits smaller or equal to x.

Now the new variant uses $\tilde{h}(a) = \lfloor h(a) \rfloor_r$ instead of h. The proof is similar to the above version, however the following issues need to be taken into account:

1. the additional accuracy error introduced by the rounding operation,
2. even if h is injective, collisions due to rounding are possible, i.e., $\tilde{h}$ may still not be injective and hence $\tilde{X}_t^\#$ may differ from $\tilde{X}_t$.

[20] Note that if $\delta^{-2} > n$, the trivial algorithm which tracks for each element of U whether it occurred in the stream using n bits outperforms either of these randomized algorithms. Hence, when judging which complexity is better it makes sense to assume $\delta^{-2} \in O(n)$.

The first issue can be solved by choosing $\tilde{r}$ and $\tilde{t}$ large enough such that the combined relative error due to rounding and the statistics of the t-th smallest element remain below δ. My initial idea for solving the second problem was to choose $\tilde{r}$ large enough, such that the probability of a collision due to rounding is bounded. It however turns out, that that condition would require a choice of $\tilde{r} \in O(\ln n)$, which is too high.

A closer look at the problem reveals that $\tilde{X}_t = \tilde{X}_t^{\#}$ as long as there is no collision within the smallest t hashed values. Estimating the probability of the latter event, requires another insight: It is actually not necessary to bound the probabilities of the events Case 1 to 3 separately.

In particular it is enough, if the probability of Case 2 and 3 happening is bounded by $\frac{2}{9}$ and if the probability of Case 1 happening under the condition that Case 2 and 3 are not is bounded by $\frac{1}{9}$.

Stated differently, it is enough to bound the probability of $\tilde{X}_t \neq \tilde{X}_t^{\#}$ only in the case $\tilde{X}_t^{\#} < v$. Thus it is enough to estimate the probability of a collision due to rounding within $[0, v)$ – the range the first t elements will be hashed to when $\tilde{Q}_v \geq t$. In the formalized proof, this is accomplished by making sure $\tilde{p} \geq 18$ and $\tilde{r} = 4\lceil \log_2(\delta^{-2}) \rceil + 23$ and bounding the probability that h is injective by $\frac{1}{18}$ and the probability of a collision due to rounding in the range $[0, v)$ by the same value.

In the accompanying formalization [33, Appendix A] I have included a detailed "hand-written" proof with the same reasoning as the formalized proof for interested readers.

3.2 Formalization of the F_0 algorithm

The following snippet contains the formalized version of the full algorithm:

```
fun f0-init :: rat ⇒ rat ⇒ nat ⇒ f0-space pmf where[6]
  f0-init δ ε n =
    do {
      let s = ⌈−18 * ln ε⌉;
      let t = ⌈80 / δ²⌉;
      let p = prime-above (max n 19);
      let r = 4 * ⌈log 2 (1 / δ)⌉ + 23;
      h ← prod-pmf {0..<s} (λ-. pmf-of-set (bounded-degree-polynomials (ZFact p) 2));
      return-pmf (s, t, p, r, h, (λ- ∈ {0..<s}. {}))
    }

fun f0-update :: nat ⇒ f0-space ⇒ f0-space pmf where
  f0-update x (s, t, p, r, h, sketch) = return-pmf (s, t, p, r, h, λi ∈ {0..<s}.
      least t (insert (float-of (truncate-down r (hash p x (h i)))) (sketch i)))

fun f0-result :: f0-space ⇒ rat pmf where
  f0-result (s, t, p, r, h, sketch) = return-pmf (median s (λi ∈ {0..<s}.
      (if card (sketch i) < t then (card (sketch i)) else t * p / (Max (sketch i)))))
```

As explained in Subsection 2.3 the algorithm is formalized using three functions, an initialization function that sets up the state of the algorithm, an update function that updates the state processing a stream element and the result function that returns an estimate for the frequency moment using the state. The parameters of the initialization algorithm are: δ the required relative accuracy; ε the required success probability; and an upper bound n on the stream elements. As explained in Subsection 2.2 the algorithm runs $s = \lceil 18 \ln(\varepsilon^{-1}) \rceil$ independent copies of the rounding KMV algorithm to achieve the desired success probability, and computes the median of the individual estimates in the *f0-result* function. The algorithm

determines the t-th smallest hashed stream element, where $t = \lceil 80\delta^{-2} \rceil$. The function *prime-above* x returns a prime in the range $\{x, \ldots, 2x+2\}$ and is used to select the field over which the polynomials for hashing are chosen. The term *ZFact* p refers to the simple prime field $GF(p)$. The function *truncate-down* r is the rounding method $\lfloor \cdot \rfloor_r$ that was described above. To understand the algorithm a little bit better. The initialization function determines the parameters s, t, p, r and randomly selects s polynomials $h_0, \ldots, h_{s-1}$ of degree less than 2 over the field $GF(p)$. And sets up s empty sets, which will later contain the t smallest hashed stream elements. The state is a 6-tuple composed of the parameters, the hash functions and the sets. In the update step, for each of the hash polynomials, the function computes the rounded hashed value of the stream element and inserts the element into the corresponding set. If the set contains more than t elements, the largest element from the set will be removed. For the estimation, the function checks each of the s sets, if it contains at least t elements, it returns the inverse of the maximal element multiplied by tp, i.e., tp times the inverse of the t-th smallest element. If the set does not contain t elements – there is no t-th smallest element – but in that case the cardinality of the set is a good approximation of F_0 hence the cardinality is used as an approximation.

3.3 Frequency Moment 2

The formalized algorithm for the second frequency moment is based on the solution described in Section 2.2 by Alon et al. [3]. The key idea is to choose h from a 4-universal hash-family with (equiprobable) values $\{-1, 1\} \in \mathbb{Z}$. The algorithm then returns:

$$X = \left(\sum_{i=1}^{m} h(a_i) \right)^2 . \tag{6}$$

Note that:

$$\begin{aligned} \mathbb{E}\, X &= \mathbb{E} \left(\sum_{u \in U} C(u,a) h(u) \right)^2 = \sum_{u,v \in U} C(u,a) C(v,a)\, \mathbb{E}\, h(u) h(v) \\ &= \sum_{u \in U} C(u,a)^2\, \mathbb{E}\, h(u)^2 + \sum_{u,v \in U} C(u,a) C(v,a)\, \mathbb{E}\, h(u)\, \mathbb{E}\, h(v) = F_2 \end{aligned}$$

where $C(u,a)$ is the count of occurrences of u in the stream a. The last equality follows from $\mathbb{E}\, h(u)^2 = 1$ and $\mathbb{E}\, h(u) = 0$ for $u \in U$. An interesting fact is that the sum over $u, v \in U$ could only be evaluated by splitting it into two sums, where the first sum is comprised of the cases where the indices are equal and the second sum is comprised of the cases where the indices are distinct. This is because $h(u)$ and $h(v)$ are independent only if $u \neq v$.

A similar split has to be done for the evaluation of the variance of X where the summation is over four variables. This results in 15 possible ways the indices can form equivalence relations. For the latter I have built a library with which it is possible to automatically split such a sum into terms for each partition of its index variables [32, §5]. With that approach the estimation of the variance happens automatically through symbolic evaluation within Isabelle.

By running independent copies of the algorithm in parallel and computing the median of means, it is possible to return an approximation with the required success probability and accuracy.

The only difference between the method presented by Alon et al. [3] and the formalization in this work is that the algorithm is adapted to work with simple prime fields. While it is impossible to obtain a two-valued uniform distribution (if $p \geq n \geq 3$) a closer look at the proof from Alon et al. reveals that the actual requirements for the hash family are:

1. $\mathbb{E}\ h(a) = 0$, $\mathbb{E}\,(h(a))^2 = 1$ and $\mathbb{E}\,(h(a))^4 \leq 3$.
2. The hash family is 4-wise independent.

If h' is uniformly chosen from the 4-universal Carter–Wegman hash family then h defined by:

$$\begin{aligned} h &: U \to \mathbb{R} \\ h(x) &:= \begin{cases} (p^2-1)^{-\frac{1}{2}}(p-1) & \text{if } h'(x) \text{ is even} \\ (p^2-1)^{-\frac{1}{2}}(-p-1) & \text{otherwise} \end{cases} \end{aligned}$$

fulfills the above requirements. Since the factor $(p^2-1)^{-\frac{1}{2}}$ is constant, it can be factored out of the sum and the squaring operation in Equation 6, so that the resulting algorithm can be implemented without using real arithmetic.

3.4 Frequency Moment k for $k \geq 3$

The formalization of the algorithm for the k-th frequency moment for $k \geq 3$ is exactly the same as the solution described in Section 2.1 by Alon et al. [3]. Contrary to the previous algorithms it does not rely on hash families. Instead a random position $i \in \{1, \ldots, m\}$ of the stream $a_1, \ldots, a_m$ is selected and the count of occurrences of that stream element from that point on is counted, whose value is described by the following random variable:

$$X(i) = |\{j \in \{i, \ldots, m\} \mid a_j = a_i\}|$$

The estimate for the k-th frequency moment is then $R(i) := m(X(i)^k - (X(i)-1)^k)$. Note that:

$$\begin{aligned} \mathbb{E}\ R &= \mathbb{E}\,(m(X^k - (X-1)^k)) = \sum_{i=1}^{m} X(i)^k - (X(i)-1)^k \\ &= \sum_{u \in U} \sum_{v=1}^{C(u,a)} v^k - (v-1)^k = \sum_{u \in U} C(u,a)^k = F_k, \end{aligned}$$

where $C(u,a)$ denotes the count of occurrences of u in the stream $a_1, \ldots, a_m$. The evaluation of the variance of the random variable is a longer calculation resulting in $\text{Var}\ R \leq F_k^2 k n^{1-1/k}$. Similar to the previous algorithm by running independent copies of the algorithm in parallel and computing the median of means, it is possible to improve the accuracy and success probability.

A remaining problem to solve is that the algorithm has to choose a random index uniformly from the stream, without knowing the length of the stream in advance.

Alon et al. [3] describe a refined version of the algorithm that solves the problem: A random boolean is chosen at every update step, which is true with probability $\frac{1}{l+1}$, where l is the number of elements that were processed before. If the boolean is true the algorithm resets the counter to 1 and chooses the element at the current index to count. They then show, that the position of the last reset is uniformly distributed over the length of the stream. The accompanying formalization [33] verifies this second version of the algorithm.

4 Related Work

In 2019 Affeldt et al. [1] formalized two tree-based succinct data structures in Coq, one of them being dynamic. They achieve their results by defining the operations on a high-level inductive data structure and a low-level version implemented on bit arrays and establish correspondence. A similar approach could also be applied here to avoid the need of the encoding functions as discussed in Subsection 2.5.

Eßmann et al. [18] formally verify non-deterministic approximation algorithms for NP-complete problems in Isabelle, such as maximum independent set. In their work, they do not need to reason with probability distributions, as the correctness of the investigated algorithms follows from combinatorial arguments.

In 2020 Gopinathan and Sergey [23] formally verified Bloom Filters using Coq. They rely on a deep embedding and similar to this work rely on probability theory and reasoning about independent random variables.

Tassarotti et al. [46] formally verify an ML procedure learning a classifier using Lean. As in this work, they also represent their algorithms using the Giry-monad.

Bao et al. [4] also tackle Bloom Filters as an application of their separation logic for reasoning about negative dependence. Negative dependence is a weaker property about random variables than independence, i.e., more sets of random variables are negatively dependent, but fewer results about independent random variables apply to negatively dependent variables. For example the property is preserved by composition with monotone functions only. They realize that the random variables induced on the bit vector are negatively dependent; greatly simplifying the proofs about the false-positive rate of bloom filters.

Eberl et al. [15] verify the average runtime of randomized quicksort and derive the expected structure of binary tree structures. They rely on the formalization of the Giry monad in Isabelle/HOL. The formalization approach for randomized algorithms in this work is based on their work.

As far as I can tell there is no prior publication on the formalization of randomized algorithms where derandomization and/or amplification techniques are used.

5 Conclusion and Future Work

While the primary focus of this work, was the formal verification of the algorithms for frequency moments – I could obtain simpler versions of the known algorithms. In particular it was possible to avoid higher order prime fields. The algorithm for F_0 matches the complexity of the best solution from [5] but its design is considerably simpler. Requiring only one hash family instead of two. Most solutions in current production database systems are verified empirically and/or rely on unverified statistical model assumptions, because of the complexity of the known correct solutions [26]. These simplifications may lead to industrial applications of these rigorously verified algorithms.

An interesting direction for future work would be the formalization of the newer results that match the lower bounds [28, 30]. Another interesting problem is the extension of the frequency moments to fractional powers, for which algorithms have been derived in [13, 29].

The algorithms presented here for F_0 and F_2 can be augmented with a merge operation, i.e., it would be possible to run the algorithm in parallel for different sections of the data stream and merge the sketches to obtain an approximation of the frequency moment for the entire stream. It would make sense to extend the obtained theorems to include the merge operation.

Another improvement would be to use probable primes[21] instead of requiring exact primes. For this, the algorithms would need to be reparameterized such that the combined failure probability of the algorithm and the false-positive rate of the primality test remains below the required maximum failure probability ε.

[21] Probabilistic primality tests have been formalized in Isabelle by Stüwe and Eberl [45].

References

1 Reynald Affeldt, Jacques Garrigue, Xuanrui Qi, and Kazunari Tanaka. Proving Tree Algorithms for Succinct Data Structures. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:19, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ITP.2019.5.

2 Archive of Formal Proofs. https://isa-afp.org. Accessed: 2021-11-13.

3 Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999. doi:10.1006/jcss.1997.1545.

4 Jialu Bao, Marco Gaboardi, Justin Hsu, and Joseph Tassarotti. A separation logic for negative dependence. *Proceedings of the ACM on Programming Languages*, 6:57:1–57:29, 2022. doi:10.1145/3498719.

5 Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In *Randomization and Approximation Techniques in Computer Science*, pages 1–10. Springer Berlin Heidelberg, 2002. doi:10.1007/3-540-45726-7_1.

6 Kevin Beyer, Peter J. Haas, Berthold Reinwald, Yannis Sismanis, and Rainer Gemulla. On synopses for distinct-value estimation under multiset operations. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 199–210, New York, 2007. doi:10.1145/1247480.1247504.

7 Lakshminath Bhuvanagiri, Sumit Ganguly, Deepanjan Kesh, and Chandan Saha. Simpler algorithm for estimating frequency moments of data streams. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, SODA '06, pages 708–713, USA, 2006. Society for Industrial and Applied Mathematics. doi:10.5555/1109557.1109634.

8 Julian Biendarra and Manuel Eberl. Bertrand's postulate. *Archive of Formal Proofs*, January 2017. , Formal proof development. URL: https://isa-afp.org/entries/Bertrands_Postulate.html.

9 Jarosław Błasiok. Optimal streaming and tracking distinct elements with high probability. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 2432–2448, 2018. doi:10.1137/1.9781611975031.156.

10 Carsten Bormann and Paul E. Hoffman. Concise Binary Object Representation (CBOR). RFC 8949, December 2020. doi:10.17487/RFC8949.

11 R. C. Bose and K. A. Bush. Orthogonal arrays of strength two and three. *The Annals of Mathematical Statistics*, 23(4):508–524, 1952. doi:10.1214/AOMS/1177729331.

12 Vladimir Braverman, Jonathan Katzman, Charles Seidell, and Gregory Vorsanger. An Optimal Algorithm for Large Frequency Moments Using $O(n^{1-2/k})$ Bits. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2014)*, volume 28 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 531–544, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.APPROX-RANDOM.2014.531.

13 Vladimir Braverman, Emanuele Viola, David P. Woodruff, and Lin F. Yang. Revisiting Frequency Moment Estimation in Random Order Streams. In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:14, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ICALP.2018.25.

14 David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and Sridhar Seshadri. Practical skew handling in parallel joins. In *Proceedings of the 18th VLDB conference*, 1992.

15 Manuel Eberl, Max W. Haslbeck, and Tobias Nipkow. Verified analysis of random binary tree structures. *J. Autom. Reason.*, 64(5):879–910, 2020. doi:10.1007/s10817-020-09545-0.

16 Manuel Eberl, Johannes Hölzl, and Tobias Nipkow. A verified compiler for probability density functions. In *Programming Languages and Systems*, pages 80–104. Springer Berlin Heidelberg, 2015. doi:10.1007/978-3-662-46669-8_4.

17 P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975. doi:10.1109/TIT.1975.1055349.

18 Robin Eßmann, Tobias Nipkow, and Simon Robillard. Verified approximation algorithms. *Automated Reasoning*, 12167:291–306, 2020. doi:10.46298/lmcs-18(1:36)2022.

19 Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete Mathematics & Theoretical Computer Science*, pages 137–156, 2007.

20 Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985. doi:10.1016/0022-0000(85)90041-8.

21 Phillip B. Gibbons and Srikanta Tirthapura. Estimating simple functions on the union of data streams. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '01, pages 281–291, 2001. doi:10.1145/378580.378687.

22 Frédéric Giroire. Order statistics and estimating cardinalities of massive data sets. *Discrete Applied Mathematics*, 157(2):406–427, 2009. doi:10.1016/j.dam.2008.06.020.

23 Kiran Gopinathan and Ilya Sergey. Certifying certainty and uncertainty in approximate membership query structures. *Computer Aided Verification*, 12225:279–303, 2020. doi:10.1007/978-3-030-53291-8_16.

24 Sebastien Gouezel. Lp spaces. *Archive of Formal Proofs*, October 2016. , Formal proof development. URL: https://isa-afp.org/entries/Lp.html.

25 Benjamin Gufler., Nikolaus Augsten., Angelika Reiser., and Alfons Kemper. Handling data skew in MapReduce. In *Proceedings of the 1st International Conference on Cloud Computing and Services Science - CLOSER*, pages 574–583. INSTICC, SciTePress, 2011. doi:10.5220/0003391105740583.

26 Stefan Heule, Marc Nunkesser, and Alexander Hall. Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 683–692, New York, 2013. ACM. doi:10.1145/2452376.2452456.

27 Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963. doi:10.1007/978-1-4612-0865-5_26.

28 Piotr Indyk and David Woodruff. Optimal approximations of the frequency moments of data streams. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 202–208, New York, 2005. doi:10.1145/1060590.1060621.

29 Daniel M. Kane, Jelani Nelson, and David P. Woodruff. On the exact space complexity of sketching and streaming small norms. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 1161–1178, USA, 2010. Society for Industrial and Applied Mathematics. doi:10.1137/1.9781611973075.93.

30 Daniel M. Kane, Jelani Nelson, and David P. Woodruff. An optimal algorithm for the distinct elements problem. In *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '10, pages 41–52, New York, 2010. doi:10.1145/1807085.1807094.

31 Emin Karayel. A combinator library for prefix-free codes. *Archive of Formal Proofs*, April 2022. , Formal proof development. URL: https://isa-afp.org/entries/Prefix_Free_Code_Combinators.html.

32 Emin Karayel. Enumeration of equivalence relations. *Archive of Formal Proofs*, February 2022. , Formal proof development. URL: https://isa-afp.org/entries/Equivalence_Relation_Enumeration.html.

33 Emin Karayel. Formalization of randomized approximation algorithms for frequency moments. *Archive of Formal Proofs*, April 2022. , Formal proof development. URL: https://isa-afp.org/entries/Frequency_Moments.html.

34 Emin Karayel. Interpolation polynomials (in hol-algebra). *Archive of Formal Proofs*, January 2022. , Formal proof development. URL: `https://isa-afp.org/entries/Interpolation_Polynomials_HOL_Algebra.html`.

35 Emin Karayel. Median method. *Archive of Formal Proofs*, January 2022. , Formal proof development. URL: `https://isa-afp.org/entries/Median_Method.html`.

36 Emin Karayel. Universal hash families. *Archive of Formal Proofs*, February 2022. , Formal proof development. URL: `https://isa-afp.org/entries/Universal_Hash_Families.html`.

37 Robert Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21(10):840–842, 1978. `doi:10.1145/359619.359627`.

38 R.C. Mullin, I.M. Onyszchuk, S.A. Vanstone, and R.M. Wilson. Optimal normal bases in GF(p^n). *Discrete Applied Mathematics*, 22(2):149–161, 1988. `doi:10.1016/0166-218X(88)90090-X`.

39 Magnus O. Myreen. The CakeML Project's Quest for Ever Stronger Correctness Theorems. In *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:10, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ITP.2021.1`.

40 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-Verlag, Berlin, Heidelberg, 1 edition, 2002.

41 Vaughan R. Pratt. Shellsort and sorting networks. In *Outstanding Dissertations in the Computer Sciences*, 1972.

42 Pedro Reviriego, Alfonso Sanchez-Macian, Shanshan Liu, and Fabrizio Lombardi. On the security of the k minimum values (KMV) sketch. *IEEE Transactions on Dependable and Secure Computing*, 2021. `doi:10.1109/TDSC.2021.3101280`.

43 Joseph H. Silverman. Fast multiplication in finite fields GF(2^n). In *Cryptographic Hardware and Embedded Systems*, pages 122–134. Springer Berlin Heidelberg, 1999. `doi:10.1007/3-540-48059-5_12`.

44 Hagen Sparka, Florian Tschorsch, and Björn Scheuermann. P2KMV: A privacy-preserving counting sketch for efficient and accurate set intersection cardinality estimations. Cryptology ePrint Archive, Report 2018/234, 2018. `doi:10.14279/DEPOSITONCE-8374`.

45 Daniel Stüwe and Manuel Eberl. Probabilistic primality testing. *Archive of Formal Proofs*, February 2019. , Formal proof development. URL: `https://isa-afp.org/entries/Probabilistic_Prime_Tests.html`.

46 Joseph Tassarotti, Koundinya Vajjha, Anindya Banerjee, and Jean-Baptiste Tristan. A formal proof of PAC learnability for decision stumps. In *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, pages 5–17, 2021. `doi:10.1145/3437992.3439917`.

47 Mikkel Thorup and Yin Zhang. Tabulation based 5-universal hashing and linear probing. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, ALENEX '10, pages 62–76, USA, 2010. Society for Industrial and Applied Mathematics. `doi:10.1137/1.9781611972900.7`.

48 Salil P. Vadhan. Pseudorandomness. *Foundations and Trends® in Theoretical Computer Science*, 7(1-3):1–336, 2012. `doi:10.1561/0400000010`.

49 Mark N. Wegman and J. Lawrence Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22(3):265–279, 1981. `doi:10.1016/0022-0000(81)90033-7`.

50 Yang Yang, Ying Zhang, Wenjie Zhang, and Zengfeng Huang. GB-KMV: An augmented KMV sketch for approximate containment similarity search. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 458–469, 2019. `doi:10.1109/ICDE.2019.00048`.

Computational Back-And-Forth Arguments in Constructive Type Theory

Dominik Kirst
Universität des Saarlandes, Saarland Informatics Campus, Saarbrücken, Germany

Abstract

The back-and-forth method is a well-known technique to establish isomorphisms of countable structures. In this proof pearl, we formalise this method abstractly in the framework of constructive type theory, emphasising the computational interpretation of the constructed isomorphisms. As prominent instances, we then deduce Cantor's and Myhill's isomorphism theorems on dense linear orders and one-one interreducible sets, respectively. By exploiting the symmetry of the abstract argument, our approach yields a particularly compact mechanisation of the method itself as well as its two instantiations, all implemented using the Coq proof assistant. As adequate for a proof pearl, we attempt to make the text and mechanisation accessible for a general mathematical audience.

2012 ACM Subject Classification Theory of computation → Constructive mathematics; Theory of computation → Type theory; Theory of computation → Logic and verification

Keywords and phrases back-and-forth method, computable isomorphisms, Coq

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.22

Supplementary Material `https://www.ps.uni-saarland.de/extras/back-and-forth`

1 The Informal Argument

We begin by explaining the back-and-forth method informally to provide some background and intuition, using the example of Cantor's isomorphism theorem on countable dense linear orders. Actually established by Cantor himself via an alternative but related method [2], his result is often considered the origin of the back-and-forth argument and textbook presentations typically follow this particular strategy (Hausdorff's "Grundzüge der Mengenlehre" [8] may be the earliest example).[1] An important consequence of Cantor's isomorphism theorem is that the first-order theory of unbounded dense linear orders is complete and therefore decidable, which is the reason why the back-and-forth method was brought to fruition with a variety of similar results in model theory (see for instance [13], especially Chapter 6). While we do not include more of these model-theoretic constructions in this paper, as a second instance of an explicitly computational flavour we will consider Myhill's isomorphism theorem [12], stating that one-one interreducible sets are recursively isomorphic, at a later point.

So to get started on Cantor's isomorphism theorem, recall that (strict) linear orders may be characterised as irreflexive and transitive binary relations $(X, <)$ satisfying trichotomy $(x < y \vee x = y \vee y < x)$. Such a linear order is called dense if whenever $x < y$, then there is some z with $x < z < y$. These requirements alone do not yet fully determine the structure, as two orderings might differ regarding their endpoints. This leaves four possible cases depending on the existence of left and/or right endpoints, we focus on the unbounded version:

$$\forall x.\, \exists y y'.\, y < x < y'$$

[1] We refer the interested reader to [17] for further historic references as well as a comparison of Cantor's actual proof to the back-and-forth method.

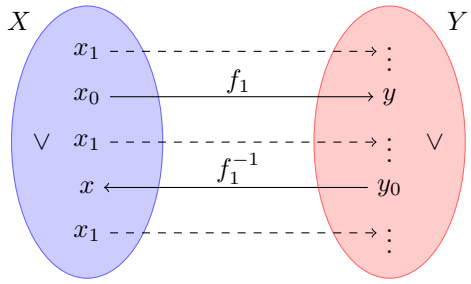


Figure 1 Illustration of the three cases how a finite stage of the order-isomorphism of $(X, <)$ and $(Y, <)$ can be extended. By construction, the stage f_1 already contains matchings for x_0 and y_0. For the construction of f_2 we want to find a matching for the next element x_1, which either lies in between the domain of f_1 or completely above or below it. In any case we find a matching partner using density, upwards unboundedness, or downwards unboundedness of Y.

Now given two unbounded dense linear orders $(X, <)$ and $(Y, <)$ that are countable, so assuming that $X = \{x_0, x_1, x_2, \dots\}$ and $Y = \{y_0, y_1, y_2, \dots\}$, we want to establish a bijection $F : X \to Y$ respecting the order, i.e. satisfying $x < x'$ iff $F(x) < F(x')$. This order-isomorphism can be constructed in finite stages, meaning that we begin with the empty partial isomorphism f_0 and iteratively extend to f_1, f_2, ... such that x_n is in the domain of f_{n+1}. To make sure that in this process no element of Y is missed, we simultaneously take care that y_n is in the range of f_{n+1}. Once the stages f_n are defined, we simply set $F(x_n) := f_{n+1}(x_n)$ and observe that the expected properties transport from the stages to F. Borrowing common set-theoretic notation, the construction can be summarised as

$$f_0 := \emptyset$$
$$f_{n+1} := \{(x_n, y), (x, y_n)\} \cup f_n$$
$$F := \bigcup_{n \in \mathbb{N}} f_n$$

where the matching partners y and x for x_n and y_n, respectively, remain to be determined.

To this end, we begin with a forwards step treating x_n. If $f_n(x_n)$ is already defined, then there is nothing to do. Otherwise, we need to find a suitable partner y not yet in the range of f_n which we can safely add as image of x_n while preserving the orderings. We distinguish three possible cases, which are also depicted in Figure 1:

- If there are x and x' in the domain of f_n such that $x < x_n < x'$, then (since f_n is finite) we can assume that x is the greatest such element while x' is the smallest. By the density of $(Y, <)$ there is y with $f_n(x) < y < f_n(x')$ which is then a suitable match for x.
- If the whole domain of f_n is below x_n, then using the unboundedness of $(Y, <)$ we choose y such that the whole range of f_n is below y. This makes y the desired match for x.
- In the remaining case where x_n is below the domain of f_n, we find a match y below the range of f_n again by unboundedness.

In any case we set $f'_{n+1}(x_n) := y$ as an intermediate extension of f_n and continue with the backwards step treating y_n, again only necessary if y_n is not yet in the range of f'_{n+1}. This step is completely symmetric to the forward step, that is, we find a suitable match x for y_n by distinguishing the same three cases regarding the position of y_n relative to the range of f'_{n+1}. Once a matching partner x is found, we extend f'_{n+1} by setting $f_{n+1}(x) := y_n$.

This concludes the construction of f_{n+1} and therefore of the order-isomorphism F in the case of unbounded orders. We just remark that the constructions work analogously if $(X, <)$ and $(Y, <)$ were to contain endpoints of the same characteristic.

In the next section, we continue by extracting the abstract idea of the above argument into a general isomorphism theorem, which is then instantiated to Cantor's result in Section 4. To illustrate the generality, in Section 5 we provide a second instantiation to Myhill's isomorphism theorem, crucially relying on the computational perspective outlined in Section 3.

All results of these technical sections are formalised in constructive type theory and mechanised in a single Coq file of about 500 lines of code.[2] While (even constructive) mechanisations of Cantor's isomorphism theorem were already given Giese and Schönegge [6] as well as Marzion [10], and while we closely follow Forster, Jahn, and Smolka [3] in the instantiation to Myhill's isomorphism theorem, we are not aware of an abstract formulation in constructive type theory, let alone a mechanisation with a proof assistant, of the back-and-forth method and its computational interpretation as such.

2 The Abstract Argument

Now switching to the formal framework of constructive type theory, we represent the two sets X and Y subject to the back-and-forth argument by types in the universe $\mathbb{T}$. Since the method only applies to countable structures, we could fix X and Y to be the type $\mathbb{N}$ of natural numbers. To explicitly include finite types, however, we just assume that X and Y are retracts of $\mathbb{N}$, so for instance X comes with an injection $i_X : X \hookrightarrow \mathbb{N}$ with explicitly given surjective left-inverse $r_X : \mathbb{N} \twoheadrightarrow X$, where we just write x_n for $r_X\, n$.[3] In order to investigate the argument abstractly, we need to axiomatise three ingredients: the particular structure of each instance, the corresponding notion of structure-preserving isomorphism, and a procedure to extend a partial isomorphism by one step.

The Abstract Assumptions

We first abstract over the required structure relating both sides of the potential isomorphism by an operation $\mathcal{A} : \mathbb{T} \times \mathbb{T} \to \mathbb{T}$. For Cantor's isomorphism theorem, an element $\mathcal{S}_{X,Y}$ of $\mathcal{A}(X, Y)$ will independently equip both X and Y with a dense unbounded linear order. Myhill's isomorphism theorem, however, imposes shared structure via mutual one-one reductions, making a binary operation necessary. To still enable the full usage of symmetry, we further assume a function associating to every structured pair $\mathcal{S}_{X,Y}$ its inverse $\mathcal{S}^{-1}_{X,Y}$ in $\mathcal{A}(Y, X)$ with

$$(\mathcal{S}^{-1}_{X,Y})^{-1} = \mathcal{S}_{X,Y}. \tag{1}$$

The next ingredient is the abstract isomorphism property we want to establish. It suffices to express this property locally, i.e. to relate pairs (x, x') in $X \times X$ and (y, y') in $Y \times Y$, written $(x, x') \sim (y, y')$, relative to $\mathcal{S}_{X,Y}$. The idea is to let $(x, x') \sim (y, y')$ express that x relates to x' in the structure over X exactly like y relates to y' in the structure over Y. In particular, we require that the structural similarity extends to equality, formally

$$(x, x') \sim (y, y') \;\to\; (x = x' \leftrightarrow y = y') \tag{2}$$

and that inverting the structure $\mathcal{S}_{X,Y}$ to $\mathcal{S}^{-1}_{X,Y}$ maintains the matching, formally

$$(x, x') \sim (y, y') \;\to\; (y, y') \sim (x, x') \tag{3}$$

[2] The Coq file can be obtained from the URL listed as supplementary material and an HTML version is accessible by simply clicking on any of the highlighted statements in this PDF.

[3] Note that for simplicity we exclude empty types as retracts of $\mathbb{N}$ by the requirement that r_X be total. For our purpose, we deem this choice reasonable as empty structures are usually trivially isomorphic.

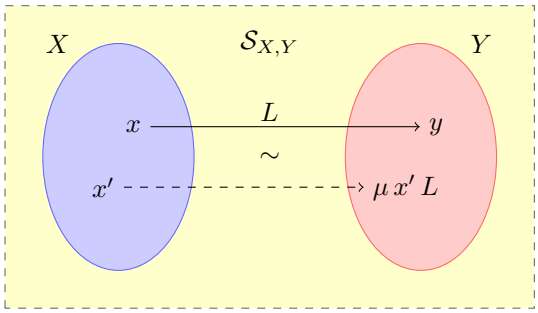


Figure 2 Illustration of the abstract assumptions. Two types X and Y are related by a surrounding structure $\mathcal{S}_{X,Y}$. If a partial isomorphism L connects an element $x : X$ with $y : Y$, then for x' the step function yields an element $\mu\, x'\, L$ relating to y like x' relates to x, yielding $(x, x') \sim (y, \mu\, x'\, L)$.

where the premise and conclusion are relative to $\mathcal{S}_{X,Y}$ and $\mathcal{S}^{-1}_{X,Y}$, respectively. The desired isomorphism can then be characterised as a function $F : X \to Y$ with inverse $F^{-1} : Y \to X$ such that $(x, x') \sim (F\, x, F\, x')$ for all x and x'.

Before we get to the final third ingredient, we need to model the notion of (finite) partial isomorphisms approximating F and F^{-1}. A simple choice is to take lists[4] $L : \mathcal{L}(X \times Y)$ of pairs $(x, y) : X \times Y$ and call them partial isomorphisms, written $\mathcal{S}_{X,Y}(L)$ to emphasise the dependency on $\mathcal{S}_{X,Y}$, if $(x, x') \sim (y, y')$ whenever $(x, y) \in L$ and $(x', y') \in L$. By flipping the components of each pair in L, we can turn L into $L^{-1} : \mathcal{L}(Y \times X)$ with the obvious property that $\mathcal{S}^{-1}_{X,Y}(L^{-1})$ whenever $\mathcal{S}_{X,Y}(L)$. We denote by $\mathsf{dom}(L)$ and $\mathsf{ran}(L)$ the domain and range of L, respectively, and write $L[x]$ for the (first) value of $x \in \mathsf{dom}(L)$.

So finally concluding the abstract ingredients, we assume a polymorphic step function

$$\mu \;:\; \forall XY.\, \mathcal{A}(X, Y) \to X \to \mathcal{L}(X \times Y) \to Y$$

finding a suitable partner y for x not yet included in L, meaning that if L is a partial isomorphism and $x \notin \mathsf{dom}(L)$, then $(x, \mu\, x\, L) :: L$ is a partial isomorphism:[5]

$$x \notin \mathsf{dom}(L) \;\to\; \mathcal{S}_{X,Y}(L) \;\to\; \mathcal{S}_{X,Y}((x, \mu\, x\, L) :: L) \tag{4}$$

To sum up, we illustrate the abstract setup graphically in Figure 2 and provide an overview of all assumptions as stated in Coq in Listing 1.

The Abstract Construction

As in the informal argument, the idea is to start with the empty partial isomorphism and then to extend in step $n + 1$ by matching up both x_n and y_n with suitable partners (if not yet accommodated), thereby ensuring that both structures are exhausted. So we first extend μ to a polymorphic function μ' adding a new match for x to L if no existing match is found:

$$\mu' \;:\; \forall XY.\, \mathcal{A}(X, Y) \to X \to \mathcal{L}(X \times Y) \to \mathcal{L}(X \times Y)$$

$$\mu'\, x\, L \;:=\; \begin{cases} (x, \mu\, x\, L) :: L & \text{if } x \notin \mathsf{dom}(L) \\ (x, L[x]) :: L & \text{if } x \in \mathsf{dom}(L) \end{cases}$$

[4] Constructed from the empty list [] and the cons operation $x :: L$ for $x : X$ and $L : \mathcal{L}(X)$.
[5] We treat the first three arguments of μ as implicit and will do so for all similar functions.

Listing 1 Abstract assumptions regarding the structure (`structure`), the corresponding notion of isomorphism (`iso`), and the extension procedure (`find`) as formulated in Coq. Note that the former two take their arguments in curried form as opposed to the cartesian representation on paper.

```
structure : Type -> Type -> Type
srev : forall X Y, structure X Y -> structure Y X
srev_invol : forall X Y (S : structure X Y), srev (srev S) = S

iso : forall X Y, structure X Y -> X -> X -> Y -> Y -> Prop
iso_eq : forall X Y (S : structure X Y) x x' y y', iso S x x' y y' -> x = x' <-> y = y'
iso_rev : forall X Y (S : structure X Y) x x' y y', iso S x x' y y' -> iso (srev S) y y' x x'

find : forall X, structure X Y -> X -> list (X * Y) -> Y
find_iso : forall X Y (S : structure X Y) L x, x ∉ dom L -> tiso S L -> tiso S ((x, find S x L) :: L)
```

Note that in the second case, where a match for x was already present in L, we could as well just return L, but the choice to repeat the match slightly simplifies the (in any case straightforward) argument for the desired property $x \in \mathsf{dom}(\mu'\, x\, L)$.

Now since μ' is polymorphic and therefore usable in both directions, the iterative process at the heart of the back-and-forth argument can be defined exploiting the structural symmetry:

$$L\_ \;:\; \forall XY.\, \mathcal{A}(X,Y) \to \mathbb{N} \to \mathcal{L}(X \times Y)$$

$$\begin{aligned} L_0 &:= [] \\ L_{n+1} &:= (\mu'\, y_n\, (\mu'\, x_n\, L_n)^{-1})^{-1} \end{aligned}$$

So in the second case, we first extend L_n with a match for x_n by the inner application of μ' (regarding $\mathcal{S}_{X,Y}$) and then flip the result to obtain a partial isomorphism in the opposite direction. This is then extended with a match for y_n by the outer application of μ' (regarding $\mathcal{S}^{-1}_{X,Y}$), after which the result is flipped again to yield a partial isomorphism in the original direction.

Since we have $x_n \in \mathsf{dom}(L_{n+1})$ and $y_n \in \mathsf{dom}(L_{n+1})^{-1} = \mathsf{ran}(L_{n+1})$ by the above remark that $x \in \mathsf{dom}(\mu'\, x\, L)$ for all x, we can define the isomorphism $F : X \to Y$ and its inverse $F^{-1} : Y \to X$ on a given structured pair $\mathcal{S}_{X,Y}$ simply by

$$F\, x_n := L_{n+1}[x_n] \qquad \text{and} \qquad F^{-1}\, x_n := L^{-1}_{n+1}[y_n]$$

where we implicitly use that for every $x : X$ there is $n : \mathbb{N}$ with $x = x_n$, analogously for $y : Y$.

The Abstract Verification

To conclude the abstract back-and-forth argument, we show that the function $F : X \to Y$ is an isomorphism for $\mathcal{S}_{X,Y}$. We first formally state that flipping and extending with μ' preserves partial isomorphisms.

▶ **Lemma 1.** *If* $\mathcal{S}_{X,Y}(L)$ *then* $\mathcal{S}^{-1}_{X,Y}(L^{-1})$ *and* $\mathcal{S}_{X,Y}(\mu'\, x\, L)$.

Proof. Straightforward by Assumptions (3) and (4), respectively. ◀

Then in particular every L_n is a partial isomorphism.

▶ **Lemma 2.** $\mathcal{S}_{X,Y}(L_n)$

Proof. By induction on n, the case of $L_0 = []$ is trivial. For L_{n+1} we assume $\mathcal{S}_{X,Y}(L_n)$ as inductive hypothesis and apply both parts of Lemma 1 in both directions to obtain $(\mathcal{S}^{-1}_{X,Y})^{-1}(L_{n+1})$, which is the same as $\mathcal{S}_{X,Y}(L)$ by Assumption (1). ◀

We can now prove the abstract isomorphism theorem with a simple argument.

▶ **Theorem 3** (Isomorphism). *F satisfies $(x, x') \sim (F\,x, F\,x')$ and is inverted by F^{-1}.*

Proof. For the first claim, let $x = x_n$, $x' = x_m$, and w.l.o.g $n \leq m$. Since then $L_{n+1} \subseteq L_{m+1}$, we have $(x, F\,x) = (x_n, L_{n+1}[x]) \in L_{m+1}$ and $(x', F\,x') = (x_m, L_{m+1}[x]) \in L_{m+1}$, from which conclude $(x, x') \sim (F\,x, F\,x')$ since $\mathcal{S}_{X,Y}(L_{m+1})$ by Lemma 2.

For the second claim, note that for some n big enough, we have both $(x, F\,x) \in L_n$ (since x is eventually handled by some forwards step) and $(F^{-1}(F\,x), F\,x) \in L_n$ (since $F\,x$ is eventually handled by some backwards step). By Lemma 2 we therefore obtain $(x, F^{-1}(F\,x)) \sim (F\,x, F\,x)$, from which $x = F^{-1}(F\,x)$ follows by Assumption (2). Analogously, we establish $y = F\,(F^{-1}\,y)$ and thus conclude the claim that F^{-1} inverts F. ◀

3 The Computational Argument

Before we consider instantiations of our abstract back-and-forth argument, we take a break to compare two quite different readings of the isomorphism theorem which are available in our constructive setting. Conventionally, Theorem 3 could be summarised as:

Countable structures with a structure-preserving extension function are isomorphic.

Here the view is that both the input extension function and the output isomorphism are ordinary (set-theoretic) functions. Moreover, the embedding of the structures into the natural numbers is a mere limitation to countable cardinality.

The alternative view is that we consider all involved functions to be computable, relying on the fact that all functions definable in constructive type theory are such.[6] This allows for a synthetic approach to computability [16, 1, 4] disposing of any reference to a formal model of computation like Turing machines, especially easing the mechanisation of otherwise quite laborious arguments regarding undecidability [5] and incompleteness [9]. Also in situations like the back-and-forth method with its set-theoretic reading covering up the inherent computational content, this approach admits an elegant combination of both perspectives.

So adopting the computational view, the input extension function provides a matching algorithm μ which we use to implement μ', L_n, and ultimately F. Since we are able to implement these functions without classical assumptions,[7] their computability is guaranteed. The only price we have to pay is that the classically trivial case distinction on $x \in \mathsf{dom}(L)$ in the definition of μ' needs to be constructively justified. Fortunately, implementing the necessary decision procedure for list membership on a type with decidable equality is a simple exercise, especially if we instead were to hastily use classical assumptions at the cost of having to program Turing machines for computational results. Finally observing that a computable embedding into natural numbers replaces countability with effective enumerability and discreteness (i.e. decidable equality), we may reinterpret Theorem 3:

Computational interpretation: ***enumerable*** *and* ***discrete*** *structures with a structure-preserving extension* ***algorithm*** *are* ***computably*** *isomorphic.*

---

[6] An intuitive justification for this fact is that constructive type theory is nothing but a dependently typed programming language. A more formal justification applicable for many constructive systems is to exhibit a realisability model with the desired properties.

[7] So for instance without appeal to the excluded middle ($\forall P.\, P \vee \neg P$), or even stronger, the axiom of choice. Both principles can be consistently added to constructive type theory but, depending on the formulation, already the former would allow the definition of uncomputable functions.

This means that in each of the upcoming instantiations we actually obtain two results: a classical and a computational isomorphism. Moreover, both readings are meaningful in their own right and it is one of the nice features of constructive mathematics and synthetic computability that we obtain them both simultaneously after doing the work just once.[8]

4 Cantor's Isomorphism Theorem

We now start reaping what we sowed in Section 2 and establish Cantor's result with our general isomorphism theorem. To admit the computational view, we represent unbounded dense linear orders $(X, <)$ with explicit functions d, l, and g for density and unboundedness:

$$x < d\,x\,y < y \qquad\qquad l\,x < x < g\,x$$

Moreover, we assume a characteristic function $f_< : X \to X \to \mathbb{B}$ using the inductive type $\mathbb{B}$ of boolean values such that $f_<\,x\,y = \mathsf{true}$ iff $x < y$.[9] In the computational view, this means that the witnesses for density and unboundedness can be computed and that the relation is decidable, while in the classical view using these functions is a mere symbolic convenience.

We now fix two countable unbounded dense linear orders $(X, <)$ and $(Y, <)$ and bring the informal idea outlined in Section 1 into the shape matching the abstract argument in Section 2. Given two pairs $(x, x') : X \times X$ and $(y, y') : Y \times Y$, we define

$$(x, x') \sim_C (y, y') \;:=\; (x = x' \leftrightarrow y = y') \wedge (x < x' \leftrightarrow y < y')$$

and call $L : \mathcal{L}(X \times Y)$ a partial order-isomorphism if $(x, x') \sim_C (y, y')$ for all $(x, y), (x', y') \in L$. By virtue of the abstract isomorphism theorem, we just need to show how a partial order-isomorphism can be extended by one step in one direction.

▶ **Definition 4.** *We define a function $\mu_C : X \to \mathcal{L}(X \times Y) \to Y$ exactly following the three cases distinguished in Section 1. So given $x \notin \mathsf{dom}(L)$, first use $f_<$ to determine which of the three situations relating x to $\mathsf{dom}(L)$ is accurate. In the case where x is in between elements of $\mathsf{dom}(L)$, use d to find a match in Y. In the two other cases where x is below or above all elements of $\mathsf{dom}(L)$, use l and g, respectively. If $x \in \mathsf{dom}(L)$, we just return a dummy value.*

▶ **Lemma 5.** *If $x \notin \mathsf{dom}(L)$ and L is a partial order-isomorphism, then so is $((x, \mu_C\,x\,L) :: L)$.*

Proof. The only non-trivial obligation is to show $(x, x') \sim_C (\mu_C\,x\,L, y')$ for $(x', y') \in L$:

- That $x = x'$ iff $\mu_C\,x\,L = y'$ is straightforward since both are actually false. The former is ruled out by $x \notin \mathsf{dom}(L)$ and the latter since in that case $\mu_C\,x\,L \notin \mathsf{ran}(L)$.
- That $x < x'$ iff $\mu_C\,x\,L < y'$ follows by analysing the cases underlying the definition of μ_C. We only consider the more complicated case where x lies in between elements of $\mathsf{dom}(L)$, the other two cases are similar and easier. In this situation, there are x_{max} and x_{min} in $\mathsf{dom}(L)$ with $x_{\mathsf{max}} < x < x_{\mathsf{min}}$ with no domain element in between. Since L is already a partial order-isomorphism and since then $\mu_C\,x\,L = d\,L[x_{\mathsf{max}}]\,L[x_{\mathsf{min}}]$, we have $L[x_{\mathsf{max}}] < \mu_C\,x\,L < L[x_{\mathsf{min}}]$ in Y with no range element in between. We then deduce:

$$x < x' \;\Leftrightarrow\; x_{\mathsf{min}} \le x' \;\Leftrightarrow\; L[x_{\mathsf{min}}] \le L[x'] \;\Leftrightarrow\; \mu_C\,x\,L < y' \qquad \blacktriangleleft$$

Now Cantor's isomorphism theorem in its classical formulation can be derived directly.

[8] As already done in Section 2 we continue to try and use neutral formulations avoiding a bias into one of the readings, although wordings like "we construct" or "we compute" impose themselves very often.

[9] Note that in Coq we prefer to formulate these assumptions with informative types, for instance d is expressed by $\forall xy.\,\Sigma z.\,x < z < y$ and $f_<$ could be expressed as $\forall xy.\,(x < y) + (x \not< y)$.

Listing 2 Coq proof script for Cantor's isomorphism theorem. After using the general isomorphism theorem `back_and_forth`, the first eight goals are arranged like the assumptions listed in Listing 1. The only non-trivial ones are the latter two where `partner` is μ_C and `step_morph` is Lemma 5. The next five goals instantiate to the given orders and the last goal is the actual derivation of the claim.

```
Theorem Cantor X Y (OX : dulo X) (OY : dulo Y) (RX : retract X nat) (RY : retract Y nat) :
{ F : X -> Y & { G | inverse F G /\ forall x x', x < x' <-> (F x) < (F x') } }.
Proof.
  unshelve edestruct back_and_forth as [F[G[H1 H2]]].
  - intros A B. exact (dulo A * dulo B).                                    (* structure *)
  - intros A B [OA OB]. exact (OB, OA).                                     (* srev *)
  - cbn. intros A B [OA OB]. reflexivity.                                   (* srev_invol *)
  - intros A B [OA OB] a a' b b'. exact ((a = a' <-> b = b') /\ (a < a' <-> b < b')).  (* iso *)
  - cbn. tauto.                                                             (* iso_eq *)
  - cbn. tauto.                                                             (* iso_rev *)
  - cbn. intros A B [OA OB] f a. exact (partner OA OB f a).                 (* find *)
  - cbn. intros A B [OA OB] f x. unfold step, tiso. now apply step_morph.   (* find_iso *)

  - exact X.
  - exact Y.
  - exact (OX, OY).
  - exact RX.
  - exact RY.

  - cbn in *. exists F, G. split; try apply H1. apply H2.
Qed.
```

▶ **Theorem 6** (Cantor). *All countable unbounded dense linear orders are isomorphic.*

Proof. We instantiate Theorem 3 as follows, see also Listing 2:

- As structure $\mathcal{A}(X, Y)$ we require that X and Y are unbounded dense linear orders.
- The inversion operation turning elements of $\mathcal{A}(X, Y)$ to $\mathcal{A}(Y, X)$ is obvious.
- As isomorphism property we choose $(x, x') \sim_C (y, y')$.
- As polymorphic one-step extension function we choose μ_C.
- Assumptions (1)–(3) are trivial.
- Assumption (4) was proven in Lemma 5.

This yields a function $F : X \to Y$ with inverse F^{-1} such that $(x, x') \sim_C (F\,x, F\,x')$. The latter contains $x < x' \leftrightarrow F\,x < F\,x'$, so F is an order-isomorphism. ◀

The corresponding computational result we obtain for free then reads:

▶ **Theorem 7** (Computational Cantor). *All decidable linear orders over enumerable domain with computable witnesses for density and unboundedness are computably isomorphic.*

5 Myhill's Isomorphism Theorem

As a second instance of our general isomorphism theorem, we tackle Myhill's result that one-one interreducible sets of natural numbers are recursively isomorphic.[10] To first provide some intuition, recall that a set $X \subseteq \mathbb{N}$ many-one reduces to a set $Y \subseteq \mathbb{N}$ if there is a computable function $f : \mathbb{N} \to \mathbb{N}$ such that $f(x) \in Y$ implies $x \in X$ and $f(x) \notin Y$ implies $x \notin Y$. As a typical usage of such a reduction f, a decision procedure for $x \in X$ can be obtained by a decider for Y by first computing $f(x)$ and then determining whether $f(x) \in Y$. If f happens to be injective (bijective), it is called one-one reduction (recursive isomorphism).

[10] We simply adopt the Coq mechanisation given by Forster et al. [3] to our abstract setting and explicitly link up the corresponding definitions and theorems in the forthcoming text.

Given that interreducible sets and recursive isomorphisms are algorithmic notions, we now fully rely on the synthetic approach to computability as explained in Section 3 and take simple type-theoretic definitions representations of these notions:

- Sets of natural numbers are more generally represented as unary predicates $p : X \to \mathbb{P}$ on enumerable and discrete types X (that is, retracts of $\mathbb{N}$).
- A many-one reduction from $p : X \to \mathbb{P}$ to $q : Y \to \mathbb{P}$ is represented as a function $f : X \to Y$ such that $p\,x$ iff $q\,(f\,x)$, which we denote by $f : p \preceq q$.
- A one-one reduction (recursive isomorphism) is an injective (bijective) many-one reduction.

Recall that if we were to use classical logic, we would spoil the computational interpretation and instead would have to explicitly require reductions to be computable in a formal model like Turing machines. This would turn at least the mechanisation level into a mess, especially since then the recursive isomorphism obtained by the back-and-forth method must be shown computable, too.

That said, we now fix two predicates $p : X \to \mathbb{P}$ and $q : Y \to \mathbb{P}$ on retracts X and Y of $\mathbb{N}$, together with one-one reductions $f : p \preceq q$ and $g : q \preceq p$. A recursive isomorphism $F : X \to Y$ will satisfy both $F : p \preceq q$ and $F^{-1} : q \preceq p$, which yields a strict correspondence of both predicates. Expressing this correspondence locally for pairs $(x, x') : X \times X$ and $(y, y') : Y \times Y$, we therefore want to establish that $p\,x$ iff $q\,y$ (by symmetry the same follows for x' and y'), along with the usual requirement that the correspondence be one-to-one:

$$(x, x') \sim_M (y, y') \;:=\; (x = x' \leftrightarrow y = y') \wedge (p\,x \leftrightarrow q\,y)$$

We call $L : \mathcal{L}(X \times Y)$ a partial (recursive) isomorphism if $(x, x') \sim_M (y, y')$ for all $(x, y), (x', y') \in L$. Again by virtue of the abstract isomorphism theorem, we just need to show how a partial isomorphism can be extended by one step in one direction.

▶ **Definition 8** (Lemma 55 in [3]). *We first define a function $\mu_M^X : X \to \mathcal{L}(X \times Y) \to X$ such that given x and L it is safe to add $f\,(\mu_M^X\,x\,L)$ as value for x to L. So determine whether $f\,x \in \mathsf{ran}(L)$, if not we can just return x. If otherwise $(x', f\,x) \in L$ return $\mu_M^X\,x'\,L'$, where L' is L with $(x', f\,x)$ removed, so we keep searching until we find some element x_0 with $f\,x_0 \notin \mathsf{ran}(L)$ while the search along f ensures that $p\,x$ iff $p\,x_0$.*[11]

We then obtain a function $\mu_M : X \to \mathcal{L}(X \times Y) \to Y$ by $\mu_M\,x\,L := f\,(\mu_M^X\,x\,L)$.

▶ **Lemma 9**. *If $x \notin \mathsf{dom}(L)$ and L is a partial isomorphism, then so is $((x, \mu_M\,x\,L) :: L)$.*

Proof. We first verify that $\mu_M\,x = f\,(\mu_M^X\,x\,L)$ indeed is a suitable match for x:

$$(p\,x \leftrightarrow p\,(\mu_M^X\,x\,L)) \;\wedge\; \mu_M^X\,x\,L \in x :: \mathsf{dom}(L) \;\wedge\; f\,(\mu_M^X\,x\,L) \notin \mathsf{ran}(L) \qquad (*)$$

We apply induction following the execution trace of μ_M^X. In the terminating case where $f\,x \notin \mathsf{ran}(L)$, we have $\mu_M^X\,x\,L = x$ and hence the claim is trivial. In the recursive case we have $\mu_M^X\,x\,L = \mu_M^X\,x'\,L'$, where L' is L without $(x', f\,x)$, and assume the claim for x' and L'.

- We deduce $p\,x \leftrightarrow p\,(\mu_M^X\,x'\,L')$ using $p\,x \leftrightarrow q\,(f\,x)$, since f is a reduction, $q\,(f\,x) \leftrightarrow p\,x'$ since L is a partial isomorphism, and $p\,x' \leftrightarrow p\,(\mu_M^X\,x'\,L'))$ by the inductive hypothesis.
- We already have $\mu_M^X\,x'\,L' \in \mathsf{dom}(L)$ since by induction $\mu_M^X\,x'\,L' \in x' :: \mathsf{dom}(L')$ and we have $x' :: \mathsf{dom}(L') = \mathsf{dom}(L)$ by construction of L'.

[11] Since this algorithm is not structurally recursive but descends on the length of L, we use the Equations package [18] to implement it in Coq.

Listing 3 Coq proof script for Myhill's isomorphism theorem. After using the general isomorphism theorem `back_and_forth`, the first eight goals are arranged like the assumptions listed in Listing 1. The only non-trivial ones are the latter two where `mstep` is μ_M and `step_corr` is Lemma 9. The next five goals instantiate to the given orders and the last goal is the actual derivation of the claim.

```
Theorem Myhill X Y (SXY : bireduction X Y) (RX : retract X nat) (RY : retract Y nat) :
{ F : X -> Y & { G | inverse F G /\ reduction p q F } }.
Proof.
  unshelve edestruct back_and_forth as [F[G[H1 H2]]].
  - intros A B. exact (bireduction A B).                                    (* structure *)
  - intros A B S. cbn in *. apply (@Build_bireduction B A eY eX q p g f); apply S.   (* srev *)
  - cbn. intros A B []. reflexivity.                                        (* srev_invol *)
  - intros A B S a a' b b'. cbn in S. exact ((a = a' <-> b = b') /\ (p a <-> q b)).  (* iso *)
  - cbn. tauto.                                                             (* iso_eq *)
  - cbn. tauto.                                                             (* iso_rev *)
  - cbn. intros A B S C a. exact (mstep f eX eY C a).                       (* find *)
  - cbn. intros A B S C a. unfold step, tiso. apply step_corr; apply S.     (* find_iso *)

  - exact X.
  - exact Y.
  - exact SXY.
  - exact RX.
  - exact RY.

  - cbn in *. exists F, G. split; try apply H1. intros x. now apply H2.
Qed.
```

- We assume $f\,(\mu_M^X\,x'\,L') \in \mathsf{ran}(L)$, so given that $\mathsf{ran}(L) = f\,x :: \mathsf{ran}(L')$ we have either $f\,(\mu_M^X\,x'\,L') \in \mathsf{ran}(L')$ or $f\,(\mu_M^X\,x'\,L') = f\,x$. Since the former contradicts the inductive hypothesis the latter must be the case, so $\mu_M^X\,x'\,L' = x$ by the injectivity of f. Since the second part of the inductive hypothesis then yields $x \in x' :: \mathsf{dom}(L')$, we obtain a contradiction to the assumption $x \notin \mathsf{dom}(L)$.

Now returning to the actual goal, we show $(x, x') \sim_M (\mu_M\,x\,L, y')$ for $(x', y') \in L$.

- That $x = x'$ iff $\mu_M\,x\,L = y'$ is trivial using $x \notin \mathsf{dom}(L)$ and $\mu_M\,x\,L \notin \mathsf{ran}(L)$, the former being an assumption and the latter part three of $(*)$.
- That $p\,x$ iff $q\,(\mu_M\,x\,L)$ is immediate using that $q\,(\mu_M\,x\,L)$ iff $p\,(\mu_M^X\,x\,L)$ since f is a reduction, and $p\,(\mu_M^X\,x\,L)$ iff $p\,x$ by part one of $(*)$. ◀

Now Myhill's isomorphism theorem follows for all enumerable and discrete types.

▶ **Theorem 10** (Myhill, Theorem 58 in [3]). *One-one interreducible unary predicates on enumerable and discrete types are recursively isomorphic.*

Proof. We instantiate Theorem 3 as follows, see also Listing 3:

- As structure $\mathcal{A}(X, Y)$ we require a pair of one-one interreducible predicates.
- The inversion operation turning elements of $\mathcal{A}(X, Y)$ to $\mathcal{A}(Y, X)$ is obvious.
- As isomorphism property we choose $(x, x') \sim_M (y, y')$.
- As polymorphic one-step extension function we choose μ_M.
- Assumptions (1)–(3) are trivial.
- Assumption (4) was proven in Lemma 9.

This yields a function $F : X \to Y$ with inverse F^{-1} such that $(x, x') \sim_M (F\,x, F\,x')$. The latter contains $p\,x \leftrightarrow q\,(F\,x)$, so F is a recursive isomorphism. ◀

To conclude, observe that if we forget about the computational view, we have just proven something quite different: the Cantor-Bernstein theorem restricted to countable sets. This restriction happens to be provable constructively (exactly due to the connection to Myhill's isomorphism theorem), while the general result is equivalent to the excluded middle [14].

▶ **Theorem 11** (Countable Cantor-Bernstein, Theorem 59 in [3]). *Countable sets X and Y with given injections $f : X \to Y$ and $g : Y \to X$ also admit a bijection $h : X \to Y$.*

Proof. If we choose as p and q the empty predicates on X and Y, respectively, then f trivially satisfies the defining property of a reduction from p to q and analogously so does g for q and p. Thus Theorem 10 yields a bijection (with a trivial additional property regarding p and q that we just ignore). ◀

6 Conclusion

In this proof pearl, we have isolated an abstract and computational approach to the back-and-forth method and considered two prominent instantiations. As the aim was a condensed and pointed presentation, we refrained from providing more instantiations in other domains for now. Nevertheless, this brief concluding section is meant to sketch a few more applications that our approach might be usable for.

First of all, for Cantor's isomorphism theorem we only discuss the characteristic of dense linear orders without endpoints, as it is the most familiar case due to its applicability to $\mathbb{Q}$. However, the three other cases of one (left or right) or two endpoints are completely analogous and can be obtained by simple modification of our proof. Of course one could also abstract over the exact characteristic, yielding all four versions simultaneously.

Secondly, regarding other domains, we already mentioned the crucial use of back-and-forth arguments in model theory, as described in Chapter 7 of [13]. Two further examples are the isomorphism of countably infinite Boolean algebras [7] as well as the uniqueness of the Rado graph as the result of a particular method to generate random graphs [15]. We believe that all these examples can be described in our abstract framework and so for a succinct Coq mechanisation one would just need to represent the corresponding structure and verify a suitable step function.

Thirdly, not immediately concerning further instantiations, the computational content of our axiom-free mechanisation could be made even more explicit by using Coq's extraction mechanism. With this feature one would obtain the algorithms implemented in the constructive proofs as executable programs in external programming languages. These algorithms could then be analysed regarding their efficiency and compared with respect to the necessary computational primitives. For instance, while the back-and-forth construction itself just follows the enumeration of the domains, the step function in the case of Cantor's isomorphism theorem may exhibit quite different complexity, depending on whether the witnesses for density and unboundedness are provided as efficient functions or need to be computed with linear search. For this example and a more general analysis of the back-and-forth method regarding different notions of computation, we refer the reader to [11].

References

1 Andrej Bauer. First steps in synthetic computability theory. *Electronic Notes in Theoretical Computer Science*, 155:5–31, 2006.

2 Georg Cantor. Beiträge zur Begründung der transfiniten Mengenlehre. *Mathematische Annalen*, 46(4):481–512, 1895.

3 Yannick Forster, Felix Jahn, and Gert Smolka. A Constructive and Synthetic Theory of Reducibility: Myhill's Isomorphism Theorem and Post's Problem for Many-one and Truth-table Reducibility in Coq (Full Version). working paper or preprint, February 2022. URL: `https://hal.inria.fr/hal-03580081`.

4 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, 2019.

5 Yannick Forster, Dominique Larchey-Wendling, Andrej Dudenhefner, Edith Heiter, Dominik Kirst, Fabian Kunze, Gert Smolka, Simon Spies, Dominik Wehr, and Maximilian Wuttke. A Coq library of undecidable problems. In *CoqPL 2020 The Sixth International Workshop on Coq for Programming Languages*, 2020.

6 Martin Giese and Arno Schönegge. *Any Two Countable, Densely Ordered Sets Without Endpoints are Isomorphic: A Formal Proof with KIV*. Univ., Fak. für Informatik, 1995.

7 Paul Halmos and Steven Givant. *Introduction to Boolean algebras*. Springer, 2009.

8 Felix Hausdorff. Grundzüge der Mengenlehre, 1914.

9 Dominik Kirst and Marc Hermes. Synthetic undecidability and incompleteness of first-order axiom systems in Coq. In *12th International Conference on Interactive Theorem Proving (ITP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

10 Evan Marzion. Visualizing Cantor's theorem on dense linear orders using Coq. Blog post. URL: `https://emarzion.github.io/Cantor-Thm/`.

11 Alexander G Melnikov and Keng Meng Ng. The back-and-forth method and computability without delay. *Israel Journal of Mathematics*, 234(2):959–1000, 2019.

12 John Myhill. Creative sets. *Journal of Symbolic Logic*, 22(1), 1957.

13 Bruno Poizat. *A course in model theory: an introduction to contemporary mathematical logic*. Springer Science & Business Media, 2012.

14 Pierre Pradic and Chad E. Brown. Cantor-Bernstein implies excluded middle. *arXiv preprint arXiv:1904.09193*, 2019.

15 Richard Rado. Universal graphs and universal functions. *Acta Arithmetica*, 9(4):331–340, 1964.

16 Fred Richman. Church's thesis without tears. *The Journal of symbolic logic*, 48(3):797–803, 1983.

17 Charles L. Silver. Who invented Cantor's back-and-forth argument? *Modern Logic*, 4(1):74–78, 1994.

18 Matthieu Sozeau and Cyprien Mangin. Equations reloaded: High-level dependently-typed functional programming and proving in Coq. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29, 2019.

Formalizing the Divergence Theorem and the Cauchy Integral Formula in Lean

Yury Kudryashov
University of Toronto at Mississauga, Canada (till July 2022)
Texas A&M University, College Station, TX, USA (since August 2022)

Abstract

I formalize a version of the divergence theorem for a function on a rectangular box that does not assume regularity of individual partial derivatives, only Fréchet differentiability of the vector field and integrability of its divergence. Then I use this theorem to prove the Cauchy-Goursat theorem (for some simple domains) and bootstrap complex analysis in the Lean mathematical library. The main tool is the GP-integral, a version of the Henstock-Kurzweil integral introduced by J. Mawhin in 1981. The divergence theorem for this integral does not require integrability of the divergence.

2012 ACM Subject Classification Security and privacy → Logic and verification; Mathematics of computing → Integral calculus

Keywords and phrases divergence theorem, Green's theorem, Gauge integral, Cauchy integral formula, Cauchy-Goursat theorem, complex analysis

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.23

Supplementary Material *InteractiveResource (Documentation Website)*: `http://div-thm.urkud.name/`

Acknowledgements I want to thank Patrick Massot for bringing up an idea of formalizing a sufficiently general version of the divergence theorem, Sébastien Gouëzel for fruitful discussions and peer review of most of the code, and my wife Nataliya Goncharuk for constant support. I also want to thank Anne Baanen, Johan Commelin, Nataliya Goncharuk, and Robert Y. Lewis for valuable comments on the draft versions of this paper and I thank my son Konstantin for finding lots of typos. I am also grateful to the anonymous referees for their valuable comments.

1 Introduction

The divergence theorem says that, under certain assumptions, the integral of the divergence of a vector field over a region is equal to the flow of this vector field through the boundary of the region. For a rectangular region on the plane, this can be written as

$$\int_a^b \int_c^d \left(\frac{\partial f(x,y)}{\partial x} + \frac{\partial g(x,y)}{\partial y} \right) dy dx = \int_a^b (g(x,d) - g(x,c))\, dx + \int_c^d (f(b,y) - f(a,y))\, dy, \quad (1)$$

where $f, g \colon \mathbb{R}^2 \to E$ are functions from the plane to some Banach space. This statement is also known as Green's theorem. For continuously differentiable functions f, g, the equality immediately follows from the Fundamental Theorem of Calculus and the Fubini Theorem.

If $F \colon \mathbb{C} \to E$ is a complex differentiable function, then, due to Cauchy-Riemann relations, the left-hand side of Green's theorem applied to $f(x,y) = F(x+iy)$ and $g(x,y) = iF(x+iy)$ is zero, and we get Cauchy's integral theorem (a.k.a. the Cauchy-Goursat theorem) for a rectangle.

There is a gap in the proof outlined above: I explained how to prove Green's formula for a *continuously differentiable* function but the Cauchy-Goursat theorem works for any *complex differentiable* function. A common misbelief is that this makes Green's formula (and the divergence theorem) unusable for the Cauchy-Goursat theorem, and one has to prove it using an explicit infinite descent.

The goal of this project is to formulate a version of the divergence theorem that implies the Cauchy-Goursat theorem without any assumptions on the derivative of a complex differentiable function. It was originally inspired by a version of the divergence theorem by F. Acker [2] that works for a vector field with *continuous divergence*, though the actual proof I formalized is very different.

Here is the list of most important definitions and theorems I formalize in this project.

- Riemann, McShane, Henstock-Kurzweil, and GP-integrals of a function from a box in $\mathbb{R}^n$ to a real Banach space, see Sections 3 and 4;
- the divergence theorem for the GP-integral, see Theorem 2 and Section 4.9;
- the divergence theorem for the Bochner integral, see Sections 3.3 and 4.11; the Bochner integral is a generalization of the Lebesgue integral to functions that take values in Banach spaces;
- some basic theorem from complex analysis, see Sec. 5:
  - the Cauchy-Goursat theorem for rectangles, annuli, and disks;
  - the Cauchy integral formula for a disk;
  - analyticity of a complex differentiable function;
  - the Riemann removable singularity theorem;
  - maximum modulus principle;
  - Liouville's theorem;
  - Schwarz lemma.

Most of these theorems were formalized earlier in other theorem provers. However, this is the first project where the divergence theorem was formalized in this generality. In particular, the Cauchy-Goursat theorem becomes a simple corollary of the general divergence theorem. Also, the assumptions in the Cauchy-Goursat theorem are slightly weaker than in most textbooks.

The most similar other project is Abdulaziz and Paulson's formalization of Green's theorem for a large class of domains in $\mathbb{R}^2$ in Isabelle [1]. Neither Isabelle nor my version of the divergence theorem implies the other. Here are the key points where one of the formalizations is more general than the other, see also Sec. 2.

shape of the domain The Isabelle formalization works with any elementary region while my formalization only works for a rectangular box;

differentiability The Isabelle formalization only requires existence of the partial derivatives while I require Fréchet differentiability of the original function;

regularity of the derivative The Isabelle formalization requires integrability of each partial derivative while my formalization makes no regularity assumptions for the GP-integral and assumes only integrability of the divergence for the Henstock-Kurzweil and Bochner integrals; this makes it possible deduce the Cauchy-Goursat theorem from my version of the divergence theorem;

domain dimension The Isabelle formalization only deals with dimension two while I formalize the divergence theorem in any finite dimension; this generalization is needed throughout differential topology, PDE theory and mathematical physics;

codomain dimension The Isabelle formalization only works for finite dimensional codomain while my formalization works for any Banach space; this is needed, for example, in spectral theory for the resolvent.

The complex analysis part of the project does not try to compete with the state of the art complex analysis libraries in Mizar [9] and Isabelle/HOL [10], both originally written by J. Harrison, and is provided mostly as a proof of concept (and the begining of a new project).

All these theorems are already in the `master` branch of the Lean [7] mathematical library `mathlib` [11]. For presentational purposes, I changed some names (the `mathlib` naming convention sometimes leads to very long names) and notation in the code listings below.

Links to the actual formal definitions in `mathlib` are marked with the symbol ↗. The links point to a website owned by me that redirects to the code in a specific version of `mathlib`. From time to time, I will update them to point to the current `master`.

Structure of the paper

In Sec. 2 I informally discuss different ways to generalize the divergence theorem mentioned above, including comparison with the formalization in Isabelle. Then in Sec. 3 I give various definitions related to the Henstock-Kurzweil, McShane, and GP-integrals and formulate the divergence theorem for the GP-integral. In Sec. 4 I discuss some design choices I made in this project. Finally, in Sec. 5 I explain how to apply this theorem to prove the Cauchy-Goursat theorem and some other basic theorems from complex analysis. Sec. 6 is devoted to my future plans.

2 Generalizations of the divergence theorem

The divergence theorem for continuously differentiable vector fields on rectangular boxes can be generalized in a few different directions, leading to several theorems, none of which implies the others.

2.1 Shape of the domain

One natural direction of generalization is to deal with non-rectangular domains. This generalization is done by Abdulaziz and Paulson in Isabelle [1] for regions that can be divided both into type I regions by vertical lines only and into type II regions by horizontal lines only. I only deal with rectangular boxes, so I am not going to discuss this in any more details.

2.2 More general codomain

Another possible direction of generalization is to deal with functions $f\colon \mathbb{R}^n \to E^n$, where E is a real Banach space, instead of vector fields $v\colon \mathbb{R}^n \to \mathbb{R}^n$ and E^n is the direct sum of n copies of E. Most proofs that work for a vector field can be generalized to this case with no or little modifications. I deal with functions $\mathbb{R}^n \to E^n$ right away.

2.3 Integrable partial derivatives

It is easy to see that the standard proof based on Fubini's theorem and FTC works for a function $f\colon \mathbb{R}^n \to E^n$ such that the partial derivatives $\frac{\partial f^i(x)}{\partial x^i}$ exist at all points of the box and are integrable on the whole box.

▶ Remark 1. Here and below I use upper indices for coordinates of vectors to disambiguate i-th coordinate of a vector from i-th element in a sequence of vectors. I do not assume implicit summation over indices that appear both as an upper and as a lower index.

Neither this generalization nor the next one imply each other. I formalized the other one because it implies the Cauchy–Goursat theorem.

2.4 Fréchet differentiability

A function $f\colon E_1 \to E_2$ between normed vector spaces is called *Fréchet differentiable* at a point x with the derivative given by a continuous linear map $f'\colon E_1 \to E_2$ if $f(x+y) = f(x) + f'(y) + o(y)$ as $y \to 0$. A Fréchet differentiable function f has all partial derivatives and they are equal to the values of f' on the basis vectors.

In 1981, Mawhin [12] introduced a generalization of the Henstock-Kurzweil integral he called the *GP-integral* (probably from *generalized Perron*). This integral allows us to prove the following theorem.

▶ **Theorem 2** (see [12]). *Let E be a real Banach space. Let $f\colon \mathbb{R}^n \to E$ be a function that is Fréchet differentiable at all points of a closed box $\overline{I} = [a, b]$.*

Then for each $i = 1, \dots, n$, the partial derivative $\frac{\partial f}{\partial x^i}$ is GP-integrable on I and its integral is equal to the difference of the integrals of f over the faces $x^i = b^i$ and $x^i = a^i$ of I.

In particular, for a function $f\colon \mathbb{R}^n \to E^n$, the divergence

$$\operatorname{div} f = \sum_{i=1}^{n} \frac{\partial f^i}{\partial x^i} \tag{2}$$

is GP-integrable on I and its integral is equal to the sum of integrals of f over the faces of I with appropriate signs.

Compared to the previous generalization, this one requires differentiability in a stronger sense (Fréchet differentiability instead of existence of the partial derivatives) but it requires less regularity from the derivatives (no requirements for the GP-integral and integrability of the divergence instead of integrability of partial derivatives for most other integrals).

2.5 Weaker assumptions on a subset of the box

One can push arguments in the proofs discussed above to weaken the regularity assumptions on a "small" subset of the box.

I prove that Theorem 2 works even if on some countable subset of the box the function is continuous, not differentiable. It is possible to push these arguments even further (especially if we use another generalization of the Henstock-Kurzweil integral) but I did not formalize these theorems (yet).

3 Henstock-Kurzweil integral: informal description

While my main goal was to prove the divergence theorem for the Bochner integral, I first proved the divergence theorem for the GP-integral, see below. This integral is one of possible generalizations of the Henstock-Kurzweil integral to higher dimension. In this section, I state the relevant definitions and theorems.

3.1 Henstock-Kurzweil integral in dimension one

I start with the definition of the one-dimensional Henstock-Kurzweil integral.

▶ **Definition 3.** *We say that a tagged partition $a = x_0 < x_1 < \cdots < x_k = b$ with tags $\xi_i \in [x_i, x_{i+1}]$, $0 \leqslant i < k$, is* subordinate *to a function $\delta\colon [a,b] \to \mathbb{R}$ if $[x_i, x_{i+1}] \subset [\xi_i - \delta(\xi_i), \xi_i + \delta(\xi_i)]$ for all $0 \leqslant i < k$.*

We say that $f\colon \mathbb{R} \to E$ has the Henstock-Kurzweil integral c over an interval $[a,b]$, $\int_a^b f(x)\,dx = c$, if for any $\varepsilon > 0$ there exists an everywhere positive function $\delta\colon [a,b] \to \mathbb{R}$ such that for any tagged partition $(\{x_i\}, \{\xi_i\})$ subordinate to δ, the Riemann sum $\sum_{i=0}^{k-1} f(\xi_i)(x_{i+1} - x_i)$ is ε-close to c.

Slight modifications of Definition 3 give the integrals of Riemann (we require δ to be a constant) and McShane (we no longer require that tags belong to their intervals but still require $\xi_i \in [a,b]$).

An equivalent (but more complicated) definition was given by A. Denjoy to integrate $\frac{1}{x}\sin\left(\frac{1}{x^3}\right)$ over $[-1,1]$. This function is instegrable in the sense of the Henstock-Kurzweil integral but it is not Lebesgue integrable.

A well-known property of the Henstock-Kurzweil integral is the following form of the Fundamental Theorem of Calculus.

▶ **Theorem 4.** *Let E be a real Banach space. Let $f\colon \mathbb{R} \to E$ be a function differentiable on an interval $[a,b]$. Then its derivative is Henstock-Kurzweil integrable on $[a,b]$ and the integral is equal to $f(b) - f(a)$.*

The one-dimensional Henstock-Kurzweil integral is not a part of `mathlib` and I decided not to formalize it. When we need it, it will be easy to define it as a thin wrapper on top of the Henstock-Kurzweil integral over a box in $\mathbb{R}^1$ `= fin 1 →` $\mathbb{R}$.

3.2 Henstock-Kurzweil integral in higher dimension

We will need a few definitions to generalize the Henstock-Kurzweil integral to functions $f\colon \mathbb{R}^n \to E$.

▶ **Definition 5.** *For $a, b \in \mathbb{R}^n$, the* open box *(a,b) is the product of open intervals (a^i, b^i), that is $(a,b) = \{x \in \mathbb{R}^n \mid \forall i, x^i \in (a^i, b^i)\}$. The* open-closed box *$(a,b]$ and the* closed box *$[a,b]$ are defined in a similar way.*

The distortion *(or* irregularity*) $\sigma(I)$ of an open-closed box $I = (a,b]$ is the maximum of the ratios $\frac{b^i - a^i}{b^j - a^j}$ over all i, j.*

A partition *of an open-closed box $I = (a,b]$ is a finite collection of pairwise disjoint boxes $J_k = (a_k, b_k]$ that cover I.*

A tagged partition *of an open-closed box $I = (a,b]$ is a partition $\{J_k\}$ together with a collection of points called* tags *$\xi_k \in [a,b]$, one per each box of the partition.*

A Henstock tagged partition *is a tagged partition such that each tag belongs to the corresponding closed box: $\xi_k \in [c_k, d_k]$, where $J_k = (c_k, d_k]$.*

A tagged partition $\{(J_k, \xi_k)\}$ of a box I is subordinate *to a* gauge function *$\delta\colon \mathbb{R}^n \to \mathbb{R}_{>0}$ if each box J_k is included by the closed box $B_{\delta(\xi_k)}(\xi_k) = [\xi_k - \delta(\xi_k), \xi_k + \delta(\xi_k)]$.*

Now we are ready to define the Henstock-Kurzweil integral of a function $f\colon \mathbb{R}^n \to E$ over a box I.

▶ **Definition 6.** *Let E be a real Banach space. We say that a function $f\colon \mathbb{R}^n \to E$ is* Henstock-Kurzweil integrable *on I with integral c if for any $\varepsilon > 0$ there exists a gauge function $\delta\colon \mathbb{R}^n \to \mathbb{R}_{>0}$ such that for any Henstock tagged partition $\{(J_k, \xi_k)\}$ of I that is subordinate to δ, the Riemann sum of f over this partition is ε-close to c.*

Similarly to the one-dimensional case, small modifications of Definition 6 provide definitions of Riemann and McShane integrability, see the paragraph after Definition 3.

Divergences of some vector fields are not integrable in the sense of this definition (and any other generalization of the Henstock-Kurzweil integral that satisfies Fubini's theorem, as shown by W. Pfeffer [13]), so we need another generalization to prove the divergence theorem.

There are several generalizations of the Henstock-Kurzweil integral such that the divergence of any Fréchet differentiable function is integrable, see Bongiorno's survey [6]. I use the one suggested by Mawhin [12].

▶ **Definition 7.** *The* distortion *(or* irregularity*) $\Sigma(\pi)$ of a partition $\pi = \{J_k\}$ is the maximum of the distortions $\sigma(J_k)$ of the boxes J_k.*

A function $f\colon \mathbb{R}^n \to E$ is GP-integrable *on a box I with integral c if for any $\varepsilon > 0$ and a real number d there exists a gauge function $\delta\colon \mathbb{R}^n \to \mathbb{R}_{>0}$ such that for any Henstock tagged partition $\{(J_k, \xi_k)\}$ of I that is subordinate to δ and has a distortion less than d, the Riemann sum of f over this partition is ε-close to c.*

I formalized this definition (together with a few other definitions of "box" integrals) and a proof of Theorem 2.

In dimension two, this theorem implies (1) for any pair of functions $f, g\colon \mathbb{R}^2 \to E^2$ Fréchet differentiable on a closed rectangle.

3.3 Application to the Bochner integral

The mathlib project uses the Bochner integral as its main integral [14]. This integral is a generalization of the Lebesgue integral. It was introduced by Bochner in [4], and it is also formalized in Coq [5] and Isabelle [3].

A Bochner integrable function on a box in $\mathbb{R}^n$ is McShane integrable (hence Henstock-Kurzweil and GP-integrable), thus Theorem 2 holds for the Bochner integral if we assume that the divergence is Bochner integrable on the box. Taking the limit over an exhaustion of the box by smaller boxes, one can generalize the theorem to functions that are differentiable in the interior of a box and continuous on the whole box. Here is the precise statement of the divergence theorem for Bochner integral.

▶ **Theorem 8.** *Let E be a real Banach space. Let $I = (a, b] \subset \mathbb{R}^n$ be an open-closed box. Let $s \subset \mathbb{R}^n$ be a countable set. Let $f\colon \mathbb{R}^n \to E^n$ be a function that is continuous on $\overline{I}$ and is Fréchet differentiable at all points of $\mathring{I}\backslash s$. Assume that the divergence* $\operatorname{div} f$ *defined by* (2) *is Bochner integrable on I. Then its integral is equal to the sum of integrals of f over the faces of $\overline{I}$ taken with appropriate signs (plus for the integral over a front face $x^i = b^i$ and minus for the integral over a back face $x^i = a^i$).*

4 Divergence theorem: design choices and implementation details

The code below uses some notation that is specific either to `mathlib` or this project.

`Icc a b`	the closed interval $[a, b]$;
`Ioo a b`	the open interval (a, b);
`Ioc a b`	the open-closed interval $(a, b]$;
`ICC a b`	the unordered closed interval $[\min(a, b), \max(a, b)]$;
`IOO a b`	the unordered open interval $(\min(a, b), \max(a, b))$;
$\mathbb{R}^n$	the vector space $\mathbb{R}^n$, implemented as `fin n → ℝ`, where `fin n` is the canonical type with n elements;
$\mathtt{E}^n$	the direct sum of n copies of a vector space E, implemented as `fin n → E`; if $E = \mathbb{R}$, then this notation agrees with the previous one;
`ℝ>0, ℝ≥0`	the types of positive and nonnegative real numbers, respectively;
`e i`	i-th basis vector in $\mathbb{R}^n$;
i	the imaginary unit;
`s ×ℂ t`	the product of sets on the real and imaginary axes in $\mathbb{C}$.

4.1 Boxes

I use open-closed boxes in $\mathbb{R}^n$ as elements of partition because, this way, the boxes are disjoint as sets and cover the whole ambient box, so one does not have to deal with interiors or closures to define a partition.

```
structure box (n : ℕ) : Type :=
(lower upper : ℝ^n)
(lower_lt_upper : ∀ i, lower i < upper i)
```

Each box can be interpreted as a set in $\mathbb{R}^n$.

```
instance : has_mem (ℝ^n) (box n) :=
⟨λ x I, ∀ i, x i ∈ Ioc (I.lower i) (I.upper i)⟩

instance : has_coe_t (box n) (set} ℝ^n) := ⟨λ I, {x | x ∈ I}⟩
```

The order on the boxes is the inclusion order on the corresponding sets.

I chose to explicitly deny empty boxes because this way the `lower` and `upper` vertices are uniquely defined by the set of points that belong to the box. The empty box is represented as the bottom element `⊥ : with_bot (box n)`, where `with_bot α` is the type `{⊥} ∪ α`, implemented as a type synonym for `option α` with custom order.

From the order theory point of view, the type `with_bot (box n)` is a lattice, where the meet of two boxes is their intersection and the join of two boxes is the minimal box that includes both of them.

For an open-closed box `I : box n`, `I.Ioo` and `I.Icc` are the corresponding open and closed boxes, respectively.

Given a box `I : box (n + 1)` and an index `i : fin (n + 1)`, I define the i-th *face* of I to be the box in $\mathbb{R}^n$ with lower and upper vertices given by `I.lower ∘ i.succ_above` and `I.upper ∘ i.succ_above`, where `i.succ_above : fin n → fin (n + 1)` is the unique monotone embedding leaving a "hole" at i: it sends $j < i$ to j and $j \geqslant i$ to $j + 1$. I also define two embeddings `I.front_face i, I.back_face i : ℝ^n → ℝ^(n+1)` that insert `I.upper i` and `I.lower i`, respectively, as the i-th coordinate. These embeddings map `I.face i` to the faces `x i = I.upper i` and `x i = I.lower i` of I.

4.2 Partitions

A *partition* of a box I is a finite collection of pairwise disjoint boxes that cover I. Sometimes, it is useful to deal with a collection of pairwise disjoint boxes that cover only a part of I, so I define a *prepartition* of a box and a predicate `is_partition` saying that a prepartition is actually a partition.

```
structure prepartition (I : box n) : Type :=
(boxes : finset (box n))
(le_of_mem' : ∀ J ∈ boxes, J ≤ I)
(pairwise_disjoint : pairwise boxes
  (disjoint on (coe : box n → set ℝ^n)))

def is_partition (π : prepartition I) : Prop :=
∀ x ∈ I, ∃ J ∈ π, x ∈ J
```

I do not use two separate structures `prepartition` and `partition` because with my approach I can define a function of prepartitions (e.g., the Riemann integral sum), then prove statements about its limits along various filters; some of these filters require a prepartition to be a partition, some do not. Also, I can define operations on prepartitions, then freely mix partitions and prepartitions in the arguments instead of adding an explicit `to_prepartition` or an implicit coercion here and there. On the other hand, I have to use `(π : prepartion I) (hπ : is_partition π)` instead of `(π : partition I)` whenever I want to argue about a partition, so I am not completely sure that I made the right choice.

I establish basic API about (pre)partitions, most notably the following predicates, relations, and operations.

- $\pi_1 \leqslant \pi_2$ — we say that one prepartition is less than or equal to another if each box of the former is included in some box of the latter; with this order, we get a bounded meet-semilattice structure on prepartitions of a box;
- `Union` — the union of all boxes of a prepartition, as a set in $\mathbb{R}^n$; for a partition, this union is equal to the original box;
- `bUnion` — given a prepartition π of I and a function π' that sends each box in $\mathbb{R}^n$ to a prepartition of that box, returns the prepartition of I formed by the boxes of $\pi' J$, $J \in \pi$;
- `split_center` — the partition of a box into 2^n subboxes by the coordinate hyperplanes passing through the center of the box;
- `split` — the partition of a box into two subboxes by a coordinate hyperplane (or the trivial one-box partition if the hyperplane does not meet the box);
- `split_many` — the partition of a box into subboxes by a finite set of coordinate hyperplanes;
- `compl` — an unspecified prepartition such that `π.compl.Union = I \ π.Union`; I prove that it exists, then use the axiom of choice to get a witness.

4.3 Tagged partitions

Recall that a *tagged (pre)partition* is a (pre)partition π with a point ("tag") chosen in each box of π. In the formal definition I require that the `tag` function is defined on all boxes. This way I can write `π.tag J` without proving that J is one of the boxes of π.

```
structure tagged_prepartition (I : box n) extends prepartition I :=
(tag : box n → ℝ^n)
(tag_mem_Icc : ∀ J, tag J ∈ I.Icc)
```

Here the `extends` keyword implicitly adds a field `to_prepartition` to the structure and introduces some syntatic sugar for composed projections (e.g., given π of type `tagged_prepartition I`, Lean unfolds π`.boxes` to π`.to_prepartition.boxes`) and constructors.

Unfortunately, similar syntax does not work for other definitions and lemmas in the same namespace, so I have to repeat some definitions and lemmas about prepartition once more (with one line proofs that reference the corresponding lemma about prepartitions). I hope that this tedious work will be automized in a future version of Lean or mathlib.

There are a few new definitions about tagged prepartitions that essentially use the tags, see Definition 5.

```
def is_Henstock (π : tagged_prepartition I) : Prop :=
∀ J ∈ π, π.tag J ∈ J.Icc

def is_subordinate (π : tagged_prepartition I) (r : ℝ^n → ℝ>0) : Prop :=
∀ J ∈ π, J.Icc ⊆ closed_ball (π.tag J) (r (π.tag J))
```

4.4 Cousin's lemma

Cousin's lemma says that for any gauge function $\delta\colon \mathbb{R}^n \to \mathbb{R}_{>0}$ and a box, there exists a tagged partition of this box that is subordinate to δ. This lemma is needed to show that the Henstock-Kurzweil integral is well defined: if it was false, any number would be the Henstock-Kurzweil integral of any function.

I prove two versions of this lemma. First I prove the lemma as stated in the previous paragraph with an additional assertion that all boxes of the partition are homothetic to the original box.

```
lemma exists_subordinate_Henstock (I : box n) (r : ℝ^n → ℝ>0) :
  ∃ π : tagged_prepartition I, π.is_partition ∧ π.is_Henstock ∧
    π.is_subordinate r ∧
    (∀ J ∈ π, ∃ m : ℕ, ∀ i, J.upper i - J.lower i =
      (I.upper i - I.lower i) / 2 ^ m) ∧
    π.distortion = I.distortion
```

Here is the sketch of the proof. If a box I does not admit a tagged partition with these properties, then the same is true for one of the 2^n boxes of the partition `I.split_center`. Thus we obtain an infinite sequence of boxes J_k, each one is twice smaller than the previous one in each direction, such that none of these boxes admit a partition with these properties. Let a be the unique common point of these boxes. For sufficiently large k, J_k is included in the closed ball with center a and radius $r(a)$, hence the one-box partition of J_k with tag at a satisfies all the required properties. This contradiction proves the lemma.

Then I use it to prove that for any prepartition π there exists a refinement of this prepartition with the same distortion, see Definition 7, and a choice of tags such that the resulting tagged prepartition is Henstock and is subordinate to a given gauge function. To prove this, I apply the previous lemma to each box of π, then merge these partitions using `prepartition.bUnion`. I use this version of Cousin's lemma to prove that the GP-integral is well defined.

```
lemma exists_le_Henstock_Union_eq (r : ℝ^n → ℝ>0) (π : prepartition I) :
  ∃ π' : tagged_prepartition I, π'.to_prepartition ⩽ π ∧
    π'.is_Henstock ∧ π'.is_subordinate r ∧ π'.distortion = π.distortion ∧
    π'.Union = π.Union :=
```

4.5 The filters

Different "box" integrals (Riemann, McShane, Henstock-Kurzweil, GP) are defined as the limits of the Riemann sums along some filters on the space of partitions of a box.

I define the structure `integration_params`[↗] that holds data needed to define either of these four integrals and a few more.

```
structure integration_params : Type :=
(bRiemann bHenstock bDistortion : bool)
```

The parameters have the following meaning:

`bRiemann`	this is a Riemann integral, the gauge function must be a constant;
`bHenstock`	tags must belong to the closure of their boxes;
`bDistortion`	the gauge function may depend on the distortion of a partition.

The integration parameters used to define the Riemann[↗], Henstock-Kurzweil[↗], McShane[↗], and GP[↗]-integrals are listed below.

```
def Riemann : integration_params :=
{ bRiemann := tt,
  bHenstock := tt,
  bDistortion := ff }

def Henstock : integration_params :=
{ bRiemann := ff,
  bHenstock := tt,
  bDistortion := ff }

def McShane : integration_params :=
{ bRiemann := ff,
  bHenstock := ff,
  bDistortion := ff }

def GP : integration_params :=
{ bRiemann := ff,
  bHenstock := tt,
  bDistortion := tt }
```

On one hand, this design choice allows me to prove some lemmas (e.g., Henstock-Sacks inequality, see Sec. 4.8) uniformly for all these integrals. On the other hand, it is hard to add more integrals to the collection, see Bongiorno's survey [6] for other reasonable generalizations of the Henstock-Kurzweil integral to higher dimension.

I require that the gauge function satisfies the following condition[↗].

```
def r_cond {n : ℕ} (l : integration_params) (r : ℝ^n → ℝ>0) : Prop :=
l.bRiemann → ∀ x, r x = r 0
```

If `l.bRiemann` is false, then this condition is trivial, otherwise it says that the gauge function is actually a constant function.

For each set of parameters `l : integration_params` and a box I, we define several filters on the space of prepartitions of I. All these filters have basis sets of the same type[↗].

```
structure mem_base_set (l : integration_params) (I : box n) (c : ℝ≥0)
  (r : ℝ^n → ℝ>0) (π : tagged_prepartition I) : Prop :=
(is_subordinate : π.is_subordinate r)
(is_Henstock : l.bHenstock → π.is_Henstock)
```

```
(distortion_le : l.bDistortion → π.distortion ⩽ c)
(exists_compl : l.bDistortion → ∃ π' : prepartition I,
  π'.Union = I \ π.Union ∧ π'.distortion ⩽ c)
```

We already saw the first three assumptions in the informal definition of the GP-integral. The last assumption is trivial if π is a partition (then we can choose π' to be the empty prepartition). One can show that it is also trivial whenever $c > 1$ but I decided to cut a corner here and introduce this assumption instead of adding an explicit assumption $c > 1$ here and there and proving one more lemma.

The most important filter related to a set of integration parameters is the filter `to_filter_Union` below. Cousin's lemma from Sec. 4.4 implies that this filter is non-trivial. The actual definition uses more intermediate filters but the result is equal (though not definitionally equal) to the code below.

```
def to_filter (l : integration_params) (I : box n) :
  filter (tagged_prepartition I) :=
⨆ c : ℝ⩾0, ⨅ (r : ℝ^n → ℝ>0) (hr : l.r_cond r),
  𝓟 {π | l.mem_base_set I c r π}

def to_filter_Union (l : integration_params) (I : box n)
  (π₀ : prepartition I) :=
l.to_filter I ⊓ 𝓟 {π | π.Union = π₀.Union}
```

The filter `to_filter` is useful to prove facts about integrals on subboxes.

4.6 Box-additive function

I introduce the following definition.

▶ **Definition 9.** *A function on boxes in $\mathbb{R}^n$ taking values in an additive group is said to be* box-additive *if for any box I and its partition π, the sum of its values on the boxes of π is equal to its value on I.*

A function is said to be box-additive on subboxes of I_0 *if the same property holds true whenever $I \leqslant I_0$.*

In order to deal with these two notions simultaneously, the actual definition takes an argument of the type `with_top (box n)`. Similarly to the type `with_bot (box n)`, this type is the disjoint union of `box n` and the top element $\top$.

```
structure box_additive_map (n : ℕ) (G : Type*) [add_comm_group G]
  (I : with_top (box n)) : Type* :=
(to_fun : box n → G)
(sum_partition_boxes' : ∀ J : box n, (J : with_top (box n)) ⩽ I →
  ∀ π : prepartition J, π.is_partition →
    ∑ Ji in π.boxes, to_fun Ji = to_fun J)
```

I use notation `n →ba G` for functions that are box-additive on the whole space and `n →ba[I] G` for functions that are box-additive on subboxes of I.

It suffices to verify additivity only on the two-box partitions `split` introduced above. Indeed, let t be the set of all hyperplanes that contain faces of a partition π of I. Then the partition `split_many I t` can be obtained by a series of splits along a single hyperplane both from the trivial one-box partition and from π. Therefore, if f is additive on the two-box partitions, then the sum of values of f over all boxes of `split_many I t` is equal both to $f(I)$ and to the sum of its values over all boxes of π.

```
def of_map_split_add (f : box n → G) (I₀ : with_top (box n))
  (hf : ∀ I : box n, (I : with_top (box n)) ⩽ I₀ →
    ∀ {i x}, x ∈ Ioo (I.lower i) (I.upper i) →
    f (I.split_lower i x) + f (I.split_upper i x) = f I) :
  n →ᵇᵃ[I₀] G
```

Here `I.split_lower i x` and `I.split_upper i x` are the boxes of the partition `split I`. Since one of them can be empty, they have type `with_bot (box n)`, so the actual code looks like `option.elim (I.split_lower i x) 0 f` instead of `f (I.split_lower i x)`.

Each locally finite measure μ defines a box-additive function. Next, if $f\colon \mathbb{R}^n \to E$ is integrable (in any reasonable sense) on a box I, then its integral over a box is a box-additive function on subboxes of I, see Sec. 4.8.

One more construction of box-additive functions appears in the proof of the divergence theorem. Given a box-additive function f_y on each cross-section $x^i = y$, $y \in [a^i, b^i]$ of a closed box `I₀.Icc = [a, b]`, the function given by `g J = f (J.upper i) (J.face i) - f (J.lower i) (J.face i)` is box additive on subboxes of I_0.

To ensure nice definitional equality properties of the result, the actual definition involves two functions and a proof that they are equal on all the boxes relevant to the definition.

```
def upper_sub_lower {G : Type u} [add_comm_group G]
  (I₀ : box (n + 1)) (i : fin (n + 1)) (f : ℝ → box n → G)
  (fb : Icc (I₀.lower i) (I₀.upper i) → n →ᵇᵃ[I₀.face i] G)
  (hf : ∀ x (hx : x ∈ Icc (I₀.lower i) (I₀.upper i)) J,
    f x J = fb ⟨x, hx⟩ J) :
  (n + 1) →ᵇᵃ[I₀] G :=
```

4.7 Box integral

The box integral of a function $f\colon \mathbb{R}^n \to E$ on an open-closed box I in the sense of integration parameters l is defined as the limit of the Riemann sum over a partition π of I along the filter `l.to_filter_Union` $\top$, where $\top$ is the one-box partition of I. If the limit does not exist, then the integral is defined to be zero.

I define the integral of a function $f\colon \mathbb{R}^n \to E$ with respect to a box-additive volume taking values in the space of continuous linear functions `E →L[ℝ] F`. This way one can use the same definition, e.g., for Riemann-Stieltjes integrals. However, I only used this definition in the case $E = F$ and `vol J x = (μ J).to_real • x` for some measure μ. So, this generalization might be a case of overengineering.

```
def integral_sum (f : ℝ^n → E) (vol : n →ᵇᵃ (E →L[ℝ] F))
  (π : tagged_prepartition I) : F :=
∑ J in π.boxes, vol J (f (π.tag J))

def has_integral (I : box n) (l : integration_params) (f : ℝ^n → E)
  (vol : n →ᵇᵃ (E →L[ℝ] F)) (y : F) : Prop :=
tendsto (integral_sum f vol) (l.to_filter_Union I ⊤) (𝓝 y)

def integrable (I : box n) (l : integration_params) (f : ℝ^n → E)
  (vol : n →ᵇᵃ (E →L[ℝ] F)) : Prop :=
∃ y, has_integral I l f vol y
```

```
def integral (I : box n) (l : integration_params) (f : ℝ^n → E)
  (vol : n →ᵇᵃ (E →L[ℝ] F)) : F :=
if h : integrable I l f vol then h.some else 0
```

Usual theorems (uniqueness of the integral, Cauchy convergence test, additivity) immediately follow from the definition and properties of the limit.

4.8 The Henstock-Sacks inequality

The Henstock-Sacks inequality for the Henstock-Kurzweil integral says the following. Let f be a function integrable on a box I; let $\delta\colon \mathbb{R}^n \to \mathbb{R}_{>0}$ be a gauge function such that for any tagged partition of I subordinate to δ, the integral sum over this partition is ε-close to the integral. Then for any tagged *prepartition* π, the integral sum over π differs from the integral of f over the part of I covered by π by at most ε.

This inequality is used, e.g., to prove that a function that is Henstock-Kurzweil integrable on a box I, is Henstock-Kurzweil integrable on any subbox of I and defines a box-additive function on subboxes of I. I prove several versions of this inequality for any of the "box" integrals.

Instead of using predicate assumptions on δ, I define `convergence_r` to be a function $\delta\colon \mathbb{R}^n \to \mathbb{R}_{>0}$ such that

- if `l.bRiemann`, then δ is a constant;
- if $\varepsilon > 0$, then for any tagged partition π of I satisfying the predicate `l.mem_base_set I c δ`, the integral sum of f over π differs from the integral of f over I by at most ε.

```
def convergence_r (h : integrable I l f vol) (ε : ℝ) (c : ℝ≥0) :
  ℝ^n → ℝ>0
```

Let me quote two versions of the Henstock-Sacks inequality here. One version compares the Riemann sums of a function over two prepartitions that cover the same part of the box.

```
lemma dist_integral_sum_le_of_mem_base_set (h : integrable I l f vol)
  (hpos₁ : 0 < ε₁) (hpos₂ : 0 < ε₂)
  (hπ₁ : l.mem_base_set I c₁ (h.convergence_r ε₁ c₁) π₁)
  (hπ₂ : l.mem_base_set I c₂ (h.convergence_r ε₂ c₂) π₂)
  (hU : π₁.Union = π₂.Union) :
  dist (integral_sum f vol π₁) (integral_sum f vol π₂) ⩽ ε₁ + ε₂
```

The other version replaces one of these Riemann sums with the sum of integrals of the function over the boxes of the partition.

```
lemma dist_integral_sum_sum_integral_le_of_mem_base_set_of_Union_eq
  (h : integrable I l f vol) (hpos : 0 < ε)
  (hπ : l.mem_base_set I c (h.convergence_r ε c) π)
  (hU : π.Union = π₀.Union) :
  dist (integral_sum f vol π) (∑ J in π₀.boxes, integral J l f vol) ⩽ ε
```

4.9 Divergence theorem for the GP-integral

To prove the divergence theorem for the GP-integral Theorem 2, we prove that each partial derivative is GP-integrable with the integral equal to the difference of the integral of the original function over the front and back faces of the box.

```
lemma has_integral_GP_pderiv (f : ℝ^(n+1) → E) (f' : ℝ^(n+1) → ℝ^(n+1) →L[ℝ] E)
  (s : set (ℝ^(n+1))) (hs : countable s)
  (Hs : ∀ x ∈ s, continuous_within_at f I.Icc x)
  (Hd : ∀ x ∈ I.Icc \ s, has_fderiv_within_at f (f' x) I.Icc x)
  (i : fin (n + 1)) :
  has_integral I GP (λ x, f' x (e i)) volume
    (integral (I.face i) GP (f ∘ I.front_face i) volume -
      integral (I.face i) GP (f ∘ I.back_face i) volume)
```

Here and in the next listing, *volume* is a notation for the box-additive volume `volume : (n + 1) →ᵇᵃ (E →L[ℝ] E)` defined by `dV J x = (∏ j, J.upper j - J.lower j) · x`.

The sum of these statements for all terms of the divergence gives us the divergence theorem.

```
lemma has_integral_GP_divergence (f : ℝ^(n+1) → E^(n+1))
  (f' : ℝ^(n+1) → ℝ^(n+1) →L[ℝ] E^(n+1)) (s : set ℝ^(n+1)) (hs : countable s)
  (Hs : ∀ x ∈ s, continuous_within_at f I.Icc x)
  (Hd : ∀ x ∈ I.Icc \ s, has_fderiv_within_at f (f' x) I.Icc x) :
  has_integral I GP (λ x, ∑ i, f' x (e i) i) volume
    (∑ i, (integral (I.face i) GP (f ∘ I.front_face i) volume -
      integral (I.face i) GP (f ∘ I.back_face i) volume))
```

The proof of the main lemma follows the same scheme as the standard proof of the Fundamental Theorem of Calculus for the Henstock-Kurzweil integral: given a positive number ε and an upper estimate c on the distortion of the partition, for each point x of differentiability, one can choose $\delta(x) > 0$ such that the estimate $f(y) = f(x) + f'(x)(y - x) + o(y - x)$ and $\Sigma(\pi) \leqslant c$ imply that the difference between the integrals of f over the i-th front and back faces of a $\delta(x)$-small box $J \ni x$ is $\varepsilon V(J)$-close to the term $\frac{\partial f}{\partial x^i} V(J)$ of the Riemann sum for the integral of the partial derivative.

I do one modification to this argument that allows me to use weaker assumptions on a countable set of points. Namely, for $x \in s$ I choose $\delta(x)$ so that for a $\delta(x)$-small box $J \ni x$, the term corresponding to this box has norm less than $\kappa(x) > 0$, where $\sum_{x \in s} \kappa(x) < \varepsilon/2$. This is possible due to the continuity of f at x.

4.10 McShane and Bochner integrability

In order to transfer the result from the GP-integral to the Bochner integral, I prove that any Bochner integrable function is integrable in the sense of any box integral with `bRiemann = ff`. In other words, a Bochner integrable function is integrable in the sense of the McShane, Henstock-Kurzweil, and GP-integrals.

```
lemma integrable_on.has_box_integral {f : ℝ^n → E} {μ : measure ℝ^n}
  [is_locally_finite_measure μ] {I : box n} (hf : integrable_on f I μ)
  (l : integration_params) (hl : l.bRiemann = ff) :
  has_integral I l f μ.to_box_additive.to_smul (∫ x in I, f x ∂μ)
```

The proof follows R. Gordon's book [8], with some modifications required to generalize it from a function $f\colon \mathbb{R} \to \mathbb{R}$ to a function $f\colon \mathbb{R}^n \to E$.

First, I prove that the indicator function of a measurable set s is McShane integrable with respect to the volume defined by a locally finite measure μ. This immediately implies integrability of simple functions. Then I show that changing the values of a function on a set of measure zero does not affect its integral. Finally, I use approximations of integrable functions by a sequence of simple functions to prove McShane integrability of a Bochner integrable function.

4.11 Divergence theorem for the Bochner integral

Since any Bochner integrable function is GP-integrable, we immediately obtain the divergence theorem for the Bochner integral under assumptions that the function is continuous on a closed box, is differentiable at all but countably many points of this box, and has integrable divergence.

```
lemma integral_divergence_aux (I : box (n + 1)) (f : ℝ^(n+1) → E^(n+1))
  (f' : ℝ^(n+1) → ℝ^(n+1) →L[ℝ] E^(n+1)) (s : set ℝ^(n+1)) (hs : countable s)
  (Hc : continuous_on f I.Icc)
  (Hd : ∀ x ∈ I.Icc \ s, has_fderiv_within_at f (f' x) I.Icc x)
  (Hi : integrable_on (λ x, ∑ i, f' x (e i) i) I.Icc) :
  ∫ x in I.Icc, ∑ i, f' x (e i) i =
    ∑ i : fin (n + 1),
      ((∫ x in (I.face i).Icc, f (I.front_face i x) i) -
        ∫ x in (I.face i).Icc, f (I.back_face i x) i)
```

I slightly generalize this result. First, I drop the differentiability assumption on the boundary of I. To do this, I apply the auxiliary result to an increasing sequence of subboxes that cover the interior of I. Second, I allow $a^i = b^i$; in this case both sides are equal to zero.

```
lemma integral_divergence (a b : ℝ^(n+1)) (hle : a ≤ b) (f : ℝ^(n+1) → E^(n+1))
  (f' : ℝ^(n+1) → ℝ^(n+1) →L[ℝ] E^(n+1)) (s : set ℝ^(n+1)) (hs : countable s)
  (Hc : continuous_on f (Icc a b))
  (Hd : ∀ x : ℝ^(n+1), (∀ i, x i ∈ Ioo (a i) (b i)) → x ∉ s,
    has_fderiv_at f (f' x) x)
  (Hi : integrable_on (λ x, ∑ i, f' x (e i) i) (Icc a b)) :
  ∫ x in Icc a b, ∑ i, f' x (e i) i =
    ∑ i : fin (n + 1), ((∫ x in face i, f (front_face i x) i) -
      ∫ x in face i, f (back_face i x) i)
```

5 Applications to complex analysis

The main goal of the project was to formalize a version of the divergence theorem that implies the Cauchy integral formula under standard assumptions. In this section, I will briefly explain how I deduce the Cauchy integral formula from Theorem 8. While many textbooks on complex analysis prove these formulas for functions $f\colon \mathbb{C} \to \mathbb{C}$, I prove them for functions that take values in a complex Banach space E. As before, I have to assume that E has a second countable topology because of the way the Bochner integral is defined in mathlib.

First, consider an open rectangle $R = \{ z \mid a \leqslant \operatorname{Re} z \leqslant b, c \leqslant \operatorname{Im} z \leqslant d \}$ on the complex plane $\mathbb{C}$. If $f\colon \mathbb{C} \to E$ is continuous on R and is complex differentiable at all but countably many points of the interior of R, then one can apply the divergence theorem to the function $F\colon \mathbb{C} \to E^2$ given by $F(z) = (-if(z), f(z))$. Due to Cauchy-Riemann equations, the left-hand side of (1) equals zero. It is easy to see that the right-hand side is equal to the integral $\int_{\partial R} f(z)\,dz$. Thus we have the Cauchy-Goursat theorem for a rectangular domain.

```
lemma cauchy_theorem_rect (f : ℂ → E) (z w : ℂ)
  (s : set ℂ) (hs : countable s)
  (Hc : continuous_on f (ICC z.re w.re ×ℂ Icc z.im w.im))
  (Hd : ∀ x ∈ (IOO z.re w.re ×ℂ IOO z.im w.im) \ s,
    differentiable_at ℂ f x) :
  (∫ x in z.re..w.re, f (x + z.im * I)) -
    (∫ x in z.re..w.re, f (x + w.im * I)) +
    (I · ∫ y in z.im..w.im, f (re w + y * I)) -
    I · ∫ y in z.im..w.im, f (re z + y * I) = 0 :=
```

To formulate the Cauchy-Goursat theorem for an annulus and a circle, I define the circle integral $\oint_{|z-c|=R} f(z)\,dz$ as $\int_0^{2\pi} (c + Re^{\theta i})' f(c + Re^{\theta i})\,d\theta$.

```
def circle_map\link{circle_map} (c : ℂ) (R : ℝ) : ℝ → ℂ := λ θ, c + R *
    exp (θ * i)

def circle_integral (f : ℂ → E) (c : ℂ) (R : ℝ) : E :=
∫ θ in 0..2 * π, deriv (circle_map c R) θ • f (circle_map c R θ)
```

Applying the Cauchy-Goursat theorem for a rectangle to the function $F(z) = e^z f(c + e^z)$ on the rectangle $\ln r \leqslant \operatorname{Re} z \leqslant \ln R$, $0 \leqslant \operatorname{Im} z \leqslant 2\pi$, I prove the Cauchy-Goursat theorem for a function differentiable on an annulus.

```
lemma cauchy_thm_annulus {c : ℂ} {r R : ℝ} (h0 : 0 < r)
  (hle : r ⩽ R) {f : ℂ → E} {s : set ℂ} (hs : countable s)
  (hc : continuous_on f (closed_ball c R \ ball c r))
  (hd : ∀ z ∈ ball c R \ closed_ball c r \ s, differentiable_at ℂ f z) :
  ∮ z in C(c, R), f z = ∮ z in C(c, r), f z
```

Next, I apply this theorem to the function $\frac{f(z)}{z-c}$ (formally, $(z-c)^{-1} \cdot f(z)$ because f is a vector-valued function) and take the limit as r tends to 0 from the right. Thus, I prove the Cauchy integral formula for the value of a complex differentiable function at the center of a disk. This theorem will be later generalized to any point of the disk, but I will use this case to prove the general version.

```
lemma cauchy_integral_disk_center {R : ℝ} (h0 : 0 < R)
  {f : ℂ → E} {c : ℂ} {s : set ℂ} (hs : countable s)
  (hc : continuous_on f (closed_ball c R))
  (hd : ∀ z ∈ ball c R \ s, differentiable_at ℂ f z) :
  ∮ z in C(c, R), (z - c)⁻¹ • f z = (2 * π * i : ℂ) • f c :=
```

Applying this lemma to the function $(z-c) \cdot f(z)$, we prove the Cauchy-Goursat theorem for a disk.

```
lemma cauchy_thm_disk {R : ℝ} (h0 : 0 ⩽ R) {f : ℂ → E}
  {c : ℂ} {s : set ℂ} (hs : countable s)
  (hc : continuous_on f (closed_ball c R))
  (hd : ∀ z ∈ ball c R \ s, differentiable_at ℂ f z) :
  ∮ z in C(c, R), f z = 0 :=
```

Next, I show the Cauchy integral formula

$$f(w) = \frac{1}{2\pi i} \oint\limits_{|z-c|=R} \frac{f(z)}{z-w}\,dz$$

for any point of the open disc, $|w-c| < R$. To obtain this result, I apply the Cauchy-Goursat theorem to the function

$$g(z) = \begin{cases} f'(w), & \text{if } z = w; \\ \frac{f(z)-f(w)}{z-w}, & \text{otherwise.} \end{cases}$$

If f is differentiable at w, then g satisfies the assumptions of the previous theorem, hence the integrals of $\frac{f(z)}{z-w}$ and $\frac{f(w)}{z-w}$ over the circle $|z-c| = R$ are equal. It is easy to see that the latter integral equals $2\pi i f(w)$.

If w belongs to the countable set where f is not guaranteed to be differentiable, then the same formula follows from the previous case by continuity.

```
lemma cauchy_integral_disk {R : ℝ} {c w : ℂ} {f : ℂ → E} {s : set ℂ}
  (hs : countable s) (hw : w ∈ ball c R)
  (hc : continuous_on f (closed_ball c R))
  (hd : ∀ x ∈ ball c R \ s, differentiable_at ℂ f x) :
  (2 * π * I : ℂ)⁻¹ • ∮ z in C(c, R), (z - w)⁻¹ • f z = f w :=
```

The Cauchy integral formula immediately implies that a function $f\colon \mathbb{C} \to E$ that is complex differentiable on an open disk and is continuous on the corresponding closed disk must be analytic on the interior of this disk. The coefficients of the Taylor series are given by Cauchy integrals.

```
def cauchy_power_series (f : ℂ → E) (c : ℂ) (R : ℝ) :
  formal_multilinear_series ℂ ℂ E :=
λ n, continuous_multilinear_map.mk_pi_field ℂ _
  ((2 * π * I : ℂ)⁻¹ • ∮ z in C(c, R), (z - c) ^ (-n - 1 : ℤ) • f z)
```

```
lemma analytic_of_differentiable {R : ℝ≥0} {c : ℂ} {f : ℂ → E}
  {s : set ℂ} (hs : countable s) (hc : continuous_on f (closed_ball c R))
  (hd : ∀ z ∈ ball c R \ s, differentiable_at ℂ f z) (hR : 0 < R) :
  has_fpower_series_on_ball f (cauchy_power_series f c R) c R :=
```

I started to use these theorems to build a complex analysis library in Lean. For example, I proved the Riemann removable singularity theorem, several versions of the maximum modulus principle, Liouville's theorem, and the Schwarz lemma.

```
lemma removable_singularity
  (hd : ∀ᶠ z in 𝓝[≠] c, differentiable_at ℂ f z)
  (ho : is_o (λ z, f z - f c) (λ z, (z - c)⁻¹) (𝓝[≠] c)) :
  ∃ y : E, tendsto f (𝓝[≠] c) (𝓝 y) :=
```

Here $\mathcal{N}[\neq]$ `c` is the filter of punctured neighborhoods of c.

```
theorem max_modulus (hU : bounded U) (hd : diff_cont_on_cl ℂ f U)
  (hC : ∀ z ∈ frontier U, ‖f z‖ ≤ C) (hz : z ∈ closure U) :
  ‖f z‖ ≤ C :=
```

Here `diff_cont_on_cl` is a predicate saying that a function is differentiable on a set and is continuous on its closure.

```
theorem liouville (hf : differentiable ℂ f) (hb : bounded (range f)) :
  ∃ c, ∀ x, f x = c :=
```

```
lemma schwarz_lemma (hd : differentiable_on ℂ f (ball 0 R))
  (h_maps : maps_to f (ball 0 R) (ball 0 R)) (h₀ : f 0 = 0)
  (hz : abs z < R) :
  abs (f z) ≤ abs z :=
```

6 Future plans

6.1 Other integrals

One possible direction of improvement would be to formalize the divergence theorem for (some of the) other integrals listed in the Bongiorno's survey [6]. Some of these theorems allow the function to be continuous, not differentiable, on a countable set of *coordinate hyperplanes*. These theorems can be transfered to more general versions of the Cauchy-Goursat theorems.

The main obstacle for this project is that I define "box" integrals in a non-extensible way, see Sec. 4.5. So, to add more integrals (or to replace the GP-integral with a better one), one has to seriously refactor the definition.

6.2 Complex analysis in higher dimension

Most of the proofs discussed in Sec. 5 can be easily generalized to the case of a function $f\colon \mathbb{C}^n \to E$. I mentor a student who tries to generalize the Cauchy integral formula as a part of a course.

6.3 The Cauchy-Goursat theorem for any domain

My goal was to show that it is possible to deduce the Cauchy-Goursat theorem from the divergence theorem, so I formalized the Cauchy-Goursat theorem only for a few special cases (a rectangle, an annulus, and a disk). One can deduce the general Cauchy-Goursat theorem from the version for a rectangle but it requires quite a few topological lemmas that are not in `mathlib` yet.

6.4 One-dimensional complex analysis

Formalization of the Cauchy-Goursat theorem makes it possible to formalize many theorems from the one-dimensional complex analysis. I already formalized a few theorems, but, clearly, this is just the beginning.

My main goal for the next year or two is to formalize Ilyashenko's proof of the fact that a polynomial vector field on $\mathbb{R}^2$ has only finitely many limit cycles (i.e., isolated periodic solutions), at least in the case of hyperbolic singular points. The proof of this theorem heavily relies on the complex analysis.

7 Conclusion

This project demonstrates that formalization of a sufficiently general version of the divergence theorem can be used to prove the Cauchy-Goursat theorem and bootstrap complex analysis. I hope that other systems will adopt this approach to formalization of the divergence theorem, or a similar one based on a better generalization of the Henstock-Kurzweil integral, see Sec. 6.1.

References

1 Mohammad Abdulaziz and Lawrence Paulson. An Isabelle/HOL formalisation of green's theorem. *J Autom Reasoning*, 63:763–786, 2019. doi:10.1007/s10817-018-9495-z.

2 Felipe Acker. The missing link. *The Mathematical Intelligencer*, 18(3):4–9, June 1996. doi:10.1007/BF03024304.

3 Jeremy Avigad, Johannes Hölzl, and Luke Serafin. A formally verified proof of the central limit theorem. *Journal of Automated Reasoning*, 59(4):389–423, 2017. doi:10.1007/s10817-017-9404-x.

4 Salomon Bochner. Integration von Funktionen, deren Werte die Elemente eines Vektorraumes sind. *Fundamenta Mathematicae*, 20, 1933.

5 Sylvie Boldo, François Clément, and Louise Leclerc. A Coq formalization of the Bochner integral, 2022. arXiv:2201.03242.

6 Benedetto Bongiorno. The Henstock-Kurzweil integral. In E. PAP, editor, *Handbook of Measure Theory*, chapter 13, pages 587–615. North-Holland, Amsterdam, 2002. doi:10.1016/B978-044450263-6/50014-2.

7 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.

8 Russel A. Gordon. *The integrals of Lebesgue, Denjoy, Perron, and Henstock*, volume 4 of *Graduate Studies in Mathematics*. American Mathematical Society, Providence, R.I, 1955.

9 John Harrison. Formalizing basic complex analysis. In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar and Rhetoric*, pages 151–165. University of Białystok, 2007. URL: http://mizar.org/trybulec65/.

10 John Harrison. Formalizing an analytic proof of the prime number theorem. *Journal of Automated Reasoning*, 43(3):243–261, 2009. doi:10.1007/s10817-009-9145-6.

11 The mathlib community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 367–381, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372885.3373824.

12 Jean Mawhin. Generalized multiple Perron integrals and the Green-Goursat theorem for differentiable vector fields. *Czechoslovak Math. J.*, 31(106)(4):614–632, 1981.

13 Washek F. Pfeffer. *The Riemann approach to integration*, volume 109 of *Cambridge Tracts in Mathematics*. Cambridge University Press, Cambridge, 1993. Local geometric theory.

14 Floris van Doorn. Formalized Haar Measure. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:17, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ITP.2021.18.

Refinement of Parallel Algorithms down to LLVM

Peter Lammich
University of Twente, Enschede, The Netherlands

Abstract

We present a stepwise refinement approach to develop verified parallel algorithms, down to efficient LLVM code. The resulting algorithms' performance is competitive with their counterparts implemented in C/C++. Our approach is backwards compatible with the Isabelle Refinement Framework, such that existing sequential formalizations can easily be adapted or re-used. As case study, we verify a parallel quicksort algorithm, and show that it performs on par with its C++ implementation, and is competitive to state-of-the-art parallel sorting algorithms.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Theory of computation → Semantics and reasoning; Computing methodologies → Parallel algorithms

Keywords and phrases Isabelle, Concurrent Separation Logic, Parallel Sorting, LLVM

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.24

Supplementary Material *Software (Isabelle Formalization)*: `https://www21.in.tum.de/~lammich/isabelle_llvm_par/`

1 Introduction

We present a stepwise refinement approach to develop verified and efficient parallel algorithms. Our method can verify total correctness down to LLVM intermediate code. The resulting verified implementations are competitive with state-of-the-art unverified implementations. Our approach is backwards compatible to the Isabelle Refinement Framework (IRF), a powerful tool to verify efficient sequential software, such as model checkers [10, 7, 38], SAT solvers [24, 25, 11], or graph algorithms [22, 28, 29]. This paper adds parallel execution to the IRF's toolbox, without invalidating the existing formalizations, which can now be used as sequential building blocks for parallel algorithms, or be modified to add parallelization.

As a case study, we verify total correctness of a parallel quicksort algorithm, re-using an existing verification of state-of-the-art sequential sorting algorithms [27]. Our verified parallel sorting algorithm is competitive to state-of-the-art parallel sorting algorithms.

1.1 Overview

This paper is based on the Isabelle Refinement Framework, a continuing effort to verify efficient implementations of complex algorithms, using stepwise refinement techniques. Figure 1 displays the components of the Isabelle Refinement Framework.

The back end layer handles the translation from Isabelle/HOL to the actual target language. The instructions of the target language are shallowly embedded into Isabelle/HOL, using a state-error (SE) monad. An instruction with undefined behaviour, or behaviour outside our supported fragment, raises an error. The state of the monad is the memory, represented via a memory model. The code generator translates the instructions to actual code. These components form the trusted code base, while all the remaining components of the Isabelle Refinement Framework generate proofs. In the back-end, the preprocessor transforms expressions to the syntactically restricted format required by the code generator, proving semantic equality of the original and transformed expression. While there exist back ends for purely functional code [30, 21], and sequential imperative code [23, 26], this paper describes a back end for parallel imperative LLVM code (Section 2).

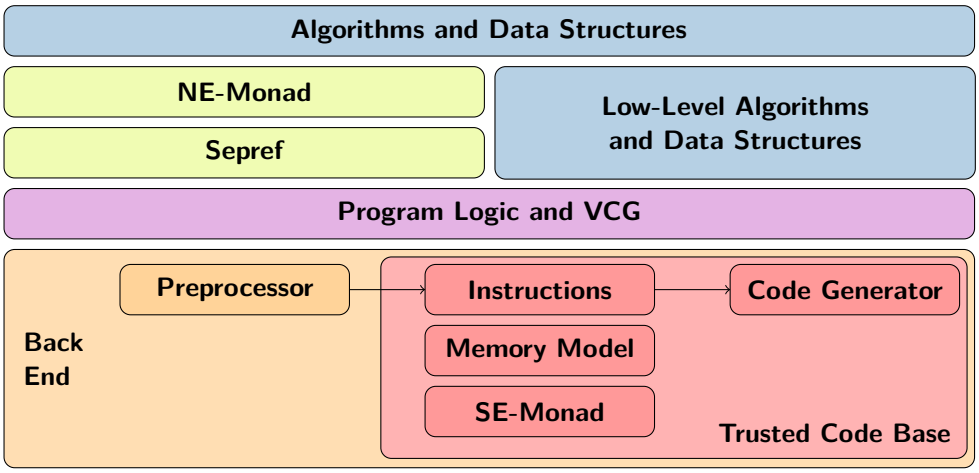


Figure 1 Components of the Isabelle Refinement Framework, with focus on the back end.

On top of the back-end, a program logic is used to prove programs correct. It uses separation logic, and provides automation like a verification condition generator (VCG). In Section 3, we describe our formalization of concurrent separation logic [33], and our VCG.

At the level of the program logic and VCG, our framework can already be used to verify simple low-level algorithms and data structures, like dynamic arrays and linked lists. More complex developments typically use a stepwise refinement approach, starting at purely functional programs modelled in a nondeterminism-error (NE) monad [30]. A semi-automatic refinement procedure (Sepref [23, 26]) translates from the purely functional code to imperative code, refining abstract functional data types to concrete imperative ones. In Section 4, we describe our extensions to support refinement to parallel executions, and a fine-grained tracking of pointer equalities, required to parallelize computations that work on disjoint parts of the same array.

Using our approach, complex algorithms and data structures can be developed and refined to optimized efficient code. The stepwise refinement ensures a separation of concerns between high-level algorithmic ideas and low-level optimizations. We have used this approach to verify a wide range of practically efficient algorithms [10, 7, 38, 24, 25, 11, 22, 28, 29, 27]. In Section 5, we use our techniques to verify a parallel sorting algorithm, with competitive performance wrt. unverified state-of-the-art algorithms.

Section 6 concludes the paper and discusses related and future work.

2 A Back End for LLVM with Parallel Execution

We formalize a semantics for parallel execution, shallowly embedded into Isabelle/HOL. As for the existing sequential back ends [23, 26], the shallow embedding is key to the flexibility and feasibility of the approach. The main idea is to make an execution report the memory that it accesses, and use this information to raise an error when joining executions that would have exhibited a data race. We use this to model an instruction that calls two functions in parallel, and waits until both have returned.

2.1 State-Nondeterminism-Error Monad with Access Reports

We define the underlying monad in two steps. We start with a nondeterminism-error monad, and then lift it to a state monad and add access reports. Defining a nondeterminism-error monad is straightforward in Isabelle/HOL:

```
'a neM ≡ spec ('a ⇒ bool) | fail
return x ≡ spec (λr. r=x)
bind fail f ≡ fail
bind (spec P) f ≡ if ∃x. P x ∧ f x = fail then fail
                  else spec (λr. ∃x Q. P x ∧ f x = spec Q ∧ Q r)
```

A program either fails, or yields a possible set of results (`spec` P), described by its characteristic function P. The `return` operation yields exactly one result, and `bind` combines all possible results, failing if there is a possibility to fail.

Now assume that we have a state (memory) type $'\mu$, and an access report type $'\rho$, which forms a monoid $(0,+)$. With this, we define our state-nondeterminism-error monad with access reports, just called M for brevity:

```
'x M ≡ 'μ ⇒ ('x × 'ρ × 'μ) neM
return_M x μ ≡ return_ne (x,0,μ)
bind_M m f μ ≡ (x1,r1,μ) ← m μ; (x2,r2,μ) ← f x1 μ; return_ne (x2,r1+r2,μ)
```

Here, `return` does not change the state, and reports no accesses (0), and `bind` sequentially composes the executions, threading through the state μ, and adding up the access reports r_1 and r_2.

Typically, the access report will contain read and written addresses, such that data races can be detected. Moreover, if parallel executions can allocate memory, we must detect those executions where the memory manager allocated the same block in both parallel strands. As we assume a thread safe memory manager, those *infeasible* executions can safely be ignored. Let *norace* $:: '\rho \Rightarrow '\rho \Rightarrow$ *bool* and *feasible* $:: '\rho \Rightarrow '\rho \Rightarrow$ *bool* be symmetric predicates, and let *combine* $:: ('\rho \times '\mu) \Rightarrow ('\rho \times '\mu) \Rightarrow ('\rho \times '\mu)$ be a commutative operator to compose two pairs of access reports and states. Then, we define a parallel composition operator for M:

```
(m1 || m2) μ ≡
  (x1,r1,μ1) ← m1 μ; (x2,r2,μ2) ← m2 μ;          – execute both strands
  assume feasible ρ1 ρ2;                          – ignore infeasible combinations
  assert norace ρ1 ρ2;                            – fail on data race
  return_ne ((x1,x2), combine (ρ1,μ1) (ρ2,μ2))    – combine results

assume P ≡ if P then return () else spec (λ_. False)
assert P ≡ if P then return () else fail
```

Here, we use `assume` to ignore infeasible executions, and `assert` to fail on data races. Note that, if one parallel strand fails, and the other parallel strand has no possible results `spec` $(\lambda\_.\ \mathit{False})$, the behaviour of the parallel composition is not clear. For this reason, we fix an invariant $\mathit{invar}_M :: ('\mu \Rightarrow ('x \times '\rho \times '\mu)\ \mathit{neM}) \Rightarrow \mathit{bool}$, which implies that every non-failing execution has at least one possible result. We define the actual type M as the subtype satisfying invar_M. Thus, we have to prove that every combinator and instruction of our semantics preserves the invariant, which is an important sanity check. As additional sanity check, we prove symmetry of parallel composition:

```
m1 || m2 = mswap (m2 || m1)    where    mswap m ≡ (x1,x2)←m; return (x2,x1)
```

2.2 Memory Model

Our memory model supports blocks of values, where values can be integers, structures, or pointers into a block:

```
datatype addr ≡ ADDR (bidx: nat) (idx: nat)
datatype ptr ≡ PTR_NULL | PTR_ADDR (the_addr: addr)
datatype val ≡ LL_INT lint | LL_STRUCT val list | LL_PTR ptr

datatype block ≡ FRESH | FREED | is_alloc: ALLOC (vals: val list)
typedef memory ≡ { μ :: nat ⇒ block. finite {b. μ b ≠ FRESH} }
```

A block is either fresh, freed, or allocated, and a memory is a mapping from block indexes to blocks, such that only finitely many blocks are not fresh. Every block's state transitions from fresh to allocated to freed. This avoids ever reusing the same block, and thus allows us to semantically detect use after free errors. Every program execution can only allocate finitely many blocks, such that we will never run out of fresh blocks[1]. An allocated block contains an array of values, modelled as a list. Thus, an address consists of a block number, and an index into the array.

To access and modify memory, we define the functions *valid*, *get*, and *put*:

```
valid μ (ADDR b i) ≡ is_alloc (μ b) ∧ i<|vals (μ b)|
get μ (ADDR b i) ≡ vals (μ b) ! i
put μ (ADDR b i) x ≡ μ(b := ALLOC ((vals (μ b))[i:=x]))
```

where $|xs|$ is the length of list xs, $xs!i$ returns the ith element of list xs, and $xs[i:=x]$ replaces the ith element of xs by x.

Note that our LLVM semantics does not support conversion of pointers to integers, nor comparison or difference of pointers to different blocks. This way, a program cannot see the internal representation of a pointer, and we can choose a simple abstract representation, while being faithful wrt. any actual memory manager implementation.

2.3 Access Reports

We now fix the state of the M-monad to be memory, and the access reports to be sets of read and written addresses, as well as sets of allocated and freed blocks:

```
acc ≡ ( r :: addr set; w :: addr set; a :: nat set; f :: nat set )
0 ≡ ( {},{},{},{} )
(r1,w1,a1,f1) + (r2,w2,a2,f2) ≡ ( r1∪ r2, w1∪ w2, a1∪ a2, f1∪ f2 )
```

Two parallel executions are feasible if they did not allocate the same block, and they have a data race if one strand accesses addresses or blocks modified by the other strand:

```
feasible (r1,w1,a1,f1) (r2,w2,a2,f2) ≡ a1 ∩ a2 = {}

norace (r1,w1,a1,f1) (r2,w2,a2,f2) ≡
  let m1 = w1 ∪ { ADDR b i. b ∈ a1 ∪ f1 } in
  let m2 = w2 ∪ { ADDR b i. b ∈ a2 ∪ f2 } in
    (r1 ∪ m1) ∩ m2 = {}   ∧   m1 ∩ (r2 ∪ m2) = {}
```

[1] If the actual system does run out of memory, we will terminate the program in a defined way.

The invariant for M states that blocks transition only from fresh to allocated to free, allocated blocks never change their size, and the access report matches the observable state change (*consistent*). It also states, that for each finite set of blocks B, there is an execution that does not allocate blocks from B. The latter is required to show that we always find feasible parallel executions:

$$\begin{array}{l} \mathit{invar}_M\ c \equiv \forall \mu\ P.\ c\ \mu = \texttt{spec}\ P \implies \\ \quad (\forall x\ \rho\ \mu'.\ P\ (x,\rho,\mu') \implies \mathit{consistent}\ \mu\ \rho\ \mu') \\ \wedge\ (\forall B.\ \mathit{finite}\ B \implies (\exists x\ \rho\ \mu'.\ P\ (x,\rho,\mu') \wedge \rho.a \cap B = \{\}\ )) \end{array}$$

The combine function joins the access reports and memories, preferring allocated over fresh, and freed over allocated memory. When joining two allocated blocks, the written addresses from the access report are used to join the blocks. We skip the rather technical definition of combine, and just state the relevant properties: Let $\rho_1{=}(r_1,w_1,a_1,f_1)$ and $\rho_2{=}(r_2,w_2,a_2,f_2)$ be feasible and race free access reports, and $\mu_1,\ \mu_2$ be memories that have evolved from a common memory μ, consistently with the access reports $\rho_1,\ \rho_2$. Let $(\rho',\mu') = \mathit{combine}\ (\rho_1,\mu_1)\ (\rho_2,\mu_2)$, and addr a valid address in μ'. Then

$$\begin{array}{ll} (1) & \mu'\ b = \mathit{FRESH} \longleftrightarrow \mu\ b = \mathit{FRESH} \wedge b \notin a_1 \cup a_2 \\ (2) & \mathit{is\_alloc}\ (\mu'\ b) \longleftrightarrow (\mathit{is\_alloc}\ (\mu\ b) \vee b \in a_1 \cup a_2) \wedge b \notin f_1 \cup f_2 \\ (3) & \mu'\ b = \mathit{FREED} \longleftrightarrow \mu\ b = \mathit{FREED} \vee b \in f_1 \cup f_2 \\ & \\ (4) & a \in w_1 \vee b \in a_1 \implies \mathit{get\_addr}\ \mu'\ a = \mathit{get\_addr}\ \mu_1\ a \\ (5) & a \in w_2 \vee b \in a_2 \implies \mathit{get\_addr}\ \mu'\ a = \mathit{get\_addr}\ \mu_2\ a \\ (6) & a \notin w_1 \cup w_2 \vee b \notin a_1 \cup a_2 \implies \mathit{get\_addr}\ \mu'\ a = \mathit{get\_addr}\ \mu\ a \end{array}$$

The properties (1)–(3) define the state of blocks in the combined memory: a fresh block in μ' was fresh already in μ, and has not been allocated (1); an allocated block was already allocated or has been allocated, but has not been freed (2); and a freed block was already freed, or has been freed (3). The properties (4)–(6) define the content: addresses written or allocated in the first or second execution get their content from μ_1 (4) or μ_2 (5) respectively. Addresses not written or allocated at all keep their original content (6).

2.4 LLVM Instructions

Based on the M-monad, we define shallowly embedded LLVM instructions. For most instructions, this is analogous to the sequential case [26]. The exceptions are memory allocation, which nondeterministically allocates some available block (the original formalization deterministically counted up the block indexes), and an instruction for parallel function call:

$$\mathit{llc\_par}\ f\ g\ a\ b \equiv f\ a\ ||\ g\ b$$

The code generator only accepts this, if f and g are constants (i.e., function names). It then generates some type-casting boilerplate, and a call to an external *parallel* function, which we implement using the Threading Building Blocks [36] library:

```
void parallel(void (*f1)(void*), void (*f2)(void*), void *x1, void *x2) {
  tbb::parallel_invoke([=]{f1(x1);}, [=]{f2(x2);}); }
```

I.e., the two functions *f1*(*x1*) and *f2*(*x2*) are called in parallel. The generated boilerplate code sets up *x1* and *x2* to point to both, the actual arguments and space for the results.

3 Parallel Separation Logic

In the previous section, we have defined a shallow embedding of LLVM programs into Isabelle/HOL. We now describe how to reason about these programs, using separation logic.

3.1 Separation Algebra

In order to reason about memory with separation logic, we define an abstraction function from the memory into a separation algebra [8]. Separation algebras formalize the intuition of combining disjoint parts of memory. They come with a *zero* (*0*) that describes the empty part, a *disjointness predicate* $a\#b$ describing that the parts a and b do not overlap, and a *disjoint union* $a+b$ that combines two disjoint parts. For the exact definition of a separation algebra, we refer to [8, 20]. We note that separation algebras naturally extend over functions and pairs, in a pointwise manner.

▶ **Example 1.** (Trivial Separation Algebra) The type α *option* = *None* | *Some* α forms a separation algebra with:

$$0 \equiv None \qquad a \mathbin{\#} b \equiv a{=}0 \vee b{=}0 \qquad a + 0 \equiv a \qquad 0 + b \equiv b$$

Intuitively, this separation algebra does not allow for combination of contents, except if one side is zero. While it is not very useful on its own, the trivial separation algebra is a useful building block for more complex separation algebras.

For our memory model, we define the following abstraction function:

$$\alpha :: memory \to (addr \to val\ option) \times (nat \to nat\ option)$$
$$\alpha\ \mu \equiv (\alpha_m\ \mu,\ \alpha_b\ \mu)$$

$$\alpha_m\ \mu\ addr \equiv \texttt{if}\ valid\ \mu\ addr\ \texttt{then}\ Some\ (get\ \mu\ addr)\ \texttt{else}\ 0$$
$$\alpha_b\ \mu\ b \equiv \texttt{if}\ is\_alloc\ (\mu\ b)\ \texttt{then}\ Some\ (|vals\ (\mu\ b)|)\ \texttt{else}\ 0$$

An abstract memory $\alpha\ \mu$ consists of two parts: $\alpha_m\ \mu$ is a map from addresses to the values stored there. It is used to reason about load and store operations. $\alpha_b\ \mu$ is a map from block indexes to the sizes of the corresponding blocks. It is used to ensure that one owns all addresses of a block when freeing it.

We continue to define a separation logic: assertions are predicates over separation algebra elements. The basic connectives are defined as follows:

$$false\ a \equiv False \qquad true\ a \equiv True \qquad \square\ a \equiv a{=}0$$
$$(P{*}Q)\ a \equiv \exists\ a_1\ a_2.\ a_1 \mathbin{\#} a_2 \wedge a = a_1 + a_2 \wedge P\ a_1 \wedge Q\ a_2$$

That is, the assertion *false* never holds and the assertion *true* holds for all abstract memories. The empty assertion $\square$ holds for the zero memory, and the separating conjunction $P{*}Q$ holds if the memory can be split into two disjoint parts, such that P holds for one, and Q holds for the other part. The lifting assertion $\uparrow\phi$ holds iff the Boolean value ϕ is true:

$$\uparrow\phi \equiv \texttt{if}\ \phi\ \texttt{then}\ \square\ \texttt{else}\ false$$

It is used to lift plain logical statements into separation logic assertions owning no memory. When clear from the context, we omit the $\uparrow$-symbol, and just mix plain statements with separation logic assertions.

3.2 Weakest Preconditions and Hoare Triples

We define a *weakest precondition* predicate directly via the semantics:

$$wp\ m\ Q\ \mu \equiv \texttt{case}\ m\ \mu\ \texttt{of spec}\ Q' \Rightarrow \forall x\ \rho\ \mu'.\ Q'\ (x,\rho,\mu')\ \Longrightarrow\ Q\ x\ \rho\ \mu' \mid \texttt{fail} \Rightarrow \mathit{False}$$

That is, $wp\ m\ Q\ \mu$ holds, iff program m run on memory μ does not fail, and all possible results (return value x, access report ρ, new memory μ') satisfy the *postcondition* Q.

To set up a verification condition generator based on separation logic, we standardize the postcondition: the reported memory accesses must be disjoint from some abstract memory *amf*, called the *frame*. We define the *weakest precondition with frame*:

$$wpf\ amf\ c\ Q\ \mu \equiv wp\ c\ (\lambda x\ \rho\ \mu'.\ Q\ x\ \mu' \wedge \mathit{disjoint}\ \rho\ amf)\ \mu$$

$$\begin{aligned}\mathit{disjoint}\ (r,w,a,f)\ (m,b) \equiv\ & (\forall addr.\ m\ addr \neq 0\ \Longrightarrow\ addr \notin r \cup w \wedge addr.bidx \notin f)\\ & \wedge (\forall i.\ b\ i \neq 0\ \Longrightarrow\ i \notin f)\end{aligned}$$

that is, when executed on memory μ, the program c does not fail, every return value x and new memory μ' satisfies Q, and no memory described by the frame *amf* is accessed.

Equipped with a weakest precondition with access restrictions, we define a Hoare-triple:

$$ABS\ amf\ P\ \mu \equiv \exists am.\ am\ \#\ amf \wedge \alpha\ \mu = am{+}amf \wedge P\ am$$

$$ht\ P\ c\ Q \equiv \forall \mu\ amf.\ ABS\ amf\ P\ \mu\ \Longrightarrow\ wpf\ amf\ c\ (\lambda x\ \mu'.\ ABS\ amf\ (Q\ x)\ \mu')\ \mu$$

The predicate $ABS\ amf\ P\ \mu$ specifies that the abstract memory $\alpha\ \mu$ can be split into a part am and the given frame *amf*, such that am satisfies the precondition P. A Hoare-triple $ht\ P\ c\ Q$ specifies that for all memories and frames for which the precondition holds ($ABS\ amf\ P\ \mu$), the program will succeed, not using any memory of the frame, and every result will satisfy the postcondition wrt. the original frame ($ABS\ amf\ (Q\ x)\ \mu'$).

3.3 Verification Condition Generator

The verification condition generator is implemented as a proof tactic that works on subgoals of the form:

$$ABS\ amf\ P\ \mu \wedge \ldots\ \Longrightarrow\ wpf\ amf\ c\ Q\ \mu$$

The tactic is guided by the syntax of the command c. Basic monad combinators are broken down using the following rules:

$$\begin{aligned}&Q\ r\ \mu\ \Longrightarrow\ wpf\ amf\ (\texttt{return}\ r)\ Q\ \mu\\ &wpf\ amf\ m\ (\lambda x.\ wpf\ amf\ (f\ x)\ Q)\ \mu\ \Longrightarrow\ wpf\ amf\ (\{x \leftarrow m;\ f\ x\})\ Q\ \mu\end{aligned}$$

For other instructions and user defined functions, the VCG expects a Hoare-triple to be already proved. It then uses the following rule:

$$\begin{aligned}&ht\ P\ c\ Q \wedge ABS\ amf\ P'\ \mu && \text{– match Hoare triple and current state}\\ &\wedge\ P' \vdash P{*}F && \text{– infer frame}\\ &\wedge\ (\textstyle\bigwedge r\ \mu.\ ABS\ amf\ (Q\ r * F)\ \mu\ \Longrightarrow\ Q'\ r\ \mu) && \text{– continue with postcondition}\\ &\Longrightarrow\ wpf\ amf\ c\ Q'\ \mu\end{aligned}$$

To process a command c, the first assumption is instantiated with the Hoare-triple for c, and the second assumption with the assertion P' for the current state. Then, a simple syntactic heuristics infers a frame F and proves that the current assertion P' entails the required precondition P and the frame. Finally, verification condition generation continues with the postcondition Q and the frame as current assertion.

3.4 Hoare-Triples for Instructions

To use the VCG to verify LLVM programs, we have to prove Hoare triples for the LLVM instructions. For parallel calls, we prove the well-known disjoint concurrency rule [33]:

$$ht\ P_1\ c_1\ Q_1 \wedge ht\ P_2\ c_2\ Q_2 \implies ht\ (P_1 * P_2)\ (par\ c_1\ c_2)\ (\lambda(r_1,r_2).\ Q_1\ r_1 * Q_2\ r_2)$$

That is, commands with disjoint preconditions can be executed in parallel.

For memory operations, we prove:

$$\models \{n{\neq}0\}\ ll\_malloc\ TYPE(\alpha)\ n\ \{\lambda p.\ range\ \{0..{<}n\}\ (\lambda\_.\ init)\ p * b\_tag\ n\ p\}$$
$$\models \{range\ \{0..{<}n\}\ xs\ p * b\_tag\ n\ p\}\ ll\_free\ p\ \{\lambda\_.\ \square\}$$
$$\models \{pto\ x\ p\}\ ll\_load\ p\ \{\lambda r.\ r{=}x * pto\ x\ p\}$$
$$\models \{pto\ y\ p\}\ ll\_store\ x\ p\ \{\lambda\_.\ pto\ x\ p\}$$

Here $b\_tag\ n\ p$ asserts that p points to the beginning of a block of size n, and $range\ I\ f\ p$ describes that for all $i \in I$, $p + i$ points to value $f\ i$. Intuitively, *ll_malloc* creates a block of size n, initialized with the default *init* value, and a tag. If one possesses both, the whole block and the tag, it can be deallocated by free. The rules for load and store are straightforward, where $pto\ x\ p$ describes that p points to value x.

4 Refinement for Parallel Programs

At this point, we have described a separation logic framework for parallel programs in LLVM. It is largely backwards compatible with the framework for sequential programs described in [26], such that we could easily port the algorithms formalized there to our new framework. The next step towards verifying complex programs is to set up a stepwise refinement framework. In this section we describe the refinement infrastructure of the Isabelle Refinement Framework, focusing on our changes to support parallel algorithms.

4.1 Abstract Programs

Abstract programs are shallowly embedded into the nondeterminism error monad $'a\ neM$ (cf. Section 2.1). They are purely functional, not modifying memory, or differentiating between sequential and parallel execution. We define a *refinement ordering* on *neM*:

$$\texttt{spec}\ P \le \texttt{spec}\ Q \equiv \forall x.\ P\ x \implies Q\ x \qquad \texttt{fail} \not\le \texttt{spec}\ Q \qquad m \le \texttt{fail}$$

Intuitively, $m_1 \le m_2$ means that m_1 returns fewer possible results than m_2, and may only fail if m_2 may fail. Note that $\le$ is a complete lattice, with top element `fail`.

We use refinement and assertions to specify that a program m satisfies a specification with precondition P and postcondition Q:

$$m \le \texttt{assert}\ P;\ \texttt{spec}\ x.\ Q\ x$$

If the precondition is false, the right hand side is `fail`, and the statement trivially holds. Otherwise, m cannot fail, and every possible result x of m must satisfy Q.

For a detailed description on using the *ne*-monad for stepwise refinement based program verification, we refer the reader to [30].

4.2 The Sepref Tool

The Sepref tool [23, 26] symbolically executes an abstract program in the *ne*-monad, keeping track of refinements for every abstract variable to a concrete representation, which may use pointers to dynamically allocated memory. During the symbolic execution, the tool synthesizes an imperative Isabelle-LLVM program, together with a refinement proof. The synthesis is automatic, but requires annotations to the abstract program.

The main concept of the Sepref tool is refinement between an abstract program c in the *ne*-monad, and a concrete program $c_\dagger$ in the M monad, as expressed by the *hnr*-predicate:

$$\begin{array}{l} hnr\ \Gamma\ c_\dagger\ \Gamma'\ R\ CP\ c \equiv \\ \quad c \neq \texttt{fail} \implies ht\ \Gamma\ c_\dagger\ (\lambda x_\dagger.\ \exists x.\ \Gamma' * R\ x\ x_\dagger\ * \uparrow(\texttt{return}\ x \leq c \wedge CP\ x_\dagger)) \end{array}$$

That is, either the abstract program c fails, or for a memory described by assertion Γ, the LLVM program $c_\dagger$ succeeds with $x_\dagger$, such that the new memory is described by $\Gamma' * R\ x\ x_\dagger$, for a possible result x of the abstract program c. Moreover, the predicate CP holds for the concrete result. Note that *hnr* trivially holds for a failing abstract program. This makes sense, as we prove that the abstract program does not fail anyway. Moreover it allows us to assume that assertions actually hold during the refinement proof:

$$(\ \phi \implies hnr\ \Gamma\ c_\dagger\ \Gamma'\ R\ CP\ c\ ) \implies hnr\ \Gamma\ c_\dagger\ \Gamma'\ R\ CP\ (\texttt{assert}\ \phi;\ c)$$

▶ **Example 2.** (Refinement of lists to arrays) We define abstract programs for indexing and updating a list:

$$lget\ xs\ i \equiv \texttt{assert}\ (i{<}|xs|);\ \texttt{return}\ xs!i \qquad lset\ xs\ i\ x \equiv \texttt{assert}\ (i{<}|xs|);\ \texttt{return}\ xs[i{:=}x]$$

These programs assert that the index is in bounds, and then return the accessed element ($xs!i$) or the updated list ($xs[i{:=}x]$) respectively. The following assertion links a pointer to a list of elements stored at the pointed-to location:

$$arr_A\ xs\ p = range\ \{0..{<}|xs|\}\ (\lambda i.\ xs!i)\ p$$

That is, for every $i < |xs|$, $p + i$ points to the ith element of xs. On arrays, indexing and updating of arrays is implemented by:

$$aget\ p\ i \equiv ll\_ofs\_ptr\ p\ i;\ ll\_load\ p \qquad aset\ p\ i\ x \equiv ll\_ofs\_ptr\ p\ i;\ ll\_store\ x\ p;\ \texttt{return}\ p$$

And the abstract and concrete programs are linked by the following refinement theorems:

$$\begin{array}{l} hnr\ (arr_A\ xs\ xs_\dagger\ *\ idx_A\ i\ i_\dagger)\ (aget\ xs_\dagger\ i_\dagger)\ (arr_A\ xs\ xs_\dagger * idx_A\ i\ i_\dagger)\ id_A\ (\lambda\_.\ True)\ (lget\ xs\ i) \\ hnr\ (arr_A\ xs\ xs_\dagger\ *\ idx_A\ i\ i_\dagger)\ (aset\ xs_\dagger\ i_\dagger\ x)\ (idx_A\ i\ i_\dagger)\ arr_A\ (\lambda r.\ r{=}xs_\dagger)\ (lset\ xs\ i\ x) \end{array}$$

That is, if the list xs is refined by array $xs_\dagger$, and the natural number i is refined by the fixed-width[2] word $i_\dagger$ ($idx_A\ i\ i_\dagger$), the *aget* operation will return the same result as the *lget* operation (id_A). The resulting memory will still contain the original array. Note that there is no explicit precondition that the array access is in bounds, as this follows already from the assertion in the abstract *lget* operation. The *aset* operation will return a pointer to an array that refines the updated list returned by *lset*. As the array is updated in place, the original refinement of the array is no longer valid. Moreover, the returned pointer r will be the same as the argument pointer $xs_\dagger$. This information is important for refining to parallel programs on disjoint parts of an array (cf. Section 4.3).

[2] We use Isabelle's word library here, which encodes the actual width as a type variable, such that our functions work with any bit width. For code generation, we will fix the width to 64 bit.

Given refinement assertions for the parameters, and *hnr*-rules for all operations in a program, the Sepref tool automatically synthesizes an LLVM program from an abstract *neM* program. The tool tries to automatically discharge additional proof obligations, typically arising from translating arithmetic operations from unbounded numbers to fixed width numbers. Where automatic proof fails, the user has to add assertions to the abstract program to help the proof. The main difference of our tool wrt. the existing Sepref tool [26] is the additional condition (*CP*) on the concrete result, which is used to track pointer equalities. We have added a heuristics to automatically synthesize and discharge these equalities.

4.3 Array Splitting

An important concept for parallel programs is to concurrently operate on disjoint parts of the memory, e.g., different slices of the same array. However, abstractly, arrays are just lists. They are updated by returning a new list, and there is no way to express that the new list is stored at the same address as the old list. Nevertheless, in order to refine a program that updates two disjoint slices of a list to one that updates disjoint parts of the array in place, we need to know that the result is stored in the same array as the input. This is handled by the *CP* argument to *hnr*. To indicate that operations shall be refined to disjoint parts of the same array, we introduce the combinator `with_split` for abstract programs:

$$
\begin{array}{l}
\texttt{with\_split}\ i\ xs\ f \equiv \\
\quad \texttt{assert}\ (i < |xs|); \\
\quad (xs_1, xs_2) \leftarrow f\ (take\ i\ xs)\ (drop\ i\ xs); \\
\quad \texttt{assert}\ (|xs_1| = i \wedge |xs_2| = |xs| - i); \\
\quad \texttt{return}\ (xs_1 @ xs_2)
\end{array}
$$

Abstractly, this is an annotation that is inlined when proving the abstract program correct. However, Sepref will translate it to the concrete combinator *awith_split*:

$$
\begin{array}{l}
awith\_split\ i\ xs_\dagger\ f_\dagger \equiv xs_{\dagger 2} \leftarrow ll\_ofs\_ptr\ xs_\dagger\ i;\ f_\dagger\ xs_\dagger\ xs_{\dagger 2};\ \texttt{return}\ xs_\dagger \\
\\
hnr\ (arr_A\ xs_1\ xs_{\dagger 1} * arr_A\ xs_2\ xs_{\dagger 2})\ (f_\dagger\ xs_{\dagger 1}\ xs_{\dagger 2})\ \square \\
\quad (arr_A \times arr_A)\ (\lambda(xs_{\dagger 1}', xs_{\dagger 2}').\ xs_{\dagger 1}' = xs_{\dagger 1} \wedge xs_{\dagger 2}' = xs_{\dagger 2}) \\
\quad (f\ xs_1\ xs_2) \\
\Longrightarrow \\
hnr\ (arr_A\ xs\ xs_\dagger * idx_A\ i\ i_\dagger)\ (awith\_split\ i_\dagger\ xs_\dagger\ f_\dagger) \\
\quad (idx_A\ i\ i_\dagger)\ (\lambda xs\ xs_\dagger.\ arr_A\ xs\ xs_\dagger)\ (\lambda xs_\dagger'.\ xs_\dagger' = xs_\dagger) \\
\quad (\texttt{with\_split}\ i\ xs\ f)
\end{array}
$$

The refinement of the function f to $f_\dagger$ requires an additional proof that the returned pointers are equal to the argument pointers ($xs_{\dagger 1}' = xs_{\dagger 1} \wedge xs_{\dagger 2}' = xs_{\dagger 2}$). Sepref tries to prove that automatically, using a simple heuristics.

4.4 Refinement to Parallel Execution

The purely functional abstract programs have no notion of parallel execution. To indicate that refinement to parallel execution is desired, we define an abstract annotation `npar`:

$$
\begin{array}{l}
\texttt{npar}\ f\ g\ a\ b \equiv x \leftarrow f\ a;\ y \leftarrow g\ b;\ \texttt{return}\ (x,y) \\
\\
\quad hnr\ Ax\ (f_\dagger\ x_\dagger)\ Ax'\ Rx\ CP_1\ (f\ x) \wedge hnr\ Ay\ (g_\dagger\ y_\dagger)\ Ay'\ Ry\ CP_2\ (g\ y) \\
\Longrightarrow \\
\quad hnr\ (Ax * Ay)\ (llc\_par\ f_\dagger\ g_\dagger\ x_\dagger\ y_\dagger)\ (Ax' * Ay')\ (Rx \times Ry) \\
\quad (\lambda(x_\dagger', y_\dagger').\ CP_1\ x_\dagger' \wedge CP_2\ y_\dagger')\ (\texttt{npar}\ f\ g\ x\ y)
\end{array}
$$

This rule can be used to automatically parallelize any (independent) abstract computations. For convenience, we also define `nseq`. Abstractly, it's the same as `npar`, but Sepref translates it to sequential execution.

5 A Parallel Sorting Algorithm

To test the usability of our framework, we verify a parallel sorting algorithm. We start with the abstract specification of an algorithm that sorts a list:

```
sort_spec xs = spec xs'. mset xs'=mset xs ∧ sorted xs
```

That is, we return a sorted permutation of the original list. Note that this is a standard specification of sorting in Isabelle. Reusing the existing development of an abstract introsort algorithm [27], we easily prove with a few refinement steps that the following abstract algorithm implements *sort_spec*:

```
1  psort xs n ≡ assert n=|xs|; if n≤1 then return xs else psort_aux xs n (log2 n * 2)
2
3  psort_aux xs n d ≡
4    assert n=|xs|
5    if d=0 ∨ n<100000 then sort_spec xs
6    else
7      (xs,m) ← partition_spec xs;
8      let bad = m<n div 8 ∨ (n−m < n div 8)
9      (_,xs) ← with_split m xs (λxs1 xs2.
10       if bad then nseq psort_aux psort_aux (xs1,m,d−1) (xs2,n−m,d−1)
11       else npar psort_aux psort_aux (xs1,m,d−1) (xs2,n−m,d−1)
12     );
13     return xs
14
15 lemma psort xs |xs| ≤ sort_spec xs
```

This algorithm is derived from the well-known quicksort and introsort algorithms [32]: like quicksort, it partitions the list (line 7), and then recursively sorts the partitions in parallel (l. 11). Like introsort, when the recursion gets too deep, or the list too short, we fall back to some (not yet specified) sequential sorting algorithm (l. 5). Similarly, when the partitioning is very unbalanced (l. 8), we sort the partitions sequentially (l. 10). These optimizations aim at not spawning threads for small sorting tasks, where the overhead of thread creation outweighs the advantages of parallel execution. A more technical aspect is the extra parameter n that we introduced for the list length. Thus, we can refine the list to just a pointer to an array, and still access its length[3].

5.1 Implementation and Correctness Theorem

Next, we have to provide implementations for the fallback *sort_spec*, and for *partition_spec*. These implementations must be proved to be in-place, i.e., return a pointer to the same array. It was straightforward to amend our existing formalization of *pdqsort* [27] with the in-place proofs: once we had amended the refinement statements, and bug-fixed the pointer equality proving heuristics that we added to Sepref, the proofs were automatic.

[3] Alternatively, we could refine a list to a pair of array pointer and length.

Given the implementations of *sort_spec* and *partition_spec*, the Sepref tool generates an LLVM program $psort_\dagger$ from the abstract *psort*, and proves a corresponding refinement lemma:

$$hnr\ (arr_A\ xs\ xs_\dagger * idx_A\ n\ n_\dagger)\ (psort_\dagger\ xs_\dagger\ n_\dagger)\ (idx_A\ n\ n_\dagger)\ arr_A\ (\lambda r.\ r = xs_\dagger)\ (psort\ xs\ n)$$

Combining this with the correctness lemma of the abstract *psort* algorithm, and unfolding the definition of *hnr*, we prove the following Hoare-triple for our final implementation:

$$\begin{array}{l} ht\ (arr_A\ xs\ xs_\dagger * idx_A\ n\ n_\dagger * n = |xs|) \\ \quad (psort_\dagger\ xs_\dagger\ n_\dagger) \\ \quad (\lambda r.\ r{=}xs_\dagger * \exists\ xs'.\ arr_A\ xs'\ xs_\dagger * sorted\ xs' * mset\ xs' = mset\ xs) \end{array}$$

That is, for a pointer $xs_\dagger$ to an array, whose contents are described by list *xs* (arr_A), and a fixed-size word $n_\dagger$ representing the natural number *n* (idx_A), which must be the number of elements in the list *xs*, our sorting algorithm returns the original pointer $xs_\dagger$, and the array contents are now *xs*′, which is sorted and a permutation of *xs*. Note that this statement uses our semantically defined Hoare triples (cf. Section 3.2). In particular, its correctness does not depend on the refinement steps, the Sepref tool, or the VCG.

5.2 A Sampling Partitioner

While we could simply re-use the existing partitioning algorithm from the pdqsort formalization, which uses a pseudomedian of nine pivot selection, we observe that the quality of the pivot is particularly important for a balanced parallelization. Moreover, the partitioning in the *psort_aux* procedure is only done for arrays above a quite big size threshold. Thus, we can invest a little more work to find a good pivot, which is still negligible compared to the cost of sorting the resulting partitions. We choose a sampling approach, using the median of 64 equidistant samples as pivot. The highly optimized partitioning algorithms that we use swap the pivot to the front of the partition, such that we need to determine its index, rather than just its value. We simply use quicksort to find the median[4]:

$$sample\ xs \equiv is \leftarrow equidist\ |xs|\ 64;\ is \leftarrow sort\_wrt\ (\lambda i\ j.\ xs!i < xs!j)\ is;\ \texttt{return}\ (is!32)$$

Proving that this algorithm finds a valid pivot index is straightforward. More challenging is to refine it to purely imperative LLVM code, which does not support closures like $\lambda i\ j.\ xs!i < xs!j$.

We resolve such closures over the comparison function manually: using Isabelle's locale mechanism [19], we parametrize over the comparison function. Moreover, we thread through an extra parameter for the data captured by the closure:

$$\begin{array}{l} \textbf{locale}\ pcmp = \\ \quad \textbf{fixes}\ lt :: 'p \Rightarrow 'e \Rightarrow 'e \Rightarrow bool\ \textbf{and}\ lt_\dagger :: 'p_\dagger \Rightarrow 'e_\dagger \Rightarrow 'e_\dagger \Rightarrow bool \\ \quad\quad \textbf{and}\ par_A :: 'p \Rightarrow 'p_\dagger \Rightarrow assn\ \textbf{and}\ elem_A :: 'e \Rightarrow 'e_\dagger \Rightarrow assn \\ \quad \textbf{assumes}\ \forall p.\ weak\_ordering\ (lt\ p) \\ \quad \textbf{assumes}\ hnr\ (par_A\ p\ pi * elem_A\ a\ ai * elem_A\ b\ bi)\ (lt_\dagger\ pi\ ai\ bi) \\ \quad\quad (par_A\ p\ pi * elem_A\ a\ ai * elem_A\ b\ bi)\ \ (bool_A)\ \ (\lambda\_.\ True)\ (lt\ p\ a\ b) \end{array}$$

[4] We leave verification of efficient median algorithms, e.g., quickselect, to future work. Note that the overhead of sorting 64 elements is negligible compared to the large partition that has to be sorted.

This defines a context in which we have an abstract compare function *lt* for the abstract elements of type *'e*. It takes an extra parameter of type *'p* (e.g. the list *xs*), and forms a weak ordering[5]. Note that the strict compare function *lt* also induces a non-strict version *le p a b* ≡ ¬*lt p b a*. Moreover, we have a concrete implementation $lt_\dagger$ of the compare function, wrt. the refinement assertions par_A for the parameter and $elem_A$ for the elements.

Our sorting algorithm is developed and verified in the context of this locale (to avoid confusion, our presentation has, up to now, just used <, ≤, and *sorted* instead of *lt p*, *le p*, and *sorted_wrt* (*le p*)). To get a sorting algorithm for an actual compare function, we have to instantiate the locale, providing an abstract and concrete compare function, along with a proof that the abstract function is a weak ordering, and the concrete function refines the abstract one. For our example of sorting indexes into an array, where the array elements are, themselves, compared by a parametrized function *lt*, we get:

```
interpretation idx: pcmp lt_idx lt_idx† (par_A × arr_A) idx_A   ⟨proof⟩

lt_idx (p,xs) i j ≡ lt p (xs!i) (xs!j)
lt_idx† (p†,xs†) i† j† ≡ x†←aget xs† i†; y†←aget xs† j†; lt† p† x† y†
```

this yields sorting algorithms for sorting indexes, taking an extra parameter for the array to index into. For our sampling application, we use *idx.introsort xs*.

5.3 Code Generation

Finally, we instantiate the sorting algorithms to sort unsigned integers and strings:

```
interpretation unat: pcmp (λ_. <) (λ_. ll_icmp_ult) unat_A^64 ⟨proof⟩
interpretation str: pcmp (λ_. <) (λ_. strcmp) str_A^64 ⟨proof⟩
```

This yields implementations $unat.psort_\dagger$ and $str.psort_\dagger$, and automatically proves instantiated versions of the correctness theorems.

In a last step, we use our code generator to generate actual LLVM text, as well as a C header file with the signatures of the generated functions[6]:

```
export_llvm
  unat.psort† is uint64_t* psort(uint64_t*, int64_t)
  str.psort† is llstring* str_psort(llstring*, int64_t)
  defines typedef struct {int64_t size; struct {int64_t capacity; char *data;};} llstring;
  file psort.ll
```

This checks that the specified C signatures are compatible with the actual types, and then generates *psort.ll* and *psort.h*, which can be used in a standard C/C++ toolchain.

5.4 Benchmarks

We have benchmarked our verified sorting algorithm against a direct implementation of the same algorithm in C++. The result was that both implementations have the same runtime, up to some minor noise. This indicates that there is no systemic slowdown: algorithms verified with our framework run as fast as their unverified counterparts implemented in C++.

[5] A weak ordering is induced by a mapping of the elements into a total ordering. It is the standard prerequisite for sorting algorithms in C++ [17].

[6] For technical reasons, we represent the array size as non-negative signed integer, thus the C signature uses *int64_t*. Moreover, we use a string implementation based on dynamic arrays, rather than C's zero terminated strings.

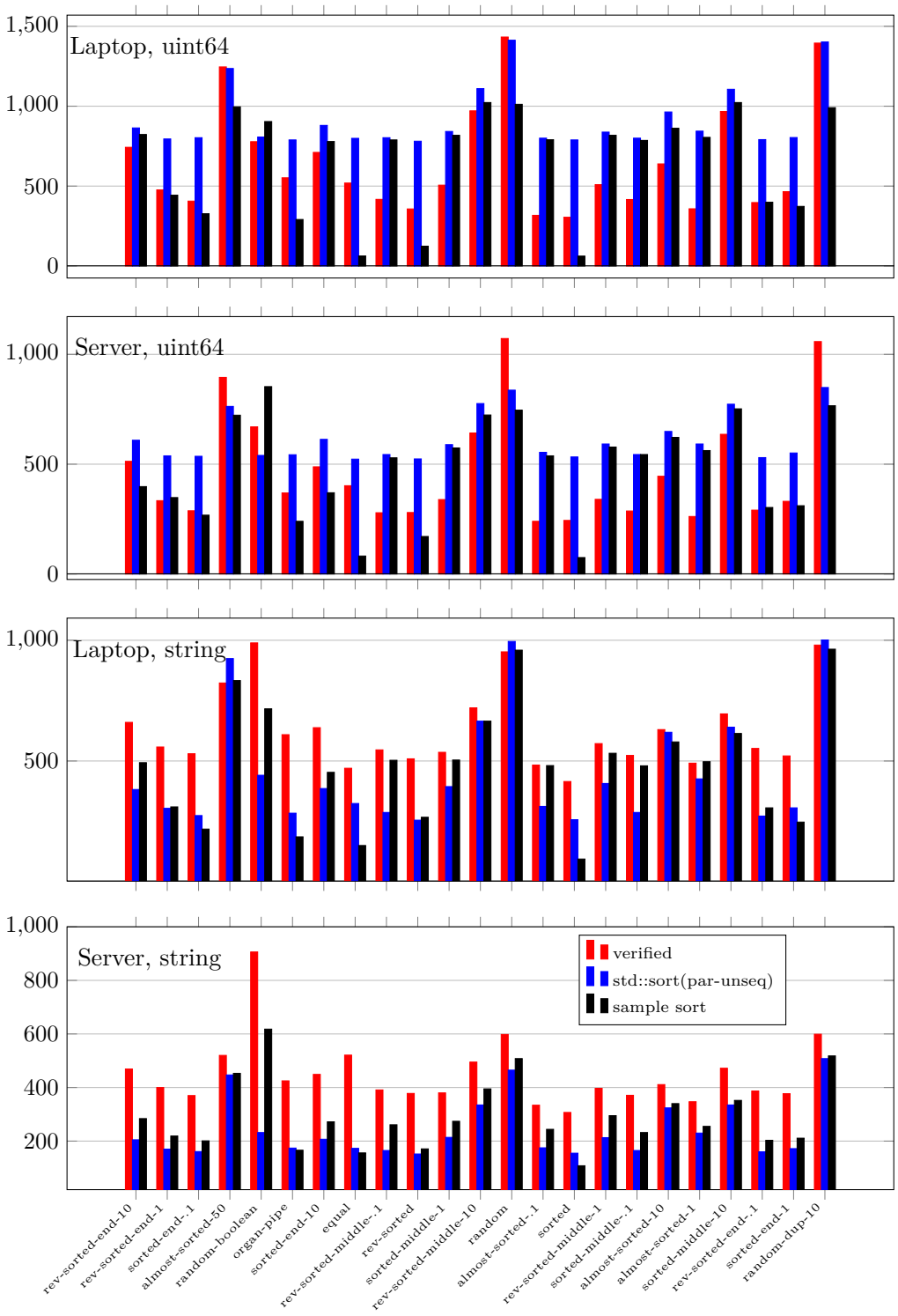


Figure 2 Runtimes in milliseconds for sorting various distributions of unsigned 64 bit integers and strings with our verified parallel sorting algorithm, C++'s standard parallel sorting algorithm, and Boost's parallel sample sort algorithm. The experiments were performed on a server machine with 22 AMD Opteron 6176 cores and 128GiB of RAM, and a laptop with a 6 core (12 threads) i7-10750H CPU and 32GiB of RAM.

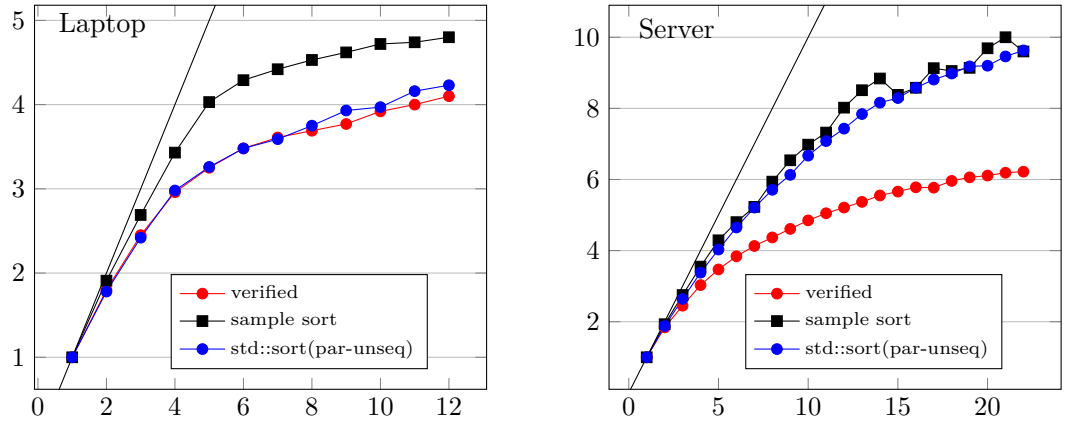


Figure 3 Speedup of the various implementations, for sorting unsigned 64 bit integers with a random distribution, on a server with 22 AMD Opteron 6176 cores and 128GiB of RAM, and a laptop with a 6 core (12 threads) i7-10750H CPU and 32GiB of RAM. The x axis ranges over the number of cores, and the y-axis gives the speedup wrt. the same implementation run on only one core. The thin black lines indicate linear speedup.

We also benchmarked against the state-of-the-art implementations *std::sort* with execution policy *par_unseq* from the GNU C++ standard library [12], and *sample_sort* from the Boost C++ libraries [4, 5]. We have benchmarked the algorithm on two different machines, and various input distributions. The results are shown in Figure 2. While our verified algorithm is clearly competitive for integer sorting on the less parallel laptop machine, it's slightly less efficient for sorting strings on the highly parallel server machine. Nevertheless, we believe that our verified implementation is already useful in practice, and leave further optimizations to future work.

Finally, we measured the speedup that the implementations achieve for a certain number of cores. The results are displayed in Figure 3. While the speedup on the moderately parallel laptop is comparable to the one of the C++ standard library, our implementation achieves lower speedups than the state-of-the-art on the highly parallel server. Again, we leave further optimizations to future work.

6 Conclusions

We have presented a stepwise refinement approach to verify total correctness of efficient parallel algorithms. Our approach targets LLVM as back end, and there is no systemic efficiency loss in our approach when compared to unverified algorithms implemented in C++.

The trusted code base of our approach is relatively small: apart from Isabelle's inference kernel, it contains our shallow embedding of a small fragment of the LLVM semantics, and the code generator. All other tools that we used, e.g., our Hoare logic, Sepref tool, and Refinement Framework for abstract programs, ultimately prove a correctness theorem that only depends on our shallowly embedded semantics.

As a case study, we have implemented a parallel sorting algorithm. It uses an existing verified sequential pdqsort algorithm as a building block, and is competitive with state-of-the-art parallel sorting algorithms, at least on moderately parallel hardware.

The main idea of our parallel extension is to shallowly embed the semantics of a parallel combinator into a sequential semantics, by making the semantics report the accessed memory locations, and fail if there is a potential data race. We only needed to change the lower

levels of our existing framework for sequential LLVM [26]. Higher-level tools like the VCG and Sepref remained largely unchanged and backwards compatible. This greatly simplified reusing of existing verification projects, like the sequential pdqsort algorithm [27].

6.1 Related Work

While there is extensive work on parallel sorting algorithm (e.g. [9, 1]), there seems to be almost no work on their formal verification. The only work we are aware of is a distributed merge sort algorithm [16], for which "no effort has been made to make it efficient"[16, Sec. 2], nor any executable code has been generated or benchmarked. Another verification [34] uses the VerCors deductive verifier to prove the permutation property ($mset\ xs' = mset\ xs$) of odd-even transposition sort [13], but neither the sortedness property nor termination.

Concurrent separation logic is used by many verification tools such as VerCors [3], and also formalized in proof assistants, for example in the VST [37] and IRIS [18] projects for Coq [2]. These formalizations contain elaborate concepts to reason about communication between threads via shared memory, and are typically used to verify partial correctness of subtle concurrent algorithms (e.g. [31]). Reasoning about total correctness is more complicated in the step-indexed separation logic provided by IRIS, and currently only supported for sequential programs [35]. Our approach is less expressive, but naturally supports total correctness, and is already sufficient for many practically relevant parallel algorithms like sorting, matrix-multiplication, or parallel algorithms from the C++ STL.

6.2 Future Work

An obvious next step is to implement a fractional separation logic [6], to reason about parallel threads that share read-only memory. While our semantics already supports shared read-only memory, our separation logic does not. We believe that implementing a fractional separation logic will be straightforward, and mainly pose technical issues for automatic frame inference.

Another obvious next step is to verify a state-of-the-art parallel sorting algorithm, like Boost's sample sort. Like our current algorithm, sample sort does not require advanced synchronization concepts, and can be implemented only with a parallel combinator.

Finally, the Sepref framework has recently been extended to reason about complexity of (sequential) LLVM programs [14, 15]. This line of work could be combined with our parallel extension, to verify the complexity (e.g. work and span) of parallel algorithms.

Extending our approach towards more advanced synchronization like locks or atomic operations may be possible: instead of accessed memory addresses, a thread could report a set of possible traces, which are checked for race-freedom and then combined.

Finally, our framework currently targets multicore CPUs. Another important architecture are general purpose GPUs. As LLVM is also available for GPUs, porting our framework to this architecture should be possible. We even expect that barrier synchronization, which is important in the GPU context, can be integrated into our approach.

References

1 Mikhail Asiatici, Damian Maiorano, and Paolo Ienne. How many cpu cores is an fpga worth? lessons learned from accelerating string sorting on a cpu-fpga system. *Journal of Signal Processing Systems*, pages 1–13, 2021.

2 Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010.

3 Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. The vercors tool set: Verification of parallel and concurrent software. In Nadia Polikarpova and Steve Schneider, editors, *Integrated Formal Methods*, pages 102–110, Cham, 2017. Springer International Publishing.

4 Boost C++ libraries. URL: `https://www.boost.org/`.

5 Boost C++ libraries sorting algorithms. URL: `https://www.boost.org/doc/libs/1_77_0/libs/sort/doc/html/index.html`.

6 Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 259–270, New York, NY, USA, 2005. ACM. `doi:10.1145/1040305.1040327`.

7 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`.

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 Jatin Chhugani, Anthony D Nguyen, Victor W Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *Proceedings of the VLDB Endowment*, 1(2):1313–1324, 2008.

10 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.

11 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.

12 The GNU C++ library 3.4.28. URL: `https://gcc.gnu.org/onlinedocs/libstdc++/`.

13 A. Nico Habermann. Parallel neighbor-sort, June 1972. `doi:10.1184/R1/6608258.v1`.

14 Maximilian P. L. Haslbeck and Peter Lammich. For a few dollars more - verified fine-grained algorithm analysis down to LLVM. *TOPLAS, S.I. ESOP'21*, 2021. to appear.

15 Maximilian P. L. Haslbeck and Peter Lammich. For a few dollars more - verified fine-grained algorithm analysis down to LLVM. In Nobuko Yoshida, editor, *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, volume 12648 of *Lecture Notes in Computer Science*, pages 292–319. Springer, 2021. `doi:10.1007/978-3-030-72019-3_11`.

16 Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris: Session-type based reasoning in separation logic. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. `doi:10.1145/3371074`.

17 Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley Professional, 2nd edition, 2012.

18 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018. `doi:10.1017/S0956796818000151`.

19 Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. Locales a sectioning concept for isabelle. In Yves Bertot, Gilles Dowek, Laurent Théry, André Hirschowitz, and Christine Paulin, editors, *Theorem Proving in Higher Order Logics*, pages 149–165, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

20 Gerwin Klein, Rafal Kolanski, and Andrew Boyton. Mechanised separation algebra. In *ITP*, pages 332–337. Springer, August 2012.

21 Peter Lammich. Automatic data refinement. In *ITP*, volume 7998 of *LNCS*, pages 84–99. Springer, 2013.

22 Peter Lammich. Verified efficient implementation of gabow's strongly connected component algorithm. In *International Conference on Interactive Theorem Proving*, pages 325–340. Springer, 2014.

23 Peter Lammich. Refinement to Imperative/HOL. In *ITP*, volume 9236 of *LNCS*, pages 253–269. Springer, 2015.

24 Peter Lammich. Efficient verified (UN)SAT certificate checking. In *Proc. of CADE*. Springer, 2017.

25 Peter Lammich. The GRAT tool chain - efficient (UN)SAT certificate checking with formal correctness guarantees. In *SAT*, pages 457–463, 2017.

26 Peter Lammich. Generating Verified LLVM from Isabelle/HOL. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:19, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ITP.2019.22.

27 Peter Lammich. Efficient verified implementation of introsort and pdqsort. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 307–323. Springer, 2020. doi:10.1007/978-3-030-51054-1_18.

28 Peter Lammich and S. Reza Sefidgar. Formalizing the Edmonds-Karp algorithm. In *Proc. of ITP*, pages 219–234, 2016.

29 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.

30 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.

31 Glen Mével and Jacques-Henri Jourdan. Formal verification of a concurrent bounded queue in a weak memory model. *Proc. ACM Program. Lang.*, 5(ICFP), August 2021. doi:10.1145/3473571.

32 DAVID R. MUSSER. Introspective sorting and selection algorithms. *Software: Practice and Experience*, 27(8):983–993, 1997. doi:10.1002/(SICI)1097-024X(199708)27:8<983::AID-SPE117>3.0.CO;2-\#.

33 Peter W. O'Hearn. Resources, concurrency and local reasoning. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004 - Concurrency Theory*, pages 49–67, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

34 Mohsen Safari and Marieke Huisman. A generic approach to the verification of the permutation property of sequential and parallel swap-based sorting algorithms. In *International Conference on Integrated Formal Methods*, pages 257–275. Springer, 2020.

35 Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. Transfinite iris: Resolving an existential dilemma of step-indexed separation logic. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 80–95, 2021.

36 Intel oneapi threading building blocks. URL: https://software.intel.com/en-us/intel-tbb.

37 Verified software toolchain project web page. URL: https://vst.cs.princeton.edu/.

38 Simon Wimmer and Peter Lammich. Verified model checking of timed automata. In *TACAS 2018*, pages 61–78, 2018.

Proof Pearl: Formalizing Spreads and Packings of the Smallest Projective Space PG(3,2) Using the Coq Proof Assistant

Nicolas Magaud
Lab. ICube UMR 7357 CNRS Université de Strasbourg, France

Abstract

We formally implement the smallest three-dimensional projective space PG(3,2) in the Coq proof assistant. This projective space features 15 points and 35 lines, related by an incidence relation. We define points and lines as two plain datatypes (one with 15 constructors for points, and one with 35 constructors for lines) and the incidence relation as a boolean function, instead of using the well-known coordinate-based approach relying on $GF(2)^4$. We prove that this implementation actually verifies all the usual properties of three-dimensional projective spaces. We then use an oracle to compute some characteristic subsets of objects of PG(3,2), namely spreads and packings. We formally verify that these computed objects exactly correspond to the spreads and packings of PG(3,2). For spreads, this means identifying 56 specific sets of 5 lines among 360 360 ($= 15 \times 14 \times 13 \times 12 \times 11$) possible ones. We then classify them, showing that the 56 spreads of PG(3,2) are all isomorphic whereas the 240 packings of PG(3,2) can be classified into two distinct classes of 120 elements. Proving these results requires partially automating the generation of some large specification files as well as some even larger proof scripts. Overall, this work can be viewed as an example of a large-scale combination of interactive and automated specifications and proofs. It is also a first step towards formalizing projective spaces of higher dimension, e.g. PG(4,2), or larger order, e.g. PG(3,3).

2012 ACM Subject Classification Theory of computation → Automated reasoning; Theory of computation → Logic and verification

Keywords and phrases Coq, projective geometry, finite models, spreads, packings, PG(3, 2)

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.25

Supplementary Material *Software (Source Code)*: `https://github.com/magaud/PG3q`
archived at `swh:1:dir:f4ea06977d8a030690dddc924ccaa0b7590831ff`

Acknowledgements We would like to thank Pascal Schreck for introducing us to the domain of (finite) projective geometry and the anonymous reviewers for their careful reading and suggestions.

1 Introduction

Projective incidence geometry [9, 6] is one of the simplest description of geometry, where only points and lines as well as their incidence properties are considered. In addition, in such a setting, we assume that two coplanar lines always meet. There exist some finite and infinite models of projective incidence geometry. Finite projective spaces are usually built from finite (Galois) fields of cardinality n denoted $GF(n)$ via a homogeneous coordinate system. Finite projective spaces arising from $GF(n)$ are denoted by $PG(d, n)$ where d is the dimension of the space and n the order of the underlying field. Several finite models are related to interesting mathematical puzzles and sometimes have practical and enjoyable applications. This is the case for the finite projective plane PG(2,7), which was used to design the card game Dobble[1]. In this game, players must identify a symbol which appears on both their card and their opponent's card. As the card desk (almost exactly) implements

[1] `https://en.wikipedia.org/wiki/Dobble`

Table 1 Numbers of points, lines and points per line depending on the dimension and the order of projective planes and spaces.

	# points	# lines	# points per line
$PG(2,2)$	7	7	3
$PG(2,3)$	13	13	4
$PG(2,5)$	31	31	6
$PG(2,n)$	n^2+n+1	n^2+n+1	$n+1$
$PG(3,2)$	15	35	3
$PG(3,3)$	40	130	4
$PG(3,4)$	85	357	5
$PG(3,n)$	$(n^2+1)(n+1)$	$(n^2+n+1)(n^2+1)$	$n+1$

the projective plane PG(2,7), given two cards (=lines), there always exists a symbol (=point) which belongs to both cards. In a three-dimensional setting, the smallest projective space PG(3,2) can be used to find some solutions to an old combinatorial problem: *Kirkman's schoolgirl problem* [7], which is stated as follows: *Fifteen young ladies in a school walk out three abreast for seven days in succession: it is required to arrange them daily, so that no two shall walk twice abreast.* As noted by Hirschfeld in [14, page 75], some solutions to this problem correspond to some packings of PG(3,2), which are one of the substructures of PG(3,2) that we study in this article.

Finite projective spaces have been studied extensively from a mathematical point of view (see e.g [14]). Recently [4], we started studying small finite projective planes/spaces from a computer science perspective. We formalized usual projective planes such as PG(2,2), PG(2,3) or PG(2,5) as well as the smallest projective space PG(3,2) using the Coq proof assistant [8, 2]. We especially focused on proving that the synthetic axioms for projective geometry hold in these models. In this paper, we follow up on experiments carried out recently [16] and we formally describe some of the characteristic subsets of PG(3,2), namely spreads of lines and packings of spreads as well as their properties.

In a three-dimensional setting, the number of points and lines increase rapidly with the order, as shown in Table 1. Thus we need to design extremely efficient proof techniques for PG(3,2) if we want our approach to be scalable to projective spaces of higher dimension or larger order. The whole Coq formalization is available online and can be retrieved at: `https://github.com/magaud/PG3q`[2]. Pointers to specific parts of the development are given throughout the document. Visual representations of the smallest projective space PG(3,2) can be retrieved from `https://demonstrations.wolfram.com/15PointProjectiveSpace/`. For illustration purposes, we reproduce a figure taken from wikipedia[3], which presents PG(3,2) as a tetrahedron (see Fig. 1).

This paper is organized as follows. In Sect. 2, we show how to formally describe PG(3,2) in Coq using plain inductive types. In Sect. 3, we define the notions of collineations, spreads and packings in the setting of PG(3,2). In Sect. 4, we characterize all the spreads of PG(3,2) and show that they are all isomorphic. In Sect. 5, we characterize all the packings of PG(3,2) and then classify them into two distinct classes. In Sect. 6, we present some proof engineering techniques and suggest some additional optimizations to make the proof development smaller and easier to compile. Finally, in Sect. 7, we draw some conclusions and outline how this work can be extended to projective spaces of higher dimension or larger order.

[2] Be aware that compiling all the .v files of this development requires about 13 hours on a standard PC.
[3] `https://en.wikipedia.org/wiki/PG(3,2)`

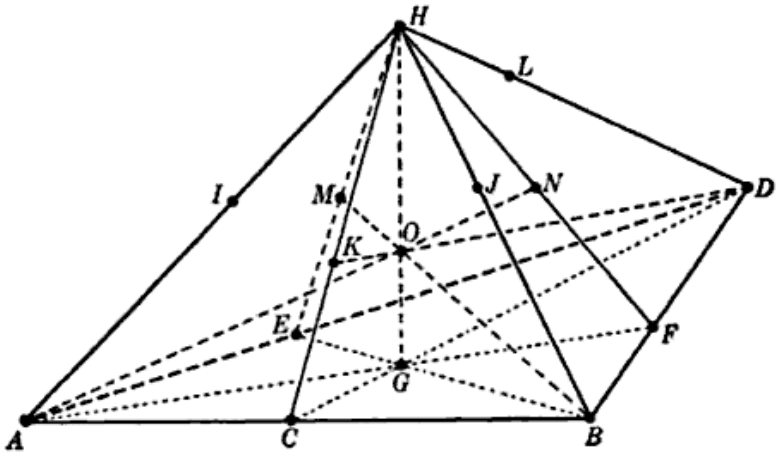


Figure 1 The smallest projective space PG(3,2), represented as a tetrahedron.

2 Formal Description of the Projective Space PG(3,2) in Coq

We first present an abstract interface (a Coq module) to describe what a projective space is. We also propose an implementation of PG(3,2), relying on plain inductive datatypes for points and lines. We then show that all axioms of the projective space are verified by this implementation.

2.1 Specification of Projective Spaces

A three-dimensional projective space is parameterized by two types `Point` and `Line` as well as an incidence relation `incid_lp` (see Table 2 for the actual specification in Coq). The two types are equipped with an equality. Axiom `a1_exists` expresses that given two distinct points, one can always define a line going through these points. Axiom `uniqueness` states that given 2 points A and B and 2 lines l and m, if A and B are both incident to both l and m, then either $A = B$ or $l = m$. Axiom `a2`, also known as Pasch axiom, states that two coplanar lines always intersect. Axiom `a3_1` expresses that given a line, there are always three distinct points on it. Axiom `a3_2` expresses that there exist two lines which are not coplanar, thus making the dimension $n > 2$. Finally axiom `a3_3` states that, given 3 lines l_1, l_2 and l_3, there always exists a fourth line m which intersects these 3 lines. This last axiom bounds the dimension so that $n \leq 3$.

2.2 Points, Lines and the Incidence Relation

We choose to use two simple inductive types to represent points and lines of PG(3,2). Points are represented by an inductive datatype of 15 constructors without arguments. Lines are represented in the same way using 35 constructors.

```
Inductive Point :=
| P0 | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 | P9
| P10 | P11 | P12 | P13 | P14.
```

```
Inductive Line :=
| L0 | L1 | L2 | L3 | L4 | L5 | L6 | L7 | L8 | L9 | L10 | L11 | L12
| L13 | L14 | L15 | L16 | L17 | L18 | L19 | L20 | L21 | L22 | L23
| L24 | L25 | L26 | L27 | L28 | L29 | L30 | L31 | L32 | L33 | L34.
```

As there are three points per line, the incidence relation `incid_lp`[4] can be represented in a compact way using the `match ... with` construct of Coq specification language.

```
Definition incid_lp (p:Point) (l:Line) : bool :=
match l with
| L0 => match p with P0 | P1 | P2 => true | _ => false end
| L1 => match p with P0 | P3 | P4 => true | _ => false end
| L2 => match p with P0 | P5 | P6 => true | _ => false end
| L3 => match p with P0 | P7 | P8 => true | _ => false end
| L4 => match p with P0 | P10 | P9 => true | _ => false end
| [...]
end.
```

In order to avoid writing too many specifications and proof scripts manually in Coq, we choose to build an external specification and proofs generator (a simple C program) which takes as input the number of points, the number of lines as well as the incidence relation as a plain file (`pg32.txt`[5]) which contains for each line of the projective space, the list of the points which are incident to it. Given these three elements, the system automatically builds the inductive datatypes for points and lines as well as the incidence relation. It also defines an artificial order on points and lines based on the index of the corresponding points and lines, i.e. `P0` $<$ `P1` $<$ `P2` $<$ ... $<$ `P14`. The specification generator also builds some auxiliary functions, which will be useful to prove existential statements of the form $\forall l1\ l2 : \texttt{Line},\ \exists\ P : \texttt{Point},\ \ldots$.

Using plain inductive datatypes may seem naive. An alternative approach to specify points and lines of PG(3,2) could be to use finite types $'I_n$ of ssreflect and the mathematical components library [12, 17]. However the main drawback is that ssreflect is designed for formal reasoning rather than computing. Thus checking the incidence between a point and a line is a highly expensive operation, which prevents us from carrying out proofs efficently. Using plain inductive types is much more efficient both to check incidence properties and to perform case analysis. The only drawback is that inductive datatypes and functions are huge to write, but this is not that important as we manage to generate these specifications automatically. Overall, our choice is to use the main features of ssreflect, especially the small-scale reflection pattern, but with our own datatypes.

2.3 Formal Proofs

Once the projective space PG(3,2) is described, we check whether all the axioms for projective space geometry hold for this model. This requires proving all axioms of the module defined in `https://github.com/magaud/PG3q/blob/master/generic/pg3x_spec.v` and presented in Figure 2. This is pretty straightforward and we try and make these proofs as generic and efficient as possible. We especially focus on writing general-purpose Ltac tactics, which can be easily reused for other models of projective space such as PG(3,3).

We also rely on our specification generator to enhance producing witnesses for existential quantification. We use a form of skolemisation to write functions which compute the existential variable from the other arguments. For instance, to achieve the proof of lemma `a3_3`, we automatically build a (large) Coq function `f_a3_3` which, given three lines l_1, l_2 and l_3 computes a line l_4 as well as its three intersection points with lines l_1, l_2 and l_3.

```
f_a3_3 : Line -> Line -> Line -> Line * (Point * Point * Point)
```

[4] `https://github.com/magaud/PG3q/blob/master/pg32/pg32_inductive.v`
[5] `https://github.com/magaud/PG3q/blob/master/pg32/pg32.txt`

Table 2 Projective spaces of dimension 3: definitions and properties (`pg3x_spec.v`).

```
Parameter Point, Line : Type.

Parameter eqP : Point -> Point -> bool.
Parameter eqL : Line -> Line -> bool.

Parameter incid_lp : Point -> Line -> bool.

Definition Intersect_In (l1 l2 :Line) (P:Point) :=
  incid_lp P l1 && incid_lp P l2.

Definition dist_3p  (A B C :Point) : bool :=
  (negb (eqP A B)) && (negb (eqP A C)) && (negb (eqP B C)).

Definition dist_4p  (A B C D:Point) : bool :=
      (negb (eqP A B)) && (negb (eqP A C)) && (negb (eqP A D))
      && (negb (eqP B C)) && (negb (eqP B D)) && (negb (eqP C D)).

Definition dist_3l (A B C :Line) : bool :=
     (negb (eqL A B)) && (negb (eqL A C)) && (negb (eqL B C)).

Axiom a1_exists : forall A B : Point,
     {l : Line| incid_lp A l && incid_lp B l}.

Axiom uniqueness : forall (A B :Point)(l1 l2:Line),
     incid_lp A l1 -> incid_lp B l1  ->
     incid_lp A l2 -> incid_lp B l2 -> A = B \/ l1 = l2.

Axiom a3_1 : forall l:Line,
    {A:Point & {B:Point & {C:Point | (dist_3p A B C) &&
      (incid_lp A l && incid_lp B l && incid_lp C l)}}}.

Axiom a2 : forall A B C D:Point, forall lAB lCD lAC lBD :Line,
      dist_4p A B C D ->
      incid_lp A lAB && incid_lp B lAB ->
      incid_lp C lCD && incid_lp D lCD ->
      incid_lp A lAC && incid_lp C lAC ->
      incid_lp B lBD && incid_lp D lBD ->
      (exists I:Point, incid_lp I lAB && incid_lp I lCD) ->
      exists J:Point, incid_lp J lAC && incid_lp J lBD.

Axiom a3_2 : exists l1:Line, exists l2:Line,
       forall p:Point, ~(incid_lp p l1 && incid_lp p l2).

Axiom a3_3 : forall l1 l2 l3:Line,
  dist_3l l1 l2 l3 ->
  exists l4 :Line, exists J1:Point,exists J2:Point,exists J3:Point,
  Intersect_In l1 l4 J1 &&
  Intersect_In l2 l4 J2 &&
  Intersect_In l3 l4 J3.
```

As a consequence, proving the statement `a3_3` boils down to feeding Coq with the correct existential variables for the line and the three intersection points, obtained by applying the function `f_a3_3` instead of trying all possible lines and points (there are 35 possible lines and 15 possible points) for each of the $35^3 = 42\,875$ possible cases for parameters l_1, l_2 and l_3 of lemma `a3_3`.

Once that we checked that our implementation of PG(3,2) verifies all the axioms of projective space, we shall study some specific subsets of points and lines, namley spreads and packings.

3 Collineations, Spreads and Packings of PG(3,2)

Spreads are sets of lines of a projective space which can be defined when the number of points per line divides the number of points. This is the case for all PG(n,q) whose dimension n is odd. PG(3,2) features 15 points and has 3 points per line. Thus spreads exist in PG(3,2).

3.1 Collineations

A collineation is a couple of two functions $f_p : \texttt{Point} \to \texttt{Point}$ and $f_l : \texttt{Line} \to \texttt{Line}$ where both f_p and f_l are bijections and respect the incidence relation.

```
Definition inj {A:Set} {B:Set} (f:A->B) : Prop :=
  forall x y:A, f x = f y -> x = y.
Definition surj {A:Set} {B:Set} (f:A->B) : Prop :=
  forall y:B, exists x:A, y=f(x).

Definition bij {A:Set} {B:Set} (f:A->B) : Prop := (inj f) /\ (surj f).

Definition is_collineation fp fl :=
  ((bij fp) /\ ((bij fl) /\
  (forall x l, incid_lp x l -> incid_lp (fp x) (fl l)))).
```

Collineations, which are automorphisms of PG(3,2) which respect the incidence relation, shall be useful to establish that two given sets of lines are isomorphic, thus allowing us to classify spreads and packings into equivalence classes.

3.2 Spreads

3.2.1 Definition and Properties

A spread of PG(3,q) is a set of $q^2 + 1$ lines which are pairwise disjoint and thus partition the set of points. In PG(3,2), it corresponds to some sets of 5 lines. As recalled in [3, 7, 15], it is well known that there is only one spread (up to isomorphism) in PG(3,2).

3.2.2 Generating all Spreads of PG(3,2)

Using our external specification and proofs generating program, we automatically compute all sets of lines of PG(3,2) which are disjoint and cover all the points. As lines contain exactly 3 points, they need to be sets of exactly 5 lines so that all the points of PG(3,2) are accounted for. We generate 56 distinct spreads (modulo permutations of the order of the lines involved). These spreads are defined in Coq as a list of 56 sets of 5 lines, as follows:

```
Definition S0 := [ L0; L19; L24; L28; L33 ].
Definition S1 := [ L0; L19; L26; L29; L32 ].
[...]
Definition spreads := [ S0 ; S1 ; S2 ; ... ; S54; S55 ].
```

We also generate automatically the collineations[6] which allows to go from the spread S_i to the spread $S_{((i+1) \bmod 56)}$ of the list `spreads`[7] as shown in the following example for the spreads `S0` and `S1`.

```
Definition fp0_1 (p:Point) := match p with P0 => P0 | P1 => P1 | P2 =>
P2 | P3 => P3 | P4 => P4 | P5 => P5 | P6 => P6 | P7 => P11 | P8 => P12
| P9 => P13 | P10 => P14 | P11 => P7 | P12 => P8 | P13 => P9 | P14 =>
P10 end.

Definition fl0_1 (l:Line) := match l with L0 => L0 | L1 => L1 | L2 =>
L2 | L3 => L5 | L4 => L6 | L5 => L3 | L6 => L4 | L7 => L7 | L8 => L9 |
L9 => L8 | L10 => L11 | L11 => L10 | L12 => L12 | L13 => L14 | L14 =>
L13 | L15 => L15 | L16 => L18 | L17 => L17 | L18 => L16 | L19 => L19 |
L20 => L20 | L21 => L21 | L22 => L22 | L23 => L25 | L24 => L26 | L25
=> L23 | L26 => L24 | L27 => L30 | L28 => L29 | L29 => L28 | L30 =>
L27 | L31 => L34 | L32 => L33 | L33 => L32 | L34 => L31 end.
```

3.3 Packings

Once that we have built spreads as (disjoint) sets of lines covering all the points, we can define packings as sets of spreads covering all the lines of PG(3,2).

3.4 Definition and Properties

A packing of PG(3,q) is a set of $q^2 + q + 1$ spreads which are pairwise disjoint and thus partition the set of lines. In PG(3,2), it corresponds to some sets of 7 spreads. There are 240 packings, each of them being a list of 7 spreads. As recalled in [3, 7, 15], there are (up to isomorphism) exactly two distinct classes of packings in PG(3,2).

3.5 Generating all Packings of PG(3,2)

We generate all sets of spreads which are disjoint and cover all the lines, and which thus are packings. As before, these sets of spreads must have 7 elements, as the number of spreads multiplied by the number of lines in each spread must be equal to the number of lines (35) of PG(3,2). As expected (see Theorem 17.5.6 in [14]), we find 240 labelled packings.

```
Definition PA0 := [ S0; S9; S19; S24; S36; S46; S53 ].
Definition PA1 := [ S0; S9; S19; S28; S38; S40; S53 ].
Definition PA2 := [ S0; S9; S20; S27; S36; S46; S49 ].
[...]
Definition packings := [ PA0 ; PA1 ; PA2 ; ... ; PA238 ; PA239 ].
```

These packings belong to two distinct classes `class0` and `class1`[8].

[6] https://github.com/magaud/PG3q/blob/master/pg32/pg32_spreads_collineations.v
[7] https://github.com/magaud/PG3q/blob/master/pg32/pg32_spreads_packings.v
[8] https://github.com/magaud/PG3q/blob/master/pg32/pg32_spreads_packings.v

```
Definition class0 := [PA0; PA3; PA5; ... PA237; PA239 ].
Definition class1 :=  complement class0 packings.
```

As for spreads, we automatically generate the collineations[9] which allow to go from one packing of the list `class0` (resp. `class1`) to the next packing of `class0` (resp. `class1`).

Now that all externally-computed spreads and packings are defined in Coq, we shall formally verify that they actually are spreads and packings of PG(3,2). We shall also check that all the spreads are isomorphic and that the 240 packings can be classified into two distinct classes of 120 elements.

4 Properties of the Spreads of PG(3,2)

4.1 Characterizing all Spreads of PG(3,2)

Spreads can be specified using the following definitions: the boolean function `is_partition` computes whether the lists of points p, q, r, s and t partition the set of points and `is_spread5`[10] used in conjunction with the function `all_points_of_line` computes whether the lines $l1, l2, l3, l4$ and $l5$ actually constitutes a spread. The boolean function `forall_Point` is a finite universal quantification: this means that `forall_Point (fun t => X t)` stands for `X P0 && X P1 && X P2 ... && X P14`.

```
  Definition is_partition (p q r s t: list Point) :bool :=
  (forall_Point
  (fun x => inb x p || inb x q || inb x r || inb x s || inb x t))
  &&
  (forall_Point
  (fun x => negb (inb x p && inb x q && inb x r &&
                            inb x s && inb x t))).

Definition is_spread5 (l1 l2 l3 l4 l5:Line) : bool :=
  disj_5l l1 l2 l3 l4 l5  &&
  is_partition (all_points_of_line l1) (all_points_of_line l2)
               (all_points_of_line l3) (all_points_of_line l4)
               (all_points_of_line l5).
```

Once these definitions are set, we prove that the spreads of PG(3,2) are exactly the ones automatically generated by our external program. On the one hand, we easily check that all the computed spreads belonging to the list `spreads` actually verify the property `is_spread5` of being a spread. On the other hand, we prove that all sets of 5 lines verifying the property `is_spread5` belong to the proposed list `spreads`. Due to the size of the proofs and in order to make them accepted by the Coq proof assistant, we need to decompose this part of the proof into 35 specific cases. Each of them corresponds to one of the cases $l1 =$ `L0`, $l1 =$ `L1`, $\ldots$, $l1 =$ `L34`. Eventually, using all these auxiliary lemmas, we prove the following property:

```
Lemma is_spread_descr : forall l1 l2 l3 l4 l5,
    leL l1 l2 && leL l2 l3 && leL l3 l4 && leL l4 l5 ->
    (is_spread5 l1 l2 l3 l4 l5) <-> In [l1;l2;l3;l4;l5] spreads.
```

In the previous statement, `leL` is a total order on the datatype `Line`, which expresses that `L0` $\leq$ `L1` $\leq \ldots$ `L34` and allows to only consider ordered spreads of lines.

[9] https://github.com/magaud/PG3q/blob/master/pg32/pg32_packings_collineations.v
[10] https://github.com/magaud/PG3q/blob/master/pg32/pg32_spreads.v

4.2 Classifying all Spreads of PG(3,2)

The next step consists in proving that all 56 spreads of PG(3,2) are isomorphic. It can be expressed by stating that there exists a collineation, i.e. an automorphism of PG(3,2) which respects incidence, between any two spreads of PG(3,2).

```
Definition are_isomorphic (s1:list Line) (s2:list Line) : Prop :=
  exists fp, exists fl, ((is_collineation fp fl) /\ (map fl s1 = s2)).
```

We show that the property `are_isomorphic`[11] is reflexive and transitive. Thanks to these results, we show that proving the equivalence can be achieved by simply proving that there exists a collineation (we actually build it) from the n-th element of the list to the $(n+1 \bmod 56)$-th element of the list `spreads`. Using this transitivity property and the collineations computed by our external program, we fairly easily prove the following statement:

```
Lemma all_isomorphic_lemma : forall t1 t2 : list Line,
  In t1 spreads -> In t2 spreads -> are_isomorphic t1 t2.
```

Overall, in this section, we formally proved in Coq that there are 56 labelled spreads in PG(3,2) and that there are all isomorphic.

5 Properties of Packings of PG(3,2)

In the following, we shall prove that there are 240 labelled packings in PG(3,2) and that they can be classified into two distinct classes.

5.1 Characterizing all Packings of PG(3,2)

A packing is defined using the predicate `is_packing7`[12] as a set of 7 spreads (each being a list of lines) which are disjoint and form a partition of the set of lines.

```
Definition is_partition7 (p q r s t u v: list Line) : bool :=
   (forall_Line
     (fun x => inbL x p || inbL x q || inbL x r || inbL x s ||
               inbL x t || inbL x u || inbL x v))
&& (forall_Line
      (fun x => negb (inbL x p && inbL x q && inbL x r &&
                      inbL x s && inbL x t && inbL x u && inbL x v))).
```

```
Definition is_packing7 (s1 s2 s3 s4 s5 s6 s7:list Line) : bool :=
  disj_7s s1 s2 s3 s4 s5 s6 s7 &&
  is_partition7 s1 s2 s3 s4 s5 s6 s7.
```

Checking that all computed elements of the list `packings` are actual packings is straightforward. Proving that all packings of PG(3,2) are in the list `packings` is a lot more challenging, especially because of the number of cases to deal with. In order to make it tractable in Coq, we prove several (56) lemmas of the form `statement_packings`[13] s for some s.

[11] https://github.com/magaud/PG3q/blob/master/pg32/pg32_spreads_collineations.v
[12] https://github.com/magaud/PG3q/blob/master/pg32/pg32_packings.v
[13] https://github.com/magaud/PG3q/blob/master/pg32/pg32_packings.v

```
Definition statement_packings s :=
  forall s2 s3 s4 s5 s6 s7 : list Line,
  In s2 spreads -> In s3 spreads -> In s4 spreads ->
  In s5 spreads -> In s6 spreads -> In s7 spreads ->
  ltS s s2 -> ltS s2 s3 -> ltS s3 s4 ->
  ltS s4 s5 -> ltS s5 s6 -> ltS s6 s7 ->
  is_packing7 s s2 s3 s4 s5 s6 s7 ->
  In [s;s2;s3;s4;s5;s6;s7] packings.
```

In each of them, we fix the first spread (e.g. s=S0) and then verify that all packings containing s as the first spread actually belong to the list `packings`.

```
  Lemma aux_S0 : statement_packings S0.
  [...]
  Lemma aux_S55 : statement_packings S55.
```

Finally, we agregate all 56 lemmas to obtain the following property:

```
Lemma is_packing_descr : forall s1 s2 s3 s4 s5 s6 s7 : list Line,
  ltS s1 s2 && ltS s2 s3 && ltS s3 s4 &&
  ltS s4 s5 && ltS s5 s6 && ltS s6 s7 ->
  In s1 spreads -> In s2 spreads -> In s3 spreads -> In s4 spreads ->
  In s5 spreads -> In s6 spreads -> In s7 spreads ->
  (is_packing7 s1 s2 s3 s4 s5 s6 s7) <->
     In [s1;s2;s3;s4;s5;s6;s7] packings.
```

In the above statements, `ltS` is an order on spreads, which implements the lexicographic order on spreads using the order on lines `ltL` as its basic order.

5.2 Classifying all Packings of PG(3,2)

In order to classify the packings of PG(3,2), we shall first prove that there are at most two distinct classes of packings in PG(3,2). This is achieved using the collineations relating packings provided in Sect. 3. Finally, considering two packings (one in each of the conjectured classes), we show that no collineation can transform the first one into the second one.

5.2.1 There are at most 2 Classes of Packings in PG(3,2)

When considering packings, the relation `are_isomorphic`[14] is a bit more complex as collineations may transform a packing into a packing whose spreads are not sorted in increasing order any more. Therefore we enforce that the images of the spreads computed using f_l must be sorted with respect to the relation `ltL`.

```
Definition are_isomorphic
  (p1:list (list Line)) (p2:list (list Line)) : Prop :=
    exists fp, exists fl,
      is_collineation fp fl /\
      forall s:(list Line), In s spreads -> In s p1 -> In
      (sort (map fl s)) spreads /\ In (sort (map fl s)) p2.
```

[14] https://github.com/magaud/PG3q/blob/master/pg32/pg32_packings_collineations.v

Once again, we prove that the property `are_isomorphic` is reflexive and transitive. This allows to prove that all elements of a class are isomorphic by performing a circular permutation, simply proving that there exists a collineation (which was built explicitly by our external specification and proofs generating program) from the n-th element of the list `class0` (resp. `class1`) to the $(n+1 \bmod 120)$-th element of the list `class0` (resp. `class1`).

```
Lemma all_isomorphic_lemma0 :  forall t1 t2 : (list (list Line)),
  In t1 class0 -> In t2 class0 -> are_isomorphic t1 t2.

Lemma all_isomorphic_lemma1 :  forall t1 t2 : (list (list Line)),
  In t1 class1 -> In t2 class1 -> are_isomorphic t1 t2.
```

At this stage, we only proved that there are at most two classes of packings in PG(3,2). The last step of the proof consists in proving that the two classes `class0` and `class1` are distinct. To do that, we choose two packings, e.g. `PA0` and `PA1`, one in each of the supposed classes. We then generate all collineations of PG(3,2) and verify that none of these collineations allows to go from the packing `PA0` to the packing `PA1`.

5.2.2 Characterizing all collineations of PG(3,2)

So far we defined a collineation as a pair of two bijective functions f_p and f_l and a property that these functions respect the incidence relation. We shall see that a collineation (f_p, f_l) can be exactly characterized by simply defining the images of the four following points `P0`, `P1`, `P3` and `P7`. This relies on the property that there are only three points by line and that collineations are bijections which respect the incidence relation.

Let us start by choosing an image for the first point `P0`. Let us then choose an image for the second point `P1`. Then the image of the point `P2`, which is on line `L0`=(`P0P1`) is imposed. It is the third point of the line generated by f_p `P0` and f_p `P1`. Let us choose the image of the third point `P3`, which lies outside line `L0`. The images of points `P4`, `P5`, `P6` (see Fig. 2 for an visual interpretation of the process) are imposed by the rules of the projective spaces and the collineation properties. We can then choose a fourth point `P7`. This point is outside the plane generated by `P0`, `P1` and `P3`. Once these four images f_p `P0`, f_p `P1`, f_p `P3` and f_p `P7` are chosen, the images of all remaining points `P8`, `P9`, `P10`, `P11`, `P12`, `P13`, `P14` are imposed as being third points of some lines defined by the combination of images of the four initial points `P0`, `P1`, `P3` and `P7`. In addition, the images of all lines are fully determined as well. Indeed, the image of the line going through points A and B is the line going through points $f_p\ A$ and $f_p\ B$.

From a combinatorial point of view, we can choose the image of `P0` by f_p among 15 points. The image of `P1` by f_p can be chosen among 14 points (all points except `P0`). The image of `P3` by f_p can be chosen among 12 points (all points except those on line `L0`, which contains points `P0`, `P1` and `P2`). Finally, the image of the fourth point `P7` by f_p can only be chosen outside of the images of points `P0`, `P1`, `P2`, `P3`, `P4`, `P5`, and `P6`, thus leaving only 8 options available. Once the images of these four points are chosen, the collineation is fully characterized because both f_p and f_l must be bijective and that they must respect the incidence relation. This means that there are $15 \times 14 \times 12 \times 8 = 20\,160$ different ways to define a collineation of PG(3,2).

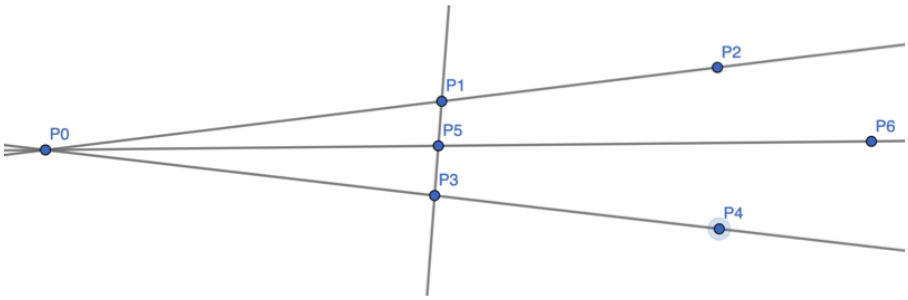


Figure 2 Describing a collineation of PG(3,2) can be achieved by simply providing the images of points `P0`, `P1`, `P3` and `P7`.

Our external specification and proofs generating program takes care of computing all possible collineations of PG(3,2). It generates some very large files: `pg32_automorphisms.v`[15] (where the 20 160 collineations are enumerated), `pg32_automorphisms_inv.v` as well as files `pg32_collineationsX.v` and `pg32_decompX.v`, with X ranging from 0 to 14. The list of all collineations is splitted into smaller lists of size 96 in order to be able to handle the proofs in Coq. Each subset of 96 collineations corresponds to specific collineations whose images of points `P0` and `P1` are the same.

```
Definition all_c0 := [
  (fp_0, fl_0); (fp_1, fl_1); ... ; (fp_94, fl_94); (fp_95, fl_95)].
[...]
Definition all_collineations :=
  all_c0 ++ all_c1 ++ all_c2 ++ ... ++all_c208 ++ all_c209.
```

On the one hand, for each of these subsets, we can check that the given collineations actually verify the property `is_collineation`. On the other hand, we verify that all collineations which verify the following conditions: f_p `P0` = `PX` and f_p `P1` = `PY` actually belong to the corresponding subsets of collineations, namely `all_cZ` where $Z = 14 \times X + Y$. As an example, all collineations which respect the conditions f_p `P0` = `P8` and f_p `P1` = `P2` belong to the subset of collineations `all_c114`.

```
Lemma is_collineations_descr_B_P8_P2 :
  forall fp fl, is_collineation fp fl -> fp P0 = P8 -> fp P1 = P2 ->
  In (fp,fl) all_c114.
```

To prove that a specific collineation (`fp`,`fl`), characterized by `fp P0` = `P8`, `fp P1` = `P2`, ..., actually corresponds to an element of the list `all_c114`, we rely on extensionality and thus add the two following safe axioms to our development:

```
Lemma fp_ext: forall (fp:Point->Point) (fp':Point->Point),
  (forall (p:Point), fp p = fp' p) -> fp = fp'.
Admitted.
```

```
Lemma fl_ext : forall (fl:Line->Line) (fl':Line->Line),
  (forall (l:Line), fl l = fl' l) -> fl = fl'.
Admitted.
```

[15] https://github.com/magaud/PG3q/blob/master/pg32/pg32_automorphisms.v

Splitting the main statement characterizing all collineations into 210 smaller statements allows to handle the proofs in Coq. Thankfully, all these 210 statements are almost automatically generated. Only some minor parts, which we plan to fully automate in a near future, require to be fixed by hand (e.g. unfolding some specific constants or making the naming of lemmas coherent to prove the general statement). The last step consists in agregating all these lemmas to obtain the following statement, which explicitly characterize all collineations of PG(3,2).

```
Lemma is_collineations_descr : forall fp fl,
  is_collineation fp fl <-> In (fp,fl) all_collineations.
```

5.2.3 There are exactly 2 Distinct Classes of Packings in PG(3,2)

Now that we have a list of all collineations of PG(3,2) at our disposal, we can traverse it to verify that none of these collineations allow to transform a packing of the class `class0`, say `PA0` into a packing not in `class0`, say `PA1`. The proof simply consists in assuming, for each collineation that they allow to transform the packing `PA0` into the packing `PA1` and exhibit a contradiction. As we must check all collineations, the Coq file has more that 20 160 lines.

```
Lemma not_iso : ~are_isomorphic PA0 PA1.
```

The statement `not_iso`[16] shows that the two classes of packings `class0` and `class1` are distinct. We can conclude that the 240 packings of PG(3,2) belong to two distinct classes, each of these classes containing exactly 120 elements.

6 Discussion

The Coq development is quite large. It contains more than 50 files. Thankfully, most for them are automaticaly generated. It consists in more than 317 345 lines of specifications and proofs, among them more than 290 000 are proof steps. Some files have about 20 000 lines, which makes them difficult (or at least very slow) to handle in an editor for Coq. Compiling the whole development requires about 13 hours (584 minutes on a Intel (R) Core(TM) i5-4460 CPU @ 3.20GHz with 32GB of memory). Therefore it is important that all proofs are as concise as possible and the development must be well structured as changes in the structure may result in several hours of compilation before being able to resume interactive theorem proving. In the following, we present some proof engineering techniques which proved very useful in our development. We also propose some possible improvements to our work.

6.1 Proof Engineering

6.1.1 Using bool instead of Prop

As we work with finite types, equality and the other relations that we use are decidable. We can directly implement such relations as operations producing elements of the boolean datatype `bool`. This is more convenient than defining them as operations producing elements of type `Prop` together with a decidability property: $\forall\ x\ y, \{x = y\} + \{\neg x = y\}$. This practical approach is inspired by the ssreflect [11] and the mathematical components [17] libraries. In this setting, logical reasoning (eliminating conjunctions or disjunctions) is a bit more technical. However this makes most proofs much easier to complete by simply computing a

[16] https://github.com/magaud/PG3q/blob/master/pg32/pg32_packings_two_distinct_classes.v

boolean value and checking that it is equal to `true`. We could push this technique further and replace all remaining occurences of Leibnitz equality (e.g. in the axiom `uniqueness`) with the two bool predicates `eqP` and `eqL`, respectively implementing equality on `Point` and `Line`.

6.1.2 Optimizing proofs

We design some optimization techniques for generating and checking proof terms. We focus on the current goal, applying some sort of locality principle which means that we try to prove a (sub-)goal the very first time we face it. This means sequences of tactics such as

```
intros a; case a; intros H;
  try (exact (degen_bool )_ H).
solve_goal.
```

must be replaced by more efficient sequences like

```
intros a; case a; intros H;
  solve [(exact (degen_bool )_ H |solve_goal].
```

In this simplified example, we try to apply the tactic `(exact (degen_bool )_ H)` for a subgoal and then we switch to the next subgoal. Eventually we solve the remaining subgoals using the `solve_goal` tactic. The idea here is to solve the goal the first time we encounter it. It is achieved by having several possibilities of tactic applications to solve the goal (this corresponds to the `solve [t1|t2|t3]` syntax). The order of the tactics `t1`, `t2` and `t3` can be highly significant as well: we should always call the tactic which is the most successful one on such subgoals first.

As we face a huge number of cases, we need to design extremely efficient prototype tactics on some specific subgoals and apply them automatically to all the subgoals at stake. Fine tuning the tactics rapidly is the key to making the proofs faster to complete.

Finally, Coq provides some sort of task parallelism in the form of the `par` tactical. It is very useful to deal with all the sub-goals of a proof, once we figure out how to prove the first one. The generic tactic proving the first goal, say `mytactic` can be easily applied to all sub-goals in parallel (in some cases, we have 35x35=1225 or more goals to deal with) by simply writing `par:mytactic`.

6.1.3 Without Loss of Generality

Most proofs are highly branching. For instance, performing case analysis on all three lines to prove the lemma `a3_3` leads to $35^3 = 42875$ cases. In order to make the proof more tractable, we propose a new tactic named `wlog`[17], which implements the *without loss of generality* principle, as it is described in [13]. This allows to reduce the number of cases to solve explicitly. To use it, we build a virtual order on the points and lines, simply mapping point `Pi` (resp. line `Li`) to the value i of its index and then extend statements of the form $\forall l1, l2 : \texttt{Line}, \ldots$ to $\forall l1, l2 : \texttt{Line}, l1 \leq l2 \rightarrow \ldots$

Surprisingly, using the without loss of generality tactic forces us to generalize our statement for Pasch axiom to accommodate all cases, depending on the order in which we consider points A, B, C, and D, as shown in Fig. 3. The usual conclusion of Pasch axiom:

[17] `https://github.com/magaud/PG3q/blob/master/generic/wlog.v`

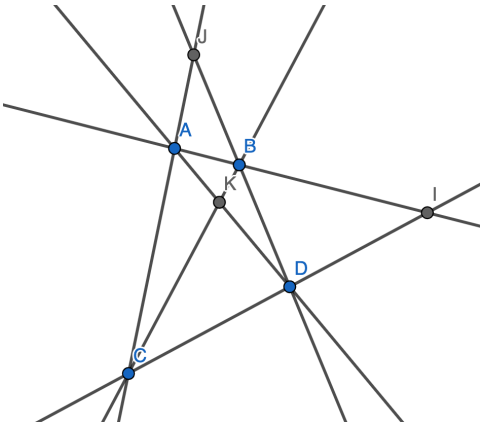


Figure 3 An illustration of the new form of Pasch axiom used to deal with symmetries.

```
(exists I:Point, incid_lp I lAB && incid_lp I lCD) ->
        exists J:Point, incid_lp J lAC &&  incid_lp J lBD.
```

is transformed into a conjunction of two existential properties:

```
(exists I:Point, incid_lp I lAB && incid_lp I lCD) ->
        (exists J:Point,  (incid_lp J lAC && incid_lp J lBD)) /\
        (exists K:Point, (incid_lp K lAD && incid_lp K lBC)).
```

The principles behind the tactic `wlog` were also extremely useful when dealing with spreads and packings, especially when checking inside Coq which sets of lines are actual spreads and which sets of spreads are actual packings.

6.2 Improvements

While carrying out such a proof development, one of the main difficulties is to decide what a *small* Coq proof is. Our first experiments crashed because we assumed Coq will handle very large specifications and proofs easily. Instead we needed to scale down our proofs and decompose them a lot to make sure they can be compiled. The current decomposition is probably too strong, but it has the advantage of being tractable by Coq.

Most definitions and properties used in this development are first order. So it would be interesting to implement the same formal description in a first-order prover such as Z3 [10]. It can also be of interest to use first-order tools such as [1] provided in Coq.

From a specification point of view, as collineations can be simply characterized by the images of only four points, we shall study how to remove the bijection on lines from the definition of the collineation and reconstruct it from the bijection on points. This would make the proof development much smaller and reduces the number of objects we are handling simultaneously. In addition, some combinatorial arguments could be used to simplify the specification of spreads `is_spread5` and packings `is_packing7`, e.g. removing the disjointness condition for the input lists. Finally, most proofs are very similar to one another. In the near future, we shall study how symmetry arguments could help reduce the number of cases to handle. We shall also investigate how to carry out circular permutations of the set of points so that some proofs can be factorized by simply specifying the first point or line at stake and then rotating the statement to obtain the other cases.

7 Conclusion and Future Work

In this work, we show how to formalize in Coq the spreads and packings of PG(3,2). Using an external specifications and proofs generating program, we build automatically all the spreads and packings, as well as all the collineations of PG(3,2). We then easily verify that these generated sets of lines (resp. spreads) are actual spreads (resp. packings). We also successfully prove that they are the only ones. In addition, we classify the spreads and packings, showing that there is only one class for the 56 spreads and that the 240 packings are splitted into two classes of 120 elements. Showing that these two classes are distinct required generating and characterizing in Coq all the 20 160 collineations of PG(3,2).

All the proofs carried out in this work are very large. A single case analysis on a point generates 15 cases, and a single case analysis on a line generates 35 cases. In order to let Coq deal correctly with all these proof scripts, we had to decompose our statements into several smaller lemmas, which could each be independently handled by Coq. During this study, we faced case analysis with a huge number of cases as well as debugging proof script with thousands of sub-goals. We propose some proof engineering techniques to make Coq process the files more easily e.g by directly providing witnesses or by pruning the proof tree by using a *without loss of generality* principle.

So far, we only address properties and transformations which remain in the same (projective) space. We are currently working on generating specifications of projective spaces automatically in order to easily have a formal description of two different projective spaces and thus to be able to formally describe constructions as the Bruck-Bose construction which allows to build translation planes from projective planes [5]. In parallel, we plan to formalize the spreads and packings of PG(3,3) and their properties, as presented in [3]. Handling the projective space PG(3,3) which features 40 points and 130 lines (instead of 15 points and 35 lines in the present case study) would require more decisive specification and proofs techniques. Indeed, PG(3,3) has 8 424 distinct spreads and 12 130 560 collineations, compared to the mere 56 spreads and the 20 160 collineations of PG(3,2). One of the main issues to tackle will be to avoid the exhaustive enumeration of all objects at stake. This could be achieved by using algorithms such as McKay algorithm [18] or Faradžev-Read algorithm [19].

References

1 Mickaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. Verifying SAT and SMT in Coq for a Fully Automated Decision Procedure. In *International Workshop on Proof-Search in Axiomatic Theories and Type Theories (PSATTT'11)*, 2011.

2 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art : The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science, An EATCS Series. Springer-Verlag, Berlin/Heidelberg, May 2004. 469 pages.

3 Anton Betten. The packings of pg(3, 3). *Designs, Codes and Cryptography*, 79(3):583–595, 2016. doi:10.1007/s10623-015-0074-6.

4 David Braun, Nicolas Magaud, and Pascal Schreck. Formalizing Some "Small" Finite Models of Projective Geometry in Coq. In Jacques Fleuriot, Dongming Wang, and Jacques Calmet, editors, *Proceedings of Artificial Intelligence and Symbolic Computation 2018 (AISC'2018)*, number 11110 in LNAI, pages 54–69, September 2018. URL: https://hal.inria.fr/hal-01835493.

5 Richard Hubert Bruck and Raj Chandra Bose. The construction of translation planes from projective spaces. *Journal of Algebra*, 1(1):85–102, 1964. doi:10.1016/0021-8693(64)90010-9.

6 Francis Buekenhout, editor. *Handbook of Incidence Geometry*. North Holland, 1995.

7 Frank Nelson Cole. Kirkman parades. *Bull. Amer. Math. Soc.*, 28(9):435–437, December 1922. URL: https://projecteuclid.org:443/euclid.bams/1183485271.

8 Coq development team. *The Coq Proof Assistant Reference Manual, Version 8.13.2*. INRIA, 2021. URL: http://coq.inria.fr.

9 Harold Scott Macdonald Coxeter. *Projective Geometry*. Springer Science & Business Media, 2003.

10 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

11 Georges Gonthier and Assia Mahboubi. A Small Scale Reflection Extension for the Coq system. Technical Report RR-6455, INRIA, 2008. URL: http://hal.inria.fr/inria-00258384/.

12 Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A small scale reflection extension for the coq system. Research Report RR-6455, Inria, Saclay Ile de France, 2015. URL: https://hal.inria.fr/inria-00258384.

13 John Harrison. Without loss of generality. 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 *LNCS*, pages 43–59. Springer, 2009. doi:10.1007/978-3-642-03359-9_3.

14 James William Peter Hirschfeld. *Finite projective spaces of three dimensions*. Oxford mathematical monographs. Clarendon Press ; New York : Oxford University Press, Oxford, 1985.

15 R.H. Jeurissen. Special sets of lines in PG(3, 2). *Linear Algebra and its Applications*, 226-228:617–638, 1995. Honoring J.J. Seidel. doi:10.1016/0024-3795(95)00200-B.

16 Nicolas Magaud. Spreads and packings of pg(3,2), formally! *Electronic Proceedings in Theoretical Computer Science*, 352:107–115, December 2021. doi:10.4204/eptcs.352.12.

17 Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, January 2021. doi:10.5281/zenodo.4457887.

18 Brendan D. McKay. Isomorph-free exhaustive generation. *Journal of Algorithms*, 26(2):306–324, 1998. doi:10.1006/jagm.1997.0898.

19 Ronald C. Read. Every one a winner or how to avoid isomorphism search when cataloguing combinatorial configurations. In B. Alspach, P. Hell, and D.J. Miller, editors, *Algorithmic Aspects of Combinatorics*, volume 2 of *Annals of Discrete Mathematics*, pages 107–120. Elsevier, 1978. doi:10.1016/S0167-5060(08)70325-X.

Formalizing a Diophantine Representation of the Set of Prime Numbers

Karol Pąk
University of Białystok, Poland

Cezary Kaliszyk
Universität Innsbruck, Austria

Abstract

The DPRM (Davis-Putnam-Robinson-Matiyasevich) theorem is the main step in the negative resolution of Hilbert's 10th problem. Almost three decades of work on the problem have resulted in several equally surprising results. These include the existence of diophantine equations with a reduced number of variables, as well as the explicit construction of polynomials that represent specific sets, in particular the set of primes. In this work, we formalize these constructions in the Mizar system. We focus on the set of prime numbers and its explicit representation using 10 variables. It is the smallest representation known today. For this, we show that the exponential function is diophantine, together with the same properties for the binomial coefficient and factorial. This formalization is the next step in the research on formal approaches to diophantine sets following the DPRM theorem.

2012 ACM Subject Classification Theory of computation → Interactive proof systems

Keywords and phrases DPRM theorem, Polynomial reduction, prime numbers

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.26

Category Short Paper

Supplementary Material Formalization can be found at:
Software (Formalization): `http://cl-informatik.uibk.ac.at/cek/itp2022/`

Funding ERC starting grant no. 714034 *SMART* and Cost action CA20111 *EuroProofNet*.

Acknowledgements We would like to thank Yuri Matiyasevich for his comments on the previous version of this paper.

1 Introduction

Hilbert's 10th problem (`H10`) asks whether there exists an algorithm[1] that can determine if a diophantine equation has a solution over the integers. A major step towards the negative resolution of the problem was achieved by the *Davis conjecture*, stating that the notions of diophantine sets and recursively enumerable sets coincide. This is the case since recursively enumerable sets without algorithms for recognizing their elements have already been known. Indeed, by the Davis Normal Form Theorem [3], for every recursively enumerable set $R \subseteq \mathbb{N}^m$ there exist a number n together with a polynomial P over $m+n+2$ variables (n of the variables are parameters and $m+2$ are unknowns) with integer coefficients, such that

$$\forall_{a_1,\ldots,a_n} R(a_1,\ldots,a_n) \iff \exists_x \forall_{y\leq x} \exists_{x_1\leq x,\ldots,x_m\leq x} P(a_1,\ldots,a_n,x,y,x_1,\ldots,x_m) = 0 \quad (1)$$

[1] Today interpreted as an adequate RAM program or equivalently a Turing machine searching for solutions.

Eliminating the single universal quantifies from Equation 1, it becomes the condition defining a diophantine relation. However, the research undertaken by Robinson, Davis, and Putnam to eliminate this quantifier required almost 30 years. It is therefore not surprising that they created several theorems that give a negative solution to the problem but under certain assumptions, which enable such quantifier elimination. One of such assumptions is that the exponential function can be defined in a diophantine way. This has been eliminated by Matiyasevich, who definitively completed the proof of the DPRM-theorem [4, 9], all every recursively enumerable set of natural numbers is diophantine.

Our work formalizes multiple consequences and results that originated from the proof of the DPRM, all concerning diophantine equations. We formalize the fact that the exponential function is diophantine [14], together with the same property for the binomial coefficient and factorial. These allow the proof of DPRM-theorem [12] in a post-Matiyasevich approach proposed by Smorynski [16], where the concept of recursively enumerable is defined using Equation 1.

We also formalize Robinson's [15] conditions for the DPRM-theorem, namely that the set of primes is representable by a diophantine polynomial. This polynomial is sufficient to prove the DPRM-theorem in the Robinson approach. On the other hand, the existence of this polynomial is guaranteed, but not its explicit statement. In fact, stating it explicitly has been a long-standing challenge and the problem of finding the polynomial defining the prime numbers with a minimum number of variables is an open problem in number theory.

In 1971, Yuri Matiyasevich proposed the construction of a diophantine polynomial of degree 37 with 24 variables (one of the variables is a parameter and 23 are unknowns) that defined the set of prime numbers. This result has been improved together with Robinson [11] to a polynomial with 14 variables including 13 unknowns. To do this, they showed that every diophantine polynomial, can be reduced to 13 unknowns. Wada et al. [6] have later reduced the polynomial to 12 variables, however, the rank of the polynomial is 13,697.

The main result of the current work is the formalization of the polynomial over 10 variables that defines the prime numbers together with a large number of formalized results in numeral analysis (263 proved top-level MML theorems totaling 922KB) necessary for this result. This polynomial, proposed by Matiyasevich [10], is today the smallest known[2] polynomial for the open problem. This work improves on the work by the first author, presented at the FMM 2021 workshop [13], where the 26-variable polynomial was defined in Mizar without any further properties.

2 DPRM Formalizations and Their Relation to Number Theory

Larchey-Wendling and Forster [8] formalized the DPRM theorem in Coq and Bayer et al. [1] in Isabelle/HOL. Both formalizations develop register machines, Minsky machines, advanced properties of the Pell equation and prove that exponentiation is diophantine. The first work proved the bounded universal quantification theorem in order to reach the final DPRM theorem, and then additionally showed the undecidability of `H10` and discusses other undecidable problems as future work. The second work proves the DPRM theorem following the approach proposed by Matiyasevich in [7] instead of proving the bounded universal quantification theorem. There, the discussed future work is to extend it to register machines to prove the undecidability of the Halting problem. Carneiro's formalization in Lean [2] uses Pell equations to prove the key lemma of Matiyasevich, stating that exponentiation is diophantine.

[2] Private email exchange with Yuri Matiyasevich, January 2022.

Our work focuses on the applications of H10 listed by Sun [17] and instead of register machines we focus on the theory of polynomials. By H10, $\exists_{x_1,\ldots,x_\nu\in\mathbb{Z}} P(a,x_1,\ldots,x_\nu)=0$ is undecidable for some diophantine polynomial P, $\nu\in\mathbb{N}$. In 1970 Matiyasevich justified $\nu<200$. Further ingenious number-theoretic ideas allowed him to prove together with Robinson [11], $\nu\geq 13$ by developing general diophantine polynomial reduction methods. Observe, that when $\nu\geq 13$, the unknowns range over positive (or non-negative) integers. In 1975, Matiyasevich further announced that $\nu\geq 9$ and Jones gave a complete proof [5], but in both cases (Matiyasevich's announcement and Jones's proof) the range of all unknowns is again limited to the positive (or non-negative) integers. Sun [17] improved this result by modifying the range of the unknowns. He showed the case $\nu\leq 9$ with only one limitation, namely $x_1,\ldots,x_8\in\mathbb{Z}$ and $x_9\in\mathbb{N}$, and thus finally obtained $\nu\leq 11$ where the unknowns range over all integers. For this reason, we focus our work on Sun's result, even if the condition $\nu\leq 9$ seems better than $\nu\leq 11$. This is also our main justification for the work to construct $J_{1+q,\mathbb{C}}$ with an arbitrary q. We use it for $q=3$, $q=7$ as required in [11], and $q=17$ in Sun's number theoretic results [17].

3 Preliminaries

We shortly remind the definition of *diophantine sets*, that will be used in the formalization. A diophantine polynomial in k variables $v_1,v_2,\ldots,v_k$ is a linear combination of monomials with non-zero coefficients of the shape $c\cdot v_1^{p_1}v_2^{p_2}v_3^{p_3}\ldots v_j^{p_j}$, where the coefficients c are integers, the exponents p_i are natural numbers, and v_i are variables. The variables will be separated into parameters and unknowns as follows. A diophantine equation, is an equation of the form $P(x_1,\ldots,x_j,y_1,\ldots,y_k)=0$, where P is a diophantine polynomial and $x_1,\ldots,x_j$, $y_1,\ldots,y_k$ indicate the parameters and unknowns, respectively. A set $D\subseteq\mathbb{N}^n$ of n-tuples is called diophantine if there exists a $n+k$-variable diophantine polynomial P such that $\langle x_1,\ldots,x_n\rangle\in D$ if and only if there exist unknowns $y_1,\ldots,y_k\in\mathbb{N}$ such that $P(x_1,\ldots,x_j,y_1,\ldots,y_k)=0$. Similarly, an n-ary predicate $\mathcal{P}$ is diophantine, iff the set of n-tuples for which the predicate $\mathcal{P}$ is satisfied is diophantine. In particular, the divisibility relation $\mathcal{P}(a,b)\equiv a\mid b$ and congruence $\mathcal{P}(a,b,c)\equiv a\mid b$ are diophantine, as $\mathcal{P}(a,b)\iff\exists_x\, ax-b=0$ and $\mathcal{P}(a,b,c)\iff\exists_x\, a-b-cx=0$.

Note, that the number of variables used in a polynomial defining a diophantine property includes the explicitly stated parameters a,b,c but also the implicitly appearing unknown x. Informally, a function is referred to as diophantine if the relation between its arguments and its results is. In particular, the key lemma of Matiyasevich, stating that the exponential function is diophantine means that the relation $\mathcal{P}(a,b,c)\equiv a=b^c$ is.

4 Pell Equation

Even if Matiyasevich originally showed the key lemma using properties of the Fibonacci sequence, further results and publications in the domain use the Pell equation instead. The Pell equation states $x^2-Dy^2=1$ with a non-square parameter D. If $D=a^2-1$, we can explicitly give all the solutions of this equation via Lucas sequences: $x=\chi_a(n), y=\psi_a(n)$:

$$\begin{aligned}
&\chi_a(0)=1, && \chi_a(1)=a, && \chi_a(n+2)=2a\chi_a(n+1)-\psi_a(n),\\
&\psi_a(0)=0, && \psi_a(1)=1, && \psi_a(n+2)=2a\psi_a(n+1)-\psi_a(n).
\end{aligned}$$

which is the approach used in the HOL-Light, the Lean [2] and Mizar [14] formalizations. Alternatively, one can consider the equation $(ax-y)^2-(a^2-1)x^2=1$. This, transformed as $x^2-bxy+y^2=1$ with $b=2a$ can be used to build the sequence of solutions $\alpha_b(n)$. That

sequence has the interesting property: if $x^2 - bxy + y^2 = 1$ then either $x = \alpha_b(m), y = \alpha_b(m+1)$ or $x = \alpha_b(m+1), y = \alpha_b(m)$. The latter approach is used in the Coq [8] and Isabelle [1] formalizations. The similarity can be analysed by noticing the relation $\alpha_b(n) = \psi_{2a}(n)$. Of course ψ is more general, while α has more properties. However, the condition $\alpha_b(k) \mid \alpha_n(m) \Leftrightarrow k \mid m$ (see Equation (3.23) in [7]) is explicitly stated in the publication describing the Coq formalization [8] and the previous version of the Isabelle formalization [1] while $\psi_a(k) \mid \psi_n(m) \Leftrightarrow k \mid m$ is proved in HOL Light, Lean [2] and in Mizar [14][3].

Irrespective of the considered sequence, all formalizations prove that $a = \alpha_b(c)$ or $a = \psi_b(c)$ can be represented using an (implicit) diophantine relation, stated as a combination of less complicated diophantine relations. Additionally these sub-relation use additional explicit unknowns and may also have implicit ones in these relations. For example Bayer et al. [1] express $3 < b \wedge a = \alpha_b(c)$ using 6 explicit unknowns and 15 relations including, e.g., 4 uses of equivalence $\equiv$. Carneiro [2] uses 5 additional unknowns explicitly and several congruences. Similarly, our previous work used 6 explicitly given additional unknowns [14], reduced to 5 explicit unknowns and a single implicit ones [13] in order to achieve the 26-variable polynomial (3). Here, in order to formalize the best known 10-variable polynomial proposed in [10], we use the representation proposed by Matiyasevich and Robinson [11]: Two explicit unknowns i, j and 3 implicit ones corresponding to the relations $=\square$ ($=\square$ is a one-argument relation, which is true when the argument is a square of a natural number, i.e., $x = \square \Leftrightarrow \exists_{n \in \mathbb{N}} x = n^2$), $\mid$ and $\leq$.

Note that the Theorem 1 also depends on the parameter $e \in \mathbb{N}$, which can be simply eliminated by $e = 0$. However, we use the original formulation to simplify the comparison of conditions in Theorem 1 and Theorem 2, where we substitute $e = L-1$.

▶ **Theorem 1** (`HILB10_8:19`). *Let $A, B, C \in \mathbb{N}$ with $A > 1$, $B > 0$ and $e \in \mathbb{N}$. Then $C = \psi_A(B)$ if and only if there exists $i, j \in \mathbb{N}$ and auxiliary unknowns $D, E, F, G, H, I \in \mathbb{Z}$ such that*

$$DFI =\square \wedge F \mid (H - C) \wedge B \leq C \tag{2}$$

and $D = (A^2-1)C^2+1$, $E = 2(i+1)D(e+1)C^2$, $F = (A^2-1)E^2 + 1$, $G = A+F(F-A)$, $H = B+2jC$, $I = (G^2-1)H^2+1$, where the auxiliary unknowns can be replaced by polynomials over A, B, C, i, j and e.

Therefore, $C = \psi_A(B)$ can be represented as $0 = (DFI-\alpha^2)^2 + (F\beta-H+C)^2(F\beta+H-C)^2 + (B+\gamma-C)^2$, where $\alpha, \beta, \gamma \in \mathbb{N}$ are hidden unknowns.

5 Prime Numbers

The main idea behind the construction of a polynomial representing the prime numbers uses Wilson's theorem, i.e., for any positive integer k, $k+1$ is prime if and only if $k+1 \mid k! + 1$. Note that in Mizar we had to formalize the fact that $y = x!$ is a diophantine relation, since it

[3] See the complete formalization statement of the theorem `Y_DIVIDES` in HOL-Light `https://github.com/jrh13/hol-light/blob/master/Examples/pell.ml`, the theorem `y_dvd_iff` in Lean `https://github.com/leanprover-community/mathlib/blob/master/src/number_theory/pell.lean`, theorems `HILB10_1:34` and `HILB10_1:36` in Mizar `http://mizar.uwb.edu.pl/version/current/html/hilb10_1.html`.

is one of the key steps to proving the DPRM-theorem in our approach. A proof that only focuses on the existence of this polynomial can be expressed in a surprisingly concise way (less than 100 Mizar lines of proof) using higher-order schemes. Compare this with more than 2000 lines required to prove that for any $k \in \mathbb{N}^+$ holds $k+1$ is prime if and only if, there exist $a - z \in \mathbb{N}$ unknowns for which

$$\begin{aligned}&(wz+h+j-q)^2 + ((gk+g+k)(h+j)+h-z)^2 + ((2k)^3(2k+2)(n+1)^2+1-f^2)^2+\\&(p+q+z+2n-e)^2+(e^3(e+2)(a+1)^2+1-o^2)^2+(x^2-(a^2-1)y^2-1)^2+(16(a^2-1)r^2y^2y^2+1-u^2)^2+\\&(((a+u^2(u^2-a))^2-1)(n+4dy)^2+1-(x+cu)^2)^2 + (m^2-(a^2-1)l^2-1)^2 + (k+i(a-1)-l)^2+\\&(n+l+v-y)^2 + (p+l(a-n-1) + b(2a(n+1)-(n+1)^2-1)-m)^2+\\&\quad(q+y(a-p-1)+s(2a(p+1)-(p+1)^2-1)-x)^2 + (z+pl(a-p)+t(2ap-p^2-1) - pm)^2\end{aligned} \tag{3}$$

equals zero.

To get closer to the 10 variables, we define the notion of prime numbers following [10]:

▶ **Theorem 2** (`HILB10_8:23`). *Let $k \in \mathbb{N}$. Then k is prime if and only if there exists $f, i, j, m, u \in \mathbb{N}^+$, $r, s, t \in \mathbb{N}$ unknowns and auxiliary unknowns $A - I, L, M, S - W, Q \in \mathbb{Z}$ such that*

$$\begin{aligned}&DFI = \square \wedge (M^2-1)S^2+1 = \square \wedge ((MU)^2-1)T^2+1 = \square \wedge\\&(4f^2-1)(r-mSTU)^2 + 4u^2S^2T^2 < 8fuST(r-mSTU) \wedge\\&FL \mid (H-C)Z + F(f+1)Q + F(k+1)((W^2-1)Su - W^2u^2+1)\end{aligned} \tag{4}$$

and $A = M(U+1)$, $B = W+1$, $C = r+W+1$, $D = (A^2-1)C^2+1$, $E = 2iC^2LD$, $F = (A^2-1)E^2+1$, $G = A+F(F-A)$, $H = B+2(j-1)C$, $I = (G^2-1)H^2+1$, $W = 100fk(k+1)$, $U = 100u^3W^3+1$, $M = 100mUW+1$, $S = (M-1)s+k+1$, $T = (MU-1)t+W-k+1$, $Q = 2MW-W^2-1$, $L = (k+1)Q$.

One can verify, that the simplest polynomial specified by Equation 4, uses 8 unknowns explicitly along with one implicit one for each relation (three occurrences of $=\square$, inequality, and divisibility). Together with k this gives a total of 14 variables. The fact, that this does not require 5 implicit unknowns, was a striking solution proposed in [10]. In the next section, we will reduce the 5 implicit unknowns used for Equation 4 to a single one.

6 The Polynomial Reduced to a Single Unknown Variable

We first notice $a = \square \equiv \exists_{x \in \mathbb{N}}\, 0 = x^2 - a = \prod(x \pm \sqrt{a})$, where the product considers all sign combinations. Of course, the product is a polynomial, but its factors are not. Additionally, a can be negative, so we need to consider $\mathbb{C}$ as the domain and take into account the non-uniqueness of the square root (however, since $a \in \mathbb{Z}$, there is only one root in the first quadrant of the complex plane).

▶ **Theorem 3.** *Suppose $A_1, \ldots, A_q \in \mathbb{Z}$. Then $A_1 = \square, \ldots, A_q = \square$ if and only if*

$$0 = \prod(X \pm \sqrt{A_1} \pm \sqrt{A_2}W \pm \ldots \pm \sqrt{A_q}W^{q-1})$$

for some $X \in \mathbb{Z}$ where $W = 1 + A_1^2 + \ldots + A_q^2$.

Theorem 3 is formulated in [11] and used for $q = 7$, however, the formalization additionally requires a justification that the product over 2^q possible combinations of signs eliminates similar elements giving a linear combination of $\binom{2^{q-1}+q-1}{q-1}$ monomials with non-zero coefficients. For this, we will define a helper polynomial $J_{n,\mathcal{R}}$ in Theorem 4. There, all factors that included square roots will appear in even powers, which will eliminate these roots. This allows using $J_{q+1,\mathbb{C}}(r_0, \sqrt{r_1 W^2}, \ldots, \sqrt{r_{n-1} W^{2q-2}})$ as an appropriate $\mathbb{Z}$-valued polynomial over $q+1$, used in Theorem 3 following Matiyasevich's elegant adaptation in order to ensure the satisfiability of all 5 predicates from *Equation* 4. Our current formalization does not include Theorem 3 in its full generality (ongoing work with most of the needed lemmas complete), as we only use it with $q = 3$ and substitution the constant $W = 2$ instead of the polynomial $W = 1 + r_1^2 + \ldots + r_q^2$.

▶ **Theorem 4** (`POLYNOM9:def 10`). *Let $\mathcal{R}$ be a commutative ring, $n \in \mathbb{N}$ with $n > 1$. There exists an $\mathcal{R}$-valued polynomial over n variables $J_{n,\mathcal{R}}$ obeying the following conditions:*

- *$J_{n,\mathcal{R}}(r_1, r_2, \ldots, r_n) = \prod(r_1 \pm r_2 \pm \ldots \pm r_n)$ for all $r_1, r_2, \ldots, r_n \in \mathcal{R}$,*
- *let $p_\alpha R^\alpha$ be any monomial with nonzero coefficients of $J_{n,\mathcal{R}}$, where $R^\alpha = r_1^{\alpha_1} \cdot r_2^{\alpha_2} \cdot \ldots \cdot r_n^{\alpha_n}$. Then every power of α_i is even, the sum of the factors $\sum_{i=1}^n \alpha_i$ is equal to 2^{n-1}, coefficient p_α is an integer multiple of $1_\mathcal{R}$, i.e., is equal to $1_\mathcal{R}+1_\mathcal{R}+\ldots+1_\mathcal{R}$ or $-1_\mathcal{R}-1_\mathcal{R}\ldots-1_\mathcal{R}$ and the coefficient of $r_1^{2^{n-1}}$ equals $1_\mathcal{R}$.*

The existence of a polynomial that has these properties is quite an involved proof by induction. We only show here the outline of the most difficult part. By the induction hypothesis: $\prod(r_1 \pm \ldots \pm r_n \pm r_{n+1}) = \prod(r_1 \pm \ldots \pm (r_n + r_{n+1})) \cdot \prod(r_1 \pm \ldots \pm (r_n - r_{n+1})) = \sum_\alpha c_\alpha R^\alpha (r_n + r_{n+1})^{2i_\alpha} \cdot \sum_\beta c_\beta R^\beta (r_n - r_{n+1})^{2i_\beta}$ where R^α represent products of $r_1, \ldots, r_{n-1}$ to even powers. We multiply the sums as follows. If $i_\alpha = i_\beta$, then $c_\alpha R^\alpha (r_n + r_{n+1})^{2i_\alpha} c_\beta R^\beta (r_n - r_{n+1})^{2i_\beta} = c_\alpha c_\beta R^\alpha R^\beta (r_n^2 - r_{n+1}^2)^{i_\alpha}$. If $i_\alpha < i_\beta$ (the case $i_\alpha > i_\beta$ is similar) we add each two summands

$$c_\alpha R^\alpha (r_n + r_{n+1})^{2i_\alpha} c_\beta R^\beta (r_n - r_{n+1})^{2i_\beta} + c_\beta R^\beta (r_n + r_{n+1})^{2i_\beta} c_\alpha R^\alpha (r_n + r_{n+1})^{2i_\alpha} = c_\alpha c_\beta R^\alpha R^\beta \cdot (r_n^2 - r_{n+1}^2)^{i_\alpha} \cdot \sum_{i=0}^{i_\beta - i_\alpha} 2 \cdot \binom{2(i_\beta - i_\alpha)}{2i} r_n^{2i} r_{n+1}^{2(i_\beta - i_\alpha - i)}. \quad (5)$$

In both cases, we obtain a polynomial, where all variables are raised to even powers, which completes the most involved part of the proof. ⌟

The complete formalized proof of this theorem includes all the required sign combinations and required 143 helper lemmas and 13K lines of proofs.

The next step in the simplification of the polynomial given in Theorem 3 (following [10]), proceeds by defining $K_1(y, x_1, x_2, x_3)$ to be $J_{4,\mathbb{C}}(-y, \sqrt{x_1}, \sqrt{4x_2}, \sqrt{16x_3})$ and proving $\exists_{y \in \mathbb{Z}} K_1(y, x_1, x_2, x_3) = 0 \Leftrightarrow x_1 = \square \wedge x_2 = \square \wedge x_3 = \square$ under the assumptions $x_1, x_2, x_3 \in \mathbb{N}$ and $2 \nmid x_1, 2 \nmid x_2$. Note, that the substitution (compare Equation 4)

$$x_1 = (M^2 - 1)S^2 + 1,\ x_2 = ((MU)^2 - 1)T^2 + 1,\ x_3 = DFI$$

satisfies these assumptions.

The next step in the informal proof performs a rational substitution in the integer polynomial and justifies that this is again an integer polynomial. This requires some work with the type system in the formalization. Indeed, we perform the substitution $y := y - \frac{r}{p}$ in K_1, where p, r are the new variables. In order to construct the polynomial $K_2(y, x_1, x_2, x_3, p, r)$ to be $p^8 \cdot K_1(y - \frac{r}{p}, x_1, x_2, x_3)$ for $y, x_1, x_2, x_3, p, r \in \mathbb{R}$ where $p \neq 0$, the formalization is split into two stages. First, we define $K_2'(y, x_1, x_2, x_3, z) = K_1(y - z, x_1, x_2, x_3)$, where the power

α of z in each monomial in K_2' is ≤ 8 and replace z^α by $p^\alpha r^{8-\alpha}$ obtaining K_2. This way, we obtain a polynomial, where $K_2(y, x_1, x_2, x_3, p, r) = 0$ ensures $p|r$ for every $p, r \in \mathbb{N}, p \neq 0$, but both factors in Equation 4 are non-negative and $FL > 0$.

The final theorem confirms that K is a polynomial:

▶ **Theorem 5** (POLYNOM9:77). *Let $x_1, x_2, x_3, p, r, n \in \mathbb{N}$, $v \in \mathbb{Z}$ where $2 \nmid x_1$, $2 \nmid x_2$, $p > 0$, $n > \sqrt{x_1} + 2\sqrt{x_2} + 4\sqrt{x_3} + r$. Then $\exists_{y \in \mathbb{N}} K(y, x_1, x_2, x_3, p, r, n, v) = 0$ iff $x_1 = \square \wedge x_2 = \square \wedge x_3 = \square \wedge p \mid r \wedge 0 \leq v$, where we have $K(y, x_1, x_2, x_3, p, r, n, v) = K_2(y - nv, x_1, x_2, x_3, p, r)$.*

With K, we can represent Equation 4 using a single implicit unknown and we can represent primes using the 10-variable polynomial with the following substitutions in K:

$$\begin{aligned} v &= 8fuST(r - mSTU) - ((4f^2 - 1)(r - mSTU)^2 + 4u^2S^2T^2) - 1, \\ n &= MS + 2MUT + 4A^2CEGH + 2(HL + FfQ + Fk(W^2Su + W^2u^2)). \end{aligned}$$

With the abbreviations expanded, this gives a diophantine polynomial *Poly* of degree > 6000 over parameter k and unknowns $f, i, j, m, u, r, s, t, y$, so we present only the non-expanded version with the following property. Let $k \in \mathbb{N}^+$. Then $k+1$ is prime if and only if there exists a 10-element vector of natural numbers v such that the first element is equal to k ($v.1 = k$) and v is a root of *Poly* ($\mathsf{eval}(Poly, v) = 0.\mathbf{F}_\mathbb{R}$). In the formal proof, rather than specify the existence of a parameter k and 9 unknowns, we simplify this by using a 10-element vector with one element equal to k. The final statement in Mizar is:

```
theorem :: POLYNOM9:85
  ex Poly be INT-valued Polynomial of 10,F_R st
    for k be positive Nat holds
      k+1 is prime iff ex v being natural-valued Function of 10,F_R st
                              v.1 = k & eval(Poly,v) = 0.F_R;
```

This is already the minimal polynomial and completes our goal.

References

1. Jonas Bayer, Marco David, Abhik Pal, Benedikt Stock, and Dierk Schleicher. The DPRM theorem in Isabelle (short paper). In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving, ITP 2019*, volume 141 of *LIPIcs*, pages 33:1–33:7. Dagstuhl, 2019. doi:10.4230/LIPIcs.ITP.2019.33.
2. Mario Carneiro. A Lean formalization of Matiyasevič's theorem, 2018. arXiv:1802.01795.
3. Martin Davis. Arithmetical problems and recursively enumerable predicates. *J. Symb. Log.*, 18(1):33–41, 1953. doi:10.2307/2266325.
4. Martin Davis, Hilary Putnam, and Julia Robinson. The decision problem for exponential diophantine equations. *Annals of Mathematics*, 74:425–436, 1961. doi:10.2307/1970289.
5. James P. Jones. Universal diophantine equation. *The Journal of Symbolic Logic*, 45(3):549–571, 1982.
6. James P. Jones, Daihachiro Sato, Hideo Wada, and Douglas Wiens. Diophantine representation of the set of prime numbers. *The American Mathematical Monthly*, 83(6):449–464, 1976.
7. Michael Lamoureux, editor. *On Hilbert's Tenth Problem*, volume 1. Pacific Institute for the Mathematical Sciences, PIMS Distinguished Chair Lectures, 2000.
8. 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 *LIPIcs*, pages 27:1–27:20, Dagstuhl, Germany, 2019. Dagstuhl. doi:10.4230/LIPIcs.FSCD.2019.27.

9 Yuri Matiyasevich. Enumerable sets are diophantine. *Doklady Akademii Nauk SSSR (in Russian)*, 191:279–282, 1970.

10 Yuri Matiyasevich. Primes are nonnegative values of a polynomial in 10 variables. *Journal of Soviet Mathematics*, 15:33–44, 1981. doi:10.1007/BF01404106.

11 Yuri Matiyasevich and Julia Robinson. Reduction of an arbitrary diophantine equation to one in 13 unknowns. *Acta Arithmetica*, 27:521–553, 1975.

12 Karol Pąk. Formalization of the MRDP theorem in the Mizar system. *Formalized Mathematics*, 27(**2**):209–221, 2019. doi:10.2478/forma-2019-0020.

13 Karol Pąk. Formalization of prime representing polynomial in Mizar. In *FMM 2021 workshop*, 2021. URL: http://alioth.uwb.edu.pl/~pakkarol/articles/KP-FMM2021.pdf.

14 Karol Pąk. The Matiyasevich Theorem. Preliminaries. *Formalized Mathematics*, 25(**4**):315–325, 2017. doi:10.1515/forma-2017-0029.

15 Julia Robinson. Diophantine decision problems. *Studies in number theory*, 6:76–116, 1969.

16 Craig Alan Smorynski. *Logical Number Theory I, An Introduction*. Universitext. Springer-Verlag Berlin Heidelberg, 1991.

17 Zhi-Wei Sun. Further results on Hilbert's Tenth Problem. *Science China Mathematics*, 64:281–306, 2021. doi:10.1007/s11425-020-1813-5.

Kalas: A Verified, End-To-End Compiler for a Choreographic Language

Johannes Åman Pohjola
University of New South Wales, Sydney, Australia

Alejandro Gómez-Londoño
Chalmers University of Technology, Gothenburg, Sweden

James Shaker
Australian National University, Canberra, Australia

Michael Norrish
Australian National University, Canberra, Australia

Abstract

Choreographies are an abstraction for globally describing deadlock-free communicating systems. A choreography can be compiled into multiple endpoints preserving the global behavior, providing a path for concrete system implementations. Of course, the soundness of this approach hinges on the correctness of the compilation function. In this paper, we present a verified compiler for Kalas, a choreographic language. Its machine-checked end-to-end proof of correctness ensures all generated endpoints adhere to the system description, preserving the top-level communication guarantees. This work uses the verified CakeML compiler and HOL4 proof assistant, allowing for concrete executable implementations and statements of correctness at the machine code level for multiple architectures.

2012 ACM Subject Classification Theory of computation → Concurrency; Software and its engineering → Software verification; Software and its engineering → Compilers

Keywords and phrases Choreographies, Interactive Theorem Proving, Compiler Verification

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.27

Supplementary Material *Software (Source Code)*: `https://github.com/CakeML/choreo/`

Acknowledgements We are grateful to Marco Carbone, Rob van Glabbeek and Magnus Myreen for insightful discussion on this work.

1 Introduction

In recent years, advances in the fields of concurrency theory and systems verification have taken us closer to the idea of a truly correct communicating system. The former abounds with beautiful high-level specification formalisms and reasoning techniques for communicating systems. At the same time, the latter provides detailed correctness proofs of the low-level computing infrastructure (e.g., compilers, language runtimes, and operating systems) needed to implement them. There is then much to be gained by joining these worlds. In particular, high-level descriptions of communicating systems – along with their guarantees – could be propagated down to low-level implementations to create an end-to-end result. One promising approach is choreographic programming, which at a high level describes communicating systems while providing by-construction guarantees.

A choreography is a global description of a communicating system, written in a style reminiscent of the `Alice` → `Bob` notation for protocol descriptions. Compared to the traditional approach of writing separate programs for every `Alice` and `Bob`, the choreographic approach has the advantage that it is impossible to write a program with a communication mismatch. In particular, deadlock freedom holds by construction. Furthermore, through a procedure called *endpoint projection* choreographies can be compiled into separate programs for each endpoint, such that their parallel composition implements the global behaviour.

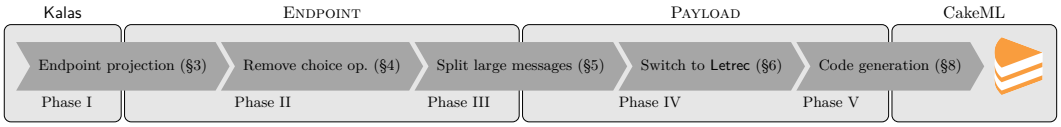


Figure 1 Compilation Steps and Intermediate Languages.

In this paper, we present a compiler for our choreographic language, Kalas, with a machine-checked, end-to-end proof of correctness. That is, we create an environment based on the HOL4 interactive theorem prover [24] where programmers can write choreographies, and then have the system automatically generate executable code for each endpoint, along with a proof of its correct compilation into machine-code.

Our compiler is structured into five phases, illustrated in Figure 1, with associated correctness result for each. The first step is endpoint projection, where the global choreography is projected into a parallel composition of sequential programs implementing each endpoint, expressed in a process algebra we call Endpoint. Second, the Endpoint operators for branch selection are encoded with more primitive operators. Third, Endpoint is compiled to a second process algebra, Payload. While messages in Endpoint can be arbitrarily large, messages in Payload have a fixed size. This step introduces a protocol that divides long messages into chunks, thus accounting for the fact that real communication protocols have bounds on message size, without burdening the application programmer with the details. A fourth step compiles Kalas's fixpoint operator (with substitution semantics) into recursive function definitions (with environment semantics), to align better with functional programming idioms. The final compilation phase compiles Payload's endpoints to CakeML [18], a sequential, functional programming language with a verified compiler, giving us semantics preservation down to the machine code.

Composing the compiler correctness results for each phase, we show that the deadlock freedom of Kalas carries over to the compiler output: the generated CakeML code never aborts with a runtime error, and – by the CakeML compiler correctness theorem – neither does the machine code (unless it runs out of memory).

As a convenient byproduct of CakeML's FFI modelcode is parameterised on primitives for sending and receiving messages, making it communication-backend agnostic. Thus, the same CakeML code can be used irrespective of whether the communication happens via (say) TCP/IP, MPI, or IPC, as long as these actions have the same semantics as the corresponding Payload primitives. Like other choreographic languages, our deadlock freedom guarantee depends on the rather strong assumptions implicit in the operational semantics: the backend stays live, and messages will never be lost in transit. In practice, our theorems are only as good as the backend's ability to abide by this.

As a proof-of-concept of our approach, we implemented a filter choreography and executed the generated code using an IPC communication backend on seL4 [17], a formally verified operating systems microkernel. Hence there is strong evidence, in the form of machine-checked proofs of functional correctness of the kernel [17] and the delivery guarantees of the component platform [9], that this backend is up to the task, even though we do not connect our proofs with the seL4 proofs.

This paper's main contributions are:

- the definition and verification of an end-to-end choreographic compiler, including:
- the proof of *endpoint projection*'s correctness w.r.t. Kalas's asynchronous semantics; and
- the implementation of a proof-of-concept choreography on top of seL4/CAmkES.

All definitions and proofs in this paper are mechanised in HOL4 [24] and available online.[1]

[1] `https://github.com/CakeML/choreo/`.

2 Kalas: A Choreographic Language

In this section we introduce our choreographic language, Kalas. To build an intuition for how choreographies operate, consider a common situation in component-based systems: a producer wishes to send a stream of messages to a consumer, but the consumer can only receive messages of a certain form. A filter that discards malformed messages is inserted.

▶ **Example 1** (Message filter - Choreography).

```
1.  while(true) do
2.    let v@producer = next_msg() in
3.    producer.v -> filter.temp;
4.    let test@filter = test(temp) in
5.    if test@filter then
6.      filter -> consumer[T];
7.      filter.temp -> consumer.v
8.    else filter -> consumer[F]
```

We assume a function `next_msg` to obtain the next message, which is then stored in the `producer`'s local variable v (line 2). The `producer` then communicates the contents of v to the `filter` which stores it locally in *temp* (line 3). The `filter` computes `test`(*temp*) (line 4). If `test`(*temp*) is true (line 5), we inform the consumer that a message is coming (line 6), and forwards the contents of *temp* to the `consumer` (line 7). Otherwise, we inform the `consumer` that a message was dropped (line 8).

This example highlights two important features: a choreography captures both the concrete behaviour of its participants and a global view of the communication occurring between them. That is, interactions between endpoints are presented together with local computation, e.g., `test` above. This allows individual endpoints to be translated into complete sequential programs. Second, communication mismatches are impossible by construction: if no message is forthcoming, the `consumer` will never be stuck waiting for one.

2.1 Syntax and Semantics

Kalas is similar to Core Choreographies (CC) [5], but features arbitrary local computation and asynchronous communication. The main datatype under consideration in our choreography language is strings or, to be precise, finite sequences of bytes. Strings are used as endpoint names (p_i), variable names (v_i), process variables (X), and as the concrete data that gets bound to variables and transmitted between endpoints (d). The use of strings as the value represents a separation of concerns: after marshalling and unmarshalling, local computations have full access to HOL's strongly typed language, but the choreography language is only concerned with data as it is really transmitted, namely as strings. The booleans (ranged over by b) are written T and F. When we use booleans where strings are expected, we tacitly identify T with [0x01], and F with [0x00]. We use a to range over the union of strings and booleans, and f to range over functions of type $\mathsf{string}^* \rightarrow \mathsf{string}$.

▶ **Definition 2** (Kalas syntax). *Choreographies in Kalas, ranged over by C, are inductively defined by the grammar*

$$
\begin{array}{lllll}
C & ::= & p_1.v_1 \twoheadrightarrow p_2.v_2; C & \textit{(com)} & p_1 \twoheadrightarrow p_2[b]; C \quad \textit{(sel)} \\
 & & \textbf{if } v@p \textbf{ then } C_1 \textbf{ else } C_2 & \textit{(if)} & \textbf{let } v@p = f(\tilde{v}) \textbf{ in } C \quad \textit{(let)} \\
 & & \mu X.\ C & \textit{(fix)} & X \quad \textit{(var)} \\
 & & \mathbf{0} & \textit{(nil)} &
\end{array}
$$

Table 1 Kalas semantics: communication rules. The function $\mathrm{wv}(\alpha)$ returns the variable (if any) that is modified by α.

$$\textsc{Com}\ \frac{s\,(v_1, p_1) \;=\; d \quad p_1 \;\neq\; p_2}{s \rhd p_1.v_1 \twoheadrightarrow p_2.v_2;\, C \xrightarrow[\epsilon]{p_1.v_1 \twoheadrightarrow p_2.v_2} s[(v_2, p_2) \;:=\; d] \rhd C}$$

$$\textsc{Com-S}\ \frac{s \rhd C \xrightarrow[l]{\alpha} s' \rhd C' \quad p_1 \notin \mathrm{fp}(\alpha) \quad p_2 \notin \mathrm{fp}(\alpha)}{s \rhd p_1.v_1 \twoheadrightarrow p_2.v_2;\, C \xrightarrow[l]{\alpha} s' \rhd p_1.v_1 \twoheadrightarrow p_2.v_2;\, C'}$$

$$\textsc{Com-A}\ \frac{s \rhd C \xrightarrow[l]{\alpha} s' \rhd C' \quad p_1 \in \mathrm{fp}(\alpha) \quad \mathrm{wv}(\alpha) \neq (v_1, p_1) \quad p_2 \notin \mathrm{fp}(\alpha)}{s \rhd p_1.v_1 \twoheadrightarrow p_2.v_2;\, C \xrightarrow[(p_1.v_1 \twoheadrightarrow p_2.v_2)::l]{\alpha} s' \rhd p_1.v_1 \twoheadrightarrow p_2.v_2;\, C'}$$

The prefix *(com)* sends the data bound to variable v_1 at endpoint p_1 to endpoint p_2 which stores it in variable v_2, *(sel)* communicates the selection of a branch from p_1 to p_2, *(if)* branches over the value in variable v at process p, and *(nil)* is the empty choreography. *(let)* performs local computation, taking all values bound to the variables $\widetilde{v}$ at endpoint p and applies them as arguments to the function f. The result is then stored in v. Note that we do not commit to any particular syntax for functions; rather, f is a function in the meta-language in which Kalas is defined. In our case, the meta-language is higher-order logic (Hol). Hence our syntax is only concerned with interaction and branching of endpoints, offloading computation to Hol. This flexibility is useful for specifying open systems, or systems with legacy components: the internal behaviour of an endpoint that we have no control over can be modelled by functions that are non-computable, underspecified, or even completely uninterpreted, and the compiler can ignore such endpoints for code generation. For endpoints that we do intend to project, we require that the f's used in their let-bindings be "sufficiently code-like" – otherwise, code generation will fail. This excludes, for example, functions that use Hilbert choice, sets or quantifiers.

Finally, *(fix)* supports choreographies with infinite behaviour. These can be unfolded, taking e.g. $\mu X.\ p_1 \twoheadrightarrow p_2[b]; X$ to $p_1 \twoheadrightarrow p_2[b]; \mu X.\ p_1 \twoheadrightarrow p_2[b]; X$. The above example also illustrates the only use of *(var)*: as a placeholder for fixpoint unfolding. The **while** loop used in Example 1 is syntactic sugar for *(fix)*.

We will use $\mathrm{fv}(C)$ to refer to the free variables of a choreography C, where each variable is paired with the name of the process that owns the variable. The binding operators are **let** and $p_1.v_1 \twoheadrightarrow p_2.v_2; C$, where $(v1, p1)$ is considered free and $(v2, p2)$ is considered bound.

The operational semantics is inductively defined, with some sample rules given in Table 1. Transitions are labelled to indicate both the action being performed (upper α), and the trace (lower l) of deferred asynchronous actions. We explain the latter mechanism below. We refer to both labels and prefixes as actions, since they directly correspond to all operations that can be performed in the language. A *store* s is a partial function $\mathsf{string} \times \mathsf{string} \hookrightarrow \mathsf{string}$ representing a global view of the endpoints' variable binding environment: $s(v, p)$, if defined, denotes the value bound to v in p's binding environment. In Hol, we use option types in the range to encode this partiality; much of the following presentation elides the logic's special handling of this (e.g., the Some and None constructors).

Kalas uses non-blocking, asynchronous communication. Hence, a sender process should be able to perform further actions before the message has arrived at the receiver. The semantics captures this by allowing an action α to occur before other interactions, provided only the

sender process is present in α. A trace of every action that was skipped over is kept, to ensure the consistency between asynchrony and concurrency rules. This trace is used in the rule for **if**, which requires that both branches defer the same actions, though not necessarily in the same order. This constraint guarantees that regardless of the choice of branch, the asynchronous actions that need to be deferred in order to perform α are the same for each of the processes involved, implying that α is independent of the branching caused by the guard.

We prove that the resulting semantics is locally confluent, which will turn out to be immensely important for taming the proofs. As a sanity check of our rather involved labels, we also show completeness with respect to a similar semantics (not shown here) with structural congruence instead of swapping rules.

3 Endpoint Projection

The first phase of our compiler is *endpoint projection*, where we translate Kalas into ENDPOINT, our first intermediate language.

Continuing with Example 1, when we apply endpoint projection to the producer-consumer-filter choreography (PCF), we obtain $[\![PCF]\!]_{\mathsf{E}} = P \mid C \mid F$, comprised of the following endpoints running in parallel:

▶ **Example 3** (Message filter – ENDPOINT).

P (Producer)

$\mu X.$
  let $v =$ `next_msg()` **in**
  send v **to** `filter`. X

C (Consumer)

$\mu X.$
  `filter` **chooses**
    T : **receive** v **from** `filter`. X
    or
    F : X

F (Filter)

$\mu X.$
  receive msg **from** `producer`.
  let $v =$ `test`(msg) **in**
  if v **then**
    choose T **for** `consumer`.
    send msg **to** `consumer`. X
  else choose F **for** `consumer`. X

Here the (Filter) endpoint receives a message from the producer and, depending on the output of `test`, communicates its choice of branch to the consumer. Conversely, the (Consumer) decides based on the filter's choice whether it should await a message or whether the message was dropped. Finally, the (producer) obtains a message and sends it to the filter. Note that no branching or choice is required in the producer, since it behaves the same whether `test` succeeds or not.

3.1 Endpoint: Syntax and Semantics

ENDPOINT inherits many design decisions from Kalas, but splits unitary Kalas systems into two layers: the endpoint layer is purely sequential, and the *network* layer is a parallel composition of endpoints, each with its own name, queue and binding environment.

A *queue* q is a function $\mathsf{string} \to \mathsf{string}^*$. The value $q(p)$ is the sequence of messages, from first to last, received from *process* p but not yet read. Let $q + (p, a)$ be q with a appended to the end of $q(p)$, and $q - p$ be q with the first element of $q(p)$ removed; if $q(p)$ is empty, $q - p$ is undefined. An *environment* e is a partial function from variable names to values.

Table 2 Endpoint semantics: communication rules.

$$\textsc{Send}\ \frac{e\ v\ =\ d \quad p_1\ \neq\ p_2}{(p_1, e, q) \triangleright \mathbf{send}\ v\ \mathbf{to}\ p_2.P\ \xrightarrow{p_1 \to p_2 : d}\ (p_1, e, q) \triangleright P}$$

$$\textsc{Enqueue}\ \frac{p_1\ \neq\ p_2}{(p_2, e, q) \triangleright P\ \xrightarrow{p_2 \leftarrow p_1 : d}\ (p_2, e, q\ +\ (p_1, d)) \triangleright P}$$

$$\textsc{Dequeue}\ \frac{q(p_2)\ =\ d :: \widetilde{a} \quad p_1\ \neq\ p_2}{(p_1, e, q) \triangleright \mathbf{receive}\ v\ \mathbf{from}\ p_2.P\ \xrightarrow{\tau}\ (p_1, e[v\ :=\ d], q\ -\ p_2) \triangleright P}$$

▶ **Definition 4** (Endpoint syntax)**.**

P, Q	$:=$	$\mathbf{send}\ v\ \mathbf{to}\ p.P$	*(output)*	$\mathbf{let}\ v = f(\widetilde{v})\ \mathbf{in}\ P$	*(let)*
		$\mathbf{receive}\ v\ \mathbf{from}\ p.P$	*(input)*	$\mu X.P$	*(fix)*
		$\mathbf{choose}\ b\ \mathbf{for}\ p.P$	*(internal choice)*	X	*(var)*
		$p\ \mathbf{chooses}\ \mathsf{T}\ :\ P\ \mathbf{or}\ \mathsf{F}\ :\ Q$	*(external choice)*	$\mathbf{0}$	*(nil)*
		$\mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q$	*(if)*		
N	$:=$	$N_1 \mid N_2$	*(parallel)*	$\mathbf{0}$	*(nil)*
		$(p, e, q) \triangleright P$	*(endpoint)*		

Table 2 shows three representative rules from Endpoint's operational semantics. $\mathbf{send}\ v\ \mathbf{to}\ p.P$ represents an endpoint ready to send the contents of variable v to p, using the Send rule; the Enqueue rule allows a message thus sent to arrive in p's queue. $\mathbf{receive}\ v\ \mathbf{from}\ p.P$ denotes a process ready to dequeue a message from its queue originating from p, and bind the contents of the message to the variable v (Dequeue); if there is no message from p, the endpoint is blocked until one arrives. Similarly, $\mathbf{choose}\ b\ \mathbf{for}\ p.P$ represents an endpoint ready to tell process p that it has chosen the b-branch. The corresponding IntChoice rule (elided) interacts with Enqueue to add the choice to b's message queue. $p\ \mathbf{chooses}\ \mathsf{T}\ :\ P\ \mathbf{or}\ \mathsf{F}\ :\ Q$ represents a process waiting for p to communicate its choice of branch. If it finds a T from p in the queue, it proceeds as P; if it finds something else from p, it proceeds as Q.

3.2 Endpoint projection

The main complication when defining endpoint projection is how to handle **if** statements, which are not always projectable. For an example, consider the choreography

$$\mathbf{if}\ \text{Alice}@v\ \mathbf{then}\ \text{Bob}.v \twoheadrightarrow \text{Alice}.v\ \mathbf{else}\ \text{Alice}.v \twoheadrightarrow \text{Bob}.v$$

where Alice makes an internal choice, and depending on the result, either Alice sends a message to Bob, or vice versa. How does Bob know whether to send or receive?

We need a projectability criterion that rules out such degenerate cases. Our criterion is, intuitively: whenever Alice chooses an **if** branch, every other endpoint whose projection depends on the choice must immediately be told which branch was chosen. Hence, the example above can be made projectable by adding selections as follows:

$$\begin{aligned}&\mathbf{if}\ \text{Alice}@v\ \mathbf{then}\ \text{Alice} \twoheadrightarrow \text{Bob}[\mathsf{T}]; \text{Bob}.v \twoheadrightarrow \text{Alice}.v\\ &\mathbf{else}\ \text{Alice} \twoheadrightarrow \text{Bob}[\mathsf{F}]; \text{Alice}.v \twoheadrightarrow \text{Bob}.v\end{aligned}$$

Table 3 Projection and projectability, with the *(sel)* case, which is similar to *(com)*, elided. Partiality is indicated with a result of $\bot$. Recursive calls (e.g., in the let case) that fail propagate that undefinedness to the top-level.

$$\mathsf{pr}_p(\gamma, \mathbf{0}) = \mathbf{0}$$

$$\mathsf{pr}_p(\gamma, p_1.v_1 \twoheadrightarrow p_2.v_2; C) = \begin{cases} \bot & \text{if } p_1 = p_2 = p \\ \mathbf{send}\, v_1\, \mathbf{to}\, p_2.\mathsf{pr}_p(\gamma, C) & \text{if } p = p_1 \neq p_2 \\ \mathbf{receive}\, v_2\, \mathbf{from}\, p_1.\mathsf{pr}_p(\gamma, C) & \text{if } p \neq p_1 = p_2 \\ \mathsf{pr}_p(\gamma, C) & \text{otherwise} \end{cases}$$

$$\mathsf{pr}_p(\gamma, \mathbf{let}\ v@p_1 = f(\widetilde{v})\ \mathbf{in}\ C) = \begin{cases} \mathbf{let}\ v = f(\widetilde{v})\ \mathbf{in}\ \mathsf{pr}_p(\gamma, C) & \text{if } p = p_1 \\ \mathsf{pr}_p(\gamma, C) & \text{otherwise} \end{cases}$$

$$\mathsf{pr}_p(\gamma, \mu X.C) = \begin{cases} \mu X.\mathsf{pr}_p(\gamma[X := \mathsf{procs}(C)], C) & \text{if } p \in \mathsf{procs}(C) \\ \mathbf{0} & \text{otherwise} \end{cases}$$

$$\mathsf{pr}_p(\gamma, X) = \begin{cases} \bot & \text{if } X \notin \mathrm{dom}(\gamma) \\ X & \text{if } p \in \gamma(X) \\ \mathbf{0} & \text{otherwise} \end{cases}$$

$$\mathsf{pr}_p(\gamma, \mathbf{if}\ v@p_1\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2)$$

$$= \begin{cases} \mathbf{if}\ v\ \mathbf{then}\ \mathsf{pr}_p(\gamma, C_1)\ \mathbf{else}\ \mathsf{pr}_p(\gamma, C_2) & \text{if } p = p_1 \\ p_1\ \mathbf{chooses}\ \mathsf{T} : \mathsf{pr}_p(\gamma, C_1')\ \mathbf{or}\ \mathsf{F} : \mathsf{pr}_p(\gamma, C_2') & \text{if } p \neq p_1 \text{ and } \mathsf{sp}_{p_1,p}(C_1) = (\mathsf{T}, C_1') \\ & \text{and } \mathsf{sp}_{p_1,p}(C_2) = (\mathsf{F}, C_2') \\ \mathsf{pr}_p(\gamma, C_1) & \text{if } p \neq p_1 \text{ and } \mathsf{pr}_p(\gamma, C_1) = \mathsf{pr}_p(\gamma, C_2) \\ \bot & \text{otherwise} \end{cases}$$

To formalise this criterion, we use the auxiliary function sp to split off initial selections pertaining to a pair of endpoints and check which branch was chosen.

▶ **Definition 5** (Split selections). *The partial function* sp *is inductively defined as follows (in all other cases,* sp *is undefined)*

$$\mathsf{sp}_{p_1,p_2}(p_3 \twoheadrightarrow p_4[b]; C) = \begin{cases} (b, C) & \textit{if } p_1 = p_3 \textit{ and } p_2 = p_4 \\ \mathsf{sp}_{p_1,p_2}(C) & \textit{if } p_1 = p_3 \textit{ and } p_2 \neq p_4 \\ \text{undefined} & \textit{otherwise} \end{cases}$$

Fixpoints motivate some additional projectability criteria: (i) orphan *(var)* statements are not allowed, and (ii) the projection of a *(fix)* statement for endpoints that do not appear in its body should be $\mathbf{0}$ (otherwise, the compiler introduces divergence). To enforce these requirements, we use a *fixpoint context* γ, a partial function from process names to sets of endpoint names that keeps track of which endpoints occur in the body of each *(fix)* statement.

We define a single partial function pr that given an endpoint name, a fixpoint context, and a choreography, returns an endpoint (its projection), if it exists.

▶ **Definition 6** (Projection and projectability). *A choreography C is* projectable *if for all $p \in \mathsf{procs}(C)$, $\mathsf{pr}_p(\epsilon, C)$ is defined. The* projection *of the endpoints $\widetilde{p}$ from a choreography C with state s is defined as $[\![s \triangleright C]\!]_{\mathsf{E}}^{\widetilde{p}} = \Pi_{p_i \in \widetilde{p}}.\ (p_i, s\downarrow_{p_i}, \epsilon) \triangleright \mathsf{pr}_{p_i}(\epsilon, C)$ where Π denotes iterated parallel composition, $s\downarrow_p$ denotes $\lambda v.s(p, v)$, ϵ is an empty fixpoint context, and* pr *is defined inductively by the equations in Table 3. $[\![s \triangleright C]\!]_{\mathsf{E}}$ abbreviates $[\![s \triangleright C]\!]_{\mathsf{E}}^{\mathsf{procs}(C)}$.*

4 Refining Choice

In Phase II, we implement Endpoint's choice primitives using send and receive actions. This simplifies reasoning about later compilation phases and the implementation of communication backends, which only need to consider two message-passing primitives instead of four.

After refining choice from the parallel composition $P \mid C \mid F$ in Example 3, we obtain $[\![P \mid C \mid F]\!]_{\mathsf{C}} = P \mid C' \mid F'$, where the producer P is unchanged because it uses no choice constructs. The filter and consumer are compiled as follows:

▶ **Example 7** (Message filter – Refining choice)**.**

F' (Filter)	C' (Consumer)
$\mu X.$ **receive** msg **from** `producer`. **let** $test = \texttt{test}(msg)$ **in** **if** $test$ **then let** $v = \mathsf{T}$ **in** **send** v **to** `consumer`. **send** msg **to** `consumer`. X **else let** $v = \mathsf{F}$ **in** **send** v **to** `consumer`. X	$\mu X.$ **receive** v **from** `filter`. **if** v **then** **receive** v **from** `filter`. X **else** X

The phase is mostly straightforward: internal choice is encoded as sending a boolean value, and external choice is encoded as receiving a value, storing it in a temporary variable v, then branching on it using **if**.

▶ **Definition 8** (Phase II)**.** *The compilation function is homomorphic on all operators except internal and external choice, where it is defined as follows for any v not free in P, Q:*

$$[\![\textbf{choose}\, b\, \textbf{for}\, p.P]\!]_{\mathsf{C}} = \textbf{let}\ v = (\lambda x.b)\epsilon\ \textbf{in}\ \textbf{send}\, v\, \textbf{to}\, p.[\![P]\!]_{\mathsf{C}}$$
$$[\![p\, \textbf{chooses}\, \mathsf{T} :\ P\, \textbf{or}\, \mathsf{F} : Q]\!]_{\mathsf{C}} = \textbf{receive}\, v\, \textbf{from}\, p.(\textbf{if}\ v\ \textbf{then}\ [\![P]\!]_{\mathsf{C}}\ \textbf{else}\ [\![Q]\!]_{\mathsf{C}})$$

Since v is not used further in the continuation, it can be reused for subsequent choice encodings, meaning that in practice, a single fresh name suffices.

The design of the Endpoint semantics anticipates this compilation phase, by allowing type confusion between boolean values and string values. This feature, which may seem otherwise undesirable, makes branch selection messages indistinguishable from other messages. This makes the difference between N_{E} and $[\![N_{\mathsf{E}}]\!]_{\mathsf{C}}$ unobservable by other processes.

5 Splitting Large Messages

In both Endpoint and our source language, transmitting a message of arbitrary size is a single atomic operation, whether it carries one bit or one terabyte of information. This is convenient for the programmer, but doesn't reflect how real communication protocols work.

Our second compiler phase introduces a protocol that divides long messages into chunks (see Figure 2), accounting for the fact that real communication protocols have bounds on message size, without burdening the application programmer with the details.

For these purposes, we introduce another intermediate representation, Payload, which is similar to Endpoint except messages have a fixed size. It turns out that this compiler phase is more proof-relevant than compiler implementation-relevant. Syntactically, the compilation function $[\![\cdot]\!]_{\mathsf{C}}$ from Endpoint to Payload is essentially the identity function. Semantically, send and receive actions are no longer atomic, which leads to a combinatorial explosion in the number of possible interleavings. The associated proof complications are largely mitigated by observing that the target terms are always locally confluent.

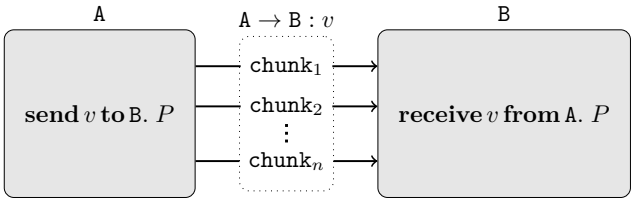


Figure 2 Messages split into `chunks`.

PAYLOAD is parameterised by a *payload size* $\sigma > 0$. Unlike in previous languages where messages can have arbitrary size, here messages in transit must be exactly $\sigma + 1$ bytes long. Longer messages are transmitted in chunks, and shorter messages are padded; the extra byte encodes the bookkeeping necessary to realise this. In particular, we must track whether a given chunk ends a message, or whether it will be continued in future messages.

▶ **Definition 9** (Payload syntax, I). *The syntax of* PAYLOAD *is obtained by removing endpoint, input, output and choice from* ENDPOINT*, and adding:*

$$(p, e, \eta, q) \triangleright P \qquad \textit{(endpoint)}$$
$$\textbf{send}\ v_n\ \textbf{to}\ p.P \qquad \textit{(output)}$$
$$\textbf{receive}\ v\ \textbf{in}\ \langle d \rangle\ \textbf{from}\ p.P \qquad \textit{(input)}$$

The message-splitting aspect of PAYLOAD's semantics is embodied here: the new input and output prefixes record how far along in a transmission we are. Hence **send** v_n **to** $p.P$ will send the value of v to p, starting from the n-th byte, divided into as many chunks as necessary, one chunk at a time. Similarly **receive** v **in** $\langle d \rangle$ **from** $p.P$ will receive chunks from p, recording every intermediate chunk in the temporary buffer d. When a final chunk arrives, all received chunks are concatenated and bound to the variable v.

The state component η is a closure environment; we will discuss it in Section 6.1, where the operators that need it are introduced. For space reasons, readers interested in the operational semantics are referred to the formalisation.

6 Introducing Closures

Finally, before we transition from PAYLOAD to CakeML, we introduce closures. That is, we translate all instances of the fixpoint operator μ into a **letrec** primitive, to better match CakeML's representation of recursive functions. Though CakeML supports global, updateable variables (SML's `ref` types), reasoning is much simpler if one remains "purely functional", and uses parameters with closures and local, immutable bindings. Thus: variables written to in the fixpoint body (ultimately from the Kalas source) must be made function parameters.

Why not just put **letrec** in the source language, if we have to add it later anyway? Briefly, it becomes technically complicated to maintain a consistent view of the global environment in the presence of out-of-order execution. Fixpoint semantics do not need environments. See Section 9 for a comparison with related approaches.

6.1 Closures: syntax and semantics

Recall from Section 5 that endpoint states contain a closure environment η. It is a mapping from function names to *closures*. Closures are triples $(e, \eta, \lambda\widetilde{x}.P)$, where: e, η are the local variable environments and closure environments, respectively; $\widetilde{x}$ is the function's parameters; and P is the function body. Note that closures and environments are mutually recursive.

▶ **Definition 10** (Payload syntax, II). *These augment the operators from Definition 9.*

$\mathbf{letrec}\ F(\widetilde{d}) = P \quad \textit{(letrec)} F(\widetilde{v}) \quad \textit{(call)}$

(letrec) and *(call)* are function definitions and function calls, respectively. They are similar to *(fix)*, but use environment semantics instead of substitution semantics. Note that **letrec** has no continuation; instead, the defined function will be called immediately. This suffices for our purposes, which is to use **letrec** to encode μ.

6.2 Compilation

To compile the fixpoint operator into the letrec operator, the basic idea is the obvious one: a fixpoint binder μX becomes a recursive function definition, and a process variable X becomes a function call. But what arguments should we give the function?

To prepare for compilation to CakeML, the main goal is to make sure we use the constructs that mutate variables (let and input) consistently with functional programming idioms. But we have a second, conflicting goal. To simplify proofs, we want the local variable environment of target terms to be identical to the global environment of their source terms. The following table illustrates the options we considered:

Source

$\mathbf{let}\ x, y = \ldots\ \mathbf{in}\ \ \mu X.\ \mathbf{let}\ x = \mathtt{f}(x, y)\ \mathbf{in}\ \ \mathbf{send}\, x\, \mathbf{to}\, \mathtt{p}.\mathbf{receive}\, z\, \mathbf{from}\, \mathtt{p}.X$

Target I	Target II	Target III
$\mathbf{let}\ x, y = \ldots\ \mathbf{in}$ $\ \mathbf{letrec}\ X(x, y, z) =$ $\ \ \mathbf{let}\ x = \mathtt{f}(x, y)\ \mathbf{in}$ $\ \ \ \mathbf{send}\, x\, \mathbf{to}\, \mathtt{p}.$ $\ \ \ \mathbf{receive}\, z\, \mathbf{from}\, \mathtt{p}.$ $\ \ \ X(x, y, z)$	$\mathbf{let}\ x, y = \ldots\ \mathbf{in}$ $\ \mathbf{letrec}\ X(x, z) =$ $\ \ \mathbf{let}\ x = \mathtt{f}(x, y)\ \mathbf{in}$ $\ \ \ \mathbf{send}\, x\, \mathbf{to}\, \mathtt{p}.$ $\ \ \ \mathbf{receive}\, z\, \mathbf{from}\, \mathtt{p}.$ $\ \ \ X(x, z)$	$\mathbf{let}\ x, y = \ldots\ \mathbf{in}$ $\ \mathbf{letrec}\ X(x) =$ $\ \ \mathbf{let}\ x = \mathtt{f}(x, y)\ \mathbf{in}$ $\ \ \ \mathbf{send}\, x\, \mathbf{to}\, \mathtt{p}.$ $\ \ \ \mathbf{receive}\, z\, \mathbf{from}\, \mathtt{p}.$ $\ \ \ X(x)$

Target I represents the simplest compilation strategy that could possibly work: every program variable becomes a parameter of every function. This, however, is rather wasteful: y is never modified within the function, so there's no need to pass it around; z is modified in the body, but not subsequently read, so there is no need to remember its value between calls.

Taking all this into account would yield Target III, which is to take as function parameters only those variables that may be read before they are written to within a fixpoint's body. Unfortunately, this is not compatible with our second goal above: while the source term retains the value of z between subsequent fixpoint unfoldings, Target III will restore z to its value at the point of X's definition at each recursive call.

As a compromise, we opt for Target II: the function parameters are the variables that may be written to within the body of the fixpoint expression. To this end, we let the function $\mathrm{wv}(e)$ return all variables that are modified (by **let** or receive) in e.

▶ **Definition 11** (Phase IV). *The compilation function $[\![\cdot]\!]^{\gamma}_{\mathsf{F}}$ is homomorphic on all operators except: letrec and call, where it is undefined; and fix and val, where it is*

$$[\![\mu X.P]\!]^{\gamma}_{\mathsf{F}} = \mathbf{letrec}\ X(\mathrm{wv}(P)) = [\![P]\!]^{\gamma[X:=\mathrm{wv}(P)]}_{\mathsf{F}}\ [\![X]\!]^{\gamma}_{\mathsf{F}} = X(\gamma(X))$$

In the above, γ is a partial function from process variables to lists of local variables.

A further minor complication is that a variable can be used as a function argument before its definition. We could add support for optional arguments, but since CakeML has no such feature we would eventually have to compile them away. Our fix is that before we apply $[\![X]\!]_{\mathsf{F}}$, we add a prelude to each endpoint that initialises all variables to a default value.

7 Compiler Correctness

In this section, we discuss the compiler correctness theorem connecting Kalas to PAYLOAD with Letrec, and its proof.

7.1 Theorem Statement

Let $[\![\cdot]\!]^{\widetilde{p}}$ denote the composition $[\![\cdot]\!]^{p}_{\mathsf{E}} \circ [\![\cdot]\!]_{\mathsf{C}} \circ [\![\cdot]\!]_{\mathsf{P}} \circ [\![\cdot]\!]^{\epsilon}_{\mathsf{F}}$ and let $[\![C]\!] = [\![C]\!]^{\mathsf{procs}(C)}$. We prove weak operational correspondence up-to strong bisimilarity (denoted $\dot{\sim}$) for $[\![\cdot]\!]^{\widetilde{p}}$:

▶ **Theorem 12.** *If c is a projectable choreography and $\mathrm{fv}(c) \subseteq \mathrm{dom}(s)$, then*

1. *(Operational completeness) If $s \triangleright C \implies s' \triangleright C'$ then there exist s'', C'', N_{F} such that $s' \triangleright C' \implies s'' \triangleright C''$ and $[\![s \triangleright C]\!] \implies N_{\mathsf{F}}$ and $N_{\mathsf{F}} \dot{\sim} [\![s'' \triangleright C'']\!]^{\mathsf{procs}(C)}$*
2. *(Operational soundness) If $[\![s \triangleright C]\!] \implies N_{\mathsf{F}}$ then there exist s', C', N'_{F} such that $N_{\mathsf{F}} \implies N'_{\mathsf{F}}$ and $s \triangleright C \implies s' \triangleright C'$ and $N'_{\mathsf{F}} \dot{\sim} [\![s' \triangleright C']\!]^{\mathsf{procs}(C)}$*

Here $\implies$ over networks denotes $\xrightarrow{\tau}{}^{\star}$, and $\implies$ over choreographies denotes $(\bigcup_{a,l} \xrightarrow[l]{a})^{\star}$. Our presentation of *operational completeness* requires a catch-up transition because projectability is not, in general, preserved by reduction. However, any non-projectable choreography reachable from a projectable choreography can always reduce to a projectable choreography.

One important consequence of Theorem 12 is that the compiler output is deadlock-free:

▶ **Theorem 13** (Network-level deadlock-freedom)**.** *If C is a projectable choreography, and $\mathrm{fv}(C) \subseteq \mathrm{dom}(s)$, and $[\![s \triangleright C]\!] \implies N_{\mathsf{F}}$, then either all endpoints in N_{F} are* Nil*, or there exists N'_{F} such that $N_{\mathsf{F}} \xrightarrow{\tau} N'_{\mathsf{F}}$*

7.2 On the proofs

As the reader may expect, we prove soundness and completeness separately for each compilation phase before composing the theorems. A common theme is strategic use of confluence to reduce the number of interleavings we must consider.

The proof of *operational completeness* for Phase I leverages local confluence to simplify reasoning in a major way. The asynchrony and swapping rules in Kalas's semantics, which are otherwise a pain point, play no role in these proofs. This is because any reduction involving them has a common successor with a reduction that only uses the syntax-directed rules (e.g., rule COM from Table 1). This yields a simpler proof than, for example, Montesi [20, Appendix C]; his language is also confluent, yet his proof makes no use of this, and includes cases for the swapping and asynchrony rules.

To prove *operational soundness* we use a technique based on *inert reduction*, first conceived by van Glabbeek to study encodings from the synchronous to the asynchronous π-calculus [28]. Intuitively, an inert reduction is one that performs a bookkeeping step without committing to a branch. We say that $N_{\mathsf{E}} \longrightarrow N'_{\mathsf{E}}$ is *inert* if for every $N''_{\mathsf{E}} \neq N'_{\mathsf{E}}$ such that $N_{\mathsf{E}} \longrightarrow N''_{\mathsf{E}}$, there is an N'''_{E} such that $N'_{\mathsf{E}} \longrightarrow N'''_{\mathsf{E}}$ and there is an inert transition $N''_{\mathsf{E}} \longrightarrow N'''_{\mathsf{E}}$. The key insight is that for encodings that only use inert catch-up transitions, operational soundness can be proven by induction on the length of the reduction sequence. Moreover, since inertness is a form of confluence, it suffices to consider just one interleaving of the intermediate steps, namely the one that directly mimics one source-language step at a time. All our catch-up

transitions are inert, which makes the proof of *operational soundness* much more tractable, with roughly half the effort going into proving confluence. The same technique is also used to great effect to tame the interleaving explosion of Phase III.

Phase II uses a traditional invariant-based technique, which we found intractable for the other phases with more complicated interleavings. The main headache here is alpha-equivalence considerations arising from the need to invent fresh names.

The proofs for Phase IV are different. In the other phases, the bulk of the effort is chasing transitions. Here, that part is trivial since we have one-to-one transition correspondence (up-to strong bisimilarity). The difficulty is in wrangling the candidate relation used to prove that the continuations of fixpoints and **letrec** unfoldings are bisimilar. The relation, which describes the precise relationship between closure environments and (possibly unfolded) fixpoints, is surprisingly complicated at almost 50 lines of HOL4 script. It is worth pointing out that this complicated relation entails no trust issues; its only use in the overall proof story is to witness an existential quantifier.

8 Compilation into CakeML

CakeML [18] is an impure, sequential, functional programming language similar to Standard ML. Its most notable feature is a compiler correctness proof in HOL4 that extends down to the machine code level for mainstream architectures such as x86-64 and ARM [25]. Interaction with the outside world is supported by a foreign function interface (FFI). We assume two foreign functions, `send` and `receive`, that support communication with the other endpoints. Compilation to CakeML consists of two parts: the static part, which is verified once and for all, and the dynamic part, which is proof-producing.

8.1 Static compiler

The static compilation is performed by the function $[\![\cdot]\!]_{\mathsf{ML}}$, which maps PAYLOAD endpoints to CakeML expressions. Its full definition would not fit here, but to show its flavour, $[\![\mathbf{receive}\, v\, \mathbf{in}\, \langle\epsilon\rangle\, \mathbf{from}\, p.P]\!]_{\mathsf{ML}}$ produces the code

```
let val v =
  let val buff = Word8Array.array (σ + 1) 0
      fun receiveloop d =
        (#(receive) p buff;
         let val m = unpad buff
         in if final buff then concat(reverse(m::d))
            else let fun zerobuf(i) =
                         if i < 0 then ()
                         else (Word8Array.update(buff,i,0); zerobuf(i-1))
                 in zerobuf(Word8Array.length(buff)-1);
                    receiveloop(m::d)
                 end
         end)
  in receiveloop [] end
in [P]_ML end
```

First, a receive buffer of size $\sigma+1$ is allocated. Then, the function `receiveloop` repeatedly calls the foreign function `#(receive)` until a final chunk from p is received, zeroing the receive buffer between every message. All chunks of the message are unpadded, concatenated and finally bound to the variable v before proceeding.

We use a small-step, relational (but deterministic) presentation of CakeML's semantics, allowing a natural expression of our eventual simulation theorem. We write $(p_0, cs_0) \rightarrow_c (p, cs)$, with p_0 the initial CakeML program, and cs_0 its accompanying state, to mean that this pair can evolve in a single step to (p, cs). The states cs_i contain FFI information (see below), the internal program state (variable environment, reference contents), as well as a continuation stack to track what remains to be done. The semantics is parametric on the behaviour of foreign functions: states include a freely chosen model of the outside world, and a freely chosen *oracle function* that describes how this model reacts to FFI calls.

We are interested in how generated CakeML code interacts with the choreography's other endpoints, so our FFI state models the outside world as triple (p, q, N), with p the name of the CakeML endpoint, q its queue, and N a PAYLOAD network that p interacts with. There is an unfortunate mismatch here: the FFI model must be a function (CakeML is deterministic), but PAYLOAD's semantics is a one-to-many relation: when we receive a message from N, there is not in general a unique N' that the network will reach after sending us our message, as actions internal to N may or may not fire before N sends the message. However, as long as all endpoints in N have unique names (a reasonable invariant), PAYLOAD reductions and send actions are locally confluent. So whether such internal actions fired or not, the resulting states are observationally equivalent from p's point of view.

Let $N \xRightarrow{p \rightarrow \widetilde{p}:\widetilde{d}} N'$ denote $N \Longrightarrow \xrightarrow{p \rightarrow p_0:d_0} \Longrightarrow \ldots \xrightarrow{p \rightarrow p_n:d_n} \Longrightarrow N'$. We define the oracle so that when `#(send)` $p\,d$ executes in a state (p_1, q, N), if there is no endpoint named p in N, we abort with a run-time error; otherwise we produce a new state $(p_1, q + \widetilde{(p, d)} + \widetilde{(p', d')}, N')$, chosen with Hilbert Choice to satisfy $N \xRightarrow{\widetilde{p} \rightarrow p_1:\widetilde{d}} \xrightarrow{p \leftarrow p_1:d} \xRightarrow{\widetilde{p'} \rightarrow p_1:\widetilde{d'}} N'$. That is, the network component N' records its delivery of some number of messages to us (from $\widetilde{p}$), the delivery of our message d to p_1, followed by its sending us possibly yet more messages (from $\widetilde{p'}$).

The semantics of `#(receive)` is similar, with the addition that the FFI call diverges if there is no reduction sequence causing a message to be enqueued. A key sanity check and technical lemma to show that this use of Hilbert choice is innocuous is the following:

▶ **Lemma 14** (FFI irrelevance). *Two CakeML steps starting from equal environments, equal expressions and bisimilar initial states yield bisimilar states and otherwise equal results.*

Let N_p denote the endpoint named p in N, and $N - p$ the network with that endpoint removed. Let $\mathsf{FFI}(cs)$ denote the FFI component of the CakeML state cs. Write $cs_1 =_{ffi} cs_2$ when $\mathsf{FFI}(cs_1)$ is bisimilar to $\mathsf{FFI}(cs_2)$ and all other components of the two states are equal.

▶ **Theorem 15** (Network Forward Correctness). *Let N be a well-formed PAYLOAD network that includes an arbitrary endpoint p. Further, assume a CakeML state cs that is appropriately related to N (see below), with $\mathsf{FFI}(cs) = (p, q, N - p)$. Then, if N can reduce to N', there exist cs', mp (the "merge program"), cs_1 and cs_2 (two "merge states") such that*

- $\mathsf{FFI}(cs') = (p, q', N' - p)$ *and cs' is appropriately related to N' ;*
- $(\llbracket N_p \rrbracket_{\mathsf{ML}}, cs) \rightarrow_c^* (mp, cs_1)$*;*
- $(\llbracket N'_p \rrbracket_{\mathsf{ML}}, cs') \rightarrow_c^* (mp, cs_2)$*; and*
- $cs_1 =_{ffi} cs_2$*.*

The "appropriate relation" above between a network and a CakeML state requires that: all bindings of N_p are present in the CakeML state's environment; our library functions (e.g., `List.drop`) are defined and have the expected behaviour; and for every function f used in a let expression in N_p, a CakeML function that is a totally correct implementation of f is present in the environment.

As CakeML is deterministic, Theorem 15 gives us that (i) all infinite traces in Payload are necessarily simulated by an infinite trace in CakeML, and (ii) compilation of a terminating choreography produces CakeML endpoints that will all also terminate successfully.

Though Theorem 15 tells us that every step taken by an endpoint will result in corresponding movement at the CakeML level, we have not transferred deadlock freedom to this level if the original choreography has only infinite paths. This is because currently, our theorems are not strong enough to rule out the possibility of *livelocks*: states where global progress is possible, but some nodes may be stuck waiting to receive. This is impossible by construction in Kalas, so while no such livelocks can occur (under weak fairness), operational correspondence by itself is only strong enough to guarantee global progress. One possible solution is to prove, in addition to operational correspondence, that an invariant stating "every receive can eventually be matched by a send" holds throughout the compilation chain.

8.2 Dynamic compiler by example

The dynamic compiler creates the initial environment assumed in Theorem 15, and proves that it is appropriate. The environment is built on top of the CakeML basis library by invoking CakeML's proof-producing code synthesis tool [21] on each function used in the endpoints' let expressions.

Kalas and the compiler are all deeply embedded in Hol4. Hence, users program choreographies by writing instances of the Hol4 datatype that encodes the choreography syntax. We define the system in Example 1 as a choreography `filter` where the producer has an infinite message stream, and where `test` is a simple function that checks if the message starts with "`A`" or not. To run the compiler, the invocation is

```
project_to_camkes builddir filename "filter";
```

This automatically performs the following tasks: (i) proves that the current environment is appropriate; (ii) evaluates the compiler in the logic to produce CakeML code for each of the three endpoints; (iii) produces end-to-end theorems for each endpoint by composing Theorems 12 and 15, discharging all assumptions; (iv) finally, generates all the glue code and build instructions necessary to create a complete system image that runs our choreography on top of the verified microkernel seL4 [17]. The system consists of three components in parallel, each running our generated CakeML code. The CakeML code is linked with a thin layer of C glue code that implements `send` and `receive` using the dataport and IPC mechanisms of the CAmkES [19] component platform. Thus: the user writes a choreography, calls `project_to_camkes`, and obtains a correctly compiled choreography running on a verified component platform on a verified microkernel.

9 Related Work

Session types [14] have seen extensive use in the π-calculus [15] and other concurrent languages [22, 29, 8, 16]. In recent years, the field has seen more mechanised proofs, perhaps motivated by past mistakes [31, 23]. In Castro et al. [3] a revised version of the session-typed π-calculus [31] is formalised in Coq [4]. Furthermore, Thiemann [27] proves type soundness and session fidelity in Agda [1] for an asynchronous functional session type language based on Gay et al. [10]. Tassarotti et al. [26] develop a higher-order concurrent logic, and verify a refinement procedure for a session-typed language as a case study.

Hallal et al. [12] synthesise the distributed components of a communicating system from a global choreography. Their result aims only to capture the communication logic of the system; by way of contrast, we consider local computation also.

Carbone and Montesi [2, 20] present a choreographic language with multi-party asynchronous session types (demonstrating the combination of the two approaches to great effect) along with a projection function into a variant of the calculus for multi-party sessions, with a proof – albeit pen-and-paper – of projection correctness. Kalas began as a simplified version of their language.

The most closely related work is two recent Coq formalisations of endpoint projection in different settings, by Cruz-Filipe *et al.* [6], and by Hirsch and Garg [13]. Cruz-Filipe et al. verify endpoint projection from CC (Core Choreographies) to a distributed process calculus. Hirsch and Garg [13] formalise endpoint projection from Pirouette, a higher-order functional choreographic language, where functions can return, and be parameterised on, choreographies. The most obvious difference between our work and these other papers is one of scope: both of [6, 13] formalise endpoint projection in isolation; for us, this is just the first step towards our goal of integrating endpoint projection into an end-to-end verified compilation toolchain that can be used to build real, runnable code.

Both CC and Pirouette are parameterised on a local language for describing computation, which is assumed to be available also in the target language. We achieve similar generality by representing local computation as shallow embeddings (functions in HOL4's logic). This lets us use a more abstract presentation, with no need to carry around an extra syntax, semantics, and associated well-formedness assumptions. The tradeoff is that we need a proof-producing (as opposed to verified) compiler phase to generate CakeML code.

In terms of semantics, one difference is that Kalas has asynchronous communication, whereas both CC and Pirouette are synchronous languages. Another interesting difference between the three languages is their representation of choreographies with infinite behaviour. Pirouette uses function closures. CC does not support the definition of local procedures, but executes in a context where a number of top-level, parameterless procedures are available. Kalas uses a fixpoint operator, which is parameterless, like CC, but supports arbitrary nesting of local procedures, like Pirouette.

CakeML has functions with closure semantics, but we nonetheless chose fixpoints over functions for Kalas. This is because, in an environment semantics, it is difficult to maintain a consistent view of the global environment in the presence of out-of-order execution: the semantics needs to track which local computations should be executed in the caller's environment (if they're ahead) or in the callee's environment (if they're behind). One solution is Cruz-Filipe et al. [6]'s approach, which breaks the abstraction of global, atomic actions by introducing an operator representing partially-completed procedure entry into the source language. Hirsch and Garg use an interesting approach, where function calls are considered global *both in source and target language*. In particular, executing a function call in a single endpoint has a CSP-like synchronous semantics where, as a single atomic action, the entire network performs the same function call together. While assuredly simplifying endpoint projection, this comes at the expense of complicated synchronisation when realising this in a distributed setting. In contrast, Kalas's unfolding of fixpoints can be implemented locally.

The target language used by Hirsch and Garg is a parallel composition of nodes expressed in the so-called *control language*. It mixes λ-calculus features with communication-enabling effects like send, receive and choose. This is rather like a functional language, which invites comparisons to our final target language, CakeML; but the role it plays in their development is much more akin to the role ENDPOINT plays in ours. Much like the relationship between Kalas and ENDPOINT, the feature set of Pirouette and the control language are essentially the same, except the latter is a localised representation.

Not all aspects considered by Hirsch and Garg, and by Cruz-Filipe et al., are present in our work. For example, Hirsch and Garg prove progress and preservation for an associated type system, while we do not consider types at all. In a companion paper, Cruz-Filipe *et al.* [7] prove that CC is Turing-complete, by showing that it can implement partial recursive functions. Turing completeness for Kalas is trivial because local computations may use arbitrary HOL functions.

10 Conclusion

We have presented what we believe to be the first end-to-end verified compiler for a choreographic language. After passing through five phases and two intermediate languages, our language, Kalas can be compiled to machine-code by reusing existing work from the CakeML project. Further, we have implemented a deployment on top of the micro-kernel seL4, itself also verified software. There, message-passing is implemented by IPC between separate user-processes.

There are a number of interesting directions for future work. Data types other than strings require a framework for verified marshalling and de-marshalling. Our model of the communication backend assumes unboundedly long message queues, which is arguably unrealistic. It would be interesting to investigate if deadlock freedom holds in a model where queues are bounded but not lossy. Alternative ITree-base [30] semantics (i.e., a co-inductive observational semantics) for Kalas and other intermediate languages, could significantly simplify projection proofs and allow for more lax projectability criteria. The CakeML compiler correctness theorem has an "unless the compiler output runs out of memory" side-condition, so liveness properties such as deadlock freedom carry over to the machine code only with this caveat, which could be discharged using CakeML's verified space-cost semantics [11]. We would also like to deploy on other communication backends, perhaps on top of TCP/IP, which would demonstrate verified distributed computation over the Internet.

References

1 Andreas Abel, Stephan Adelsberger, and Anton Setzer. Interactive programming in Agda – Objects and graphical user interfaces. *J. Funct. Program.*, 27:e8, 2017. doi:10.1017/S0956796816000319.

2 Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: Multiparty asynchronous global programming. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome*, pages 263–274. ACM, 2013. doi:10.1145/2429069.2429101.

3 David Castro, Francisco Ferreira, and Nobuko Yoshida. EMTST: engineering the meta-theory of session types. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems – 26th International Conference, TACAS 2020, Dublin, Ireland. Proceedings, Part II*, volume 12079 of *Lecture Notes in Computer Science*, pages 278–285. Springer, 2020. URL: http://doi.org/10.1007/978-3-030-45237-7_17, doi:10.1007/978-3-030-45237-7_17.

4 Coq Development Team. The Coq proof assistant, version 8.11.0, January 2020. doi:10.5281/zenodo.3744225.

5 Luís Cruz-Filipe and Fabrizio Montesi. A core model for choreographic programming. *Theor. Comput. Sci.*, 802:38–66, 2020. doi:10.1016/j.tcs.2019.07.005.

6 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Certifying choreography compilation. In Antonio Cerone and Peter Csaba Ölveczky, editors, *Theoretical Aspects of Computing – ICTAC 2021 – 18th International Colloquium, Virtual Event, Nur-Sultan, Kazakhstan, September 8-10, 2021, Proceedings*, volume 12819 of *Lecture Notes in Computer Science*, pages 115–133. Springer, 2021. doi:10.1007/978-3-030-85315-0_8.

7 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Formalising a Turing-complete choreographic language in Coq. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021. Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 15:1–15:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. URL: http://doi.org/10.4230/LIPIcs.ITP.2021.15, doi:10.4230/LIPIcs.ITP.2021.15.

8 Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session types for object-oriented languages. In Dave Thomas, editor, *ECOOP 2006 – Object-Oriented Programming, 20th European Conference, Nantes, France. Proceedings*, volume 4067 of *Lecture Notes in Computer Science*, pages 328–352. Springer, 2006. doi:10.1007/11785477_20.

9 Matthew Fernandez, June Andronick, Gerwin Klein, and Ihor Kuz. Automated verification of RPC stub code. In Nikolaj Bjørner and Frank S. de Boer, editors, *FM 2015: Formal Methods – 20th International Symposium, Oslo, Norway. Proceedings*, volume 9109 of *Lecture Notes in Computer Science*, pages 273–290. Springer, 2015. doi:10.1007/978-3-319-19249-9_18.

10 Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010. doi:10.1017/S0956796809990268.

11 Alejandro Gómez-Londoño, Johannes Åman Pohjola, Hira Taqdees Syeda, Magnus O. Myreen, and Yong Kiam Tan. Do you have space for dessert? a verified space cost semantics for CakeML programs. *Proc. ACM Program. Lang.*, 4(OOPSLA):204:1–204:29, 2020. doi:10.1145/3428272.

12 Rayan Hallal, Mohamad Jaber, and Rasha Abdallah. From global choreography to efficient distributed implementation. In *2018 International Conference on High Performance Computing & Simulation, HPCS 2018, Orleans, France, July 16-20, 2018*, pages 756–763. IEEE, 2018. URL: http://doi.org/10.1109/HPCS.2018.00122, doi:10.1109/HPCS.2018.00122.

13 Andrew K. Hirsch and Deepak Garg. Pirouette: Higher-order typed functional choreographies. *Proc. ACM Program. Lang.*, 6(POPL), January 2022. doi:10.1145/3498684.

14 Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.

15 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems – ESOP'98, 7th European Symposium on Programming, Lisbon. Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.

16 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016. doi:10.1145/2827695.

17 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):107–115, 2010. doi:10.1145/1743546.1743574.

18 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*, pages 179–192. ACM, 2014. doi:10.1145/2535838.2535841.

19 Ihor Kuz, Yan Liu, Ian Gorton, and Gernot Heiser. CAmkES: a component model for secure microkernel-based embedded systems. *J. Syst. Softw.*, 80(5):687–699, 2007. doi:10.1016/j.jss.2006.08.039.

20 Fabrizio Montesi. *Choreographic Programming.* Ph.D. thesis, IT University of Copenhagen, 2013. URL: http://www.fabriziomontesi.com/files/choreographic_programming.pdf.

21 Magnus O. Myreen and Scott Owens. Proof-producing synthesis of ML from higher-order logic. In Peter Thiemann and Robby Bruce Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 115–126. ACM, 2012. doi:10.1145/2364527.2364545.

22 Matthias Neubauer and Peter Thiemann. An implementation of session types. In Bharat Jayaraman, editor, *Practical Aspects of Declarative Languages, 6th International Symposium, PADL 2004, Dallas, TX, USA, June 18-19, 2004, Proceedings*, volume 3057 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2004. doi:10.1007/978-3-540-24836-1_5.

23 Randy Pollack. Closure under alpha-conversion. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers*, volume 806 of *Lecture Notes in Computer Science*, pages 313–332. Springer, 1993. doi:10.1007/3-540-58085-9_82.

24 Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montréal. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008. doi:10.1007/978-3-540-71067-7_6.

25 Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *J. Funct. Program.*, 29:e2, 2019. doi:10.1017/S0956796818000229.

26 Joseph Tassarotti, Ralf Jung, and Robert Harper. A higher-order logic for concurrent termination-preserving refinement. In Hongseok Yang, editor, *Programming Languages and Systems – 26th European Symposium on Programming, ESOP 2017, Uppsala, Sweden. Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 909–936. Springer, 2017. URL: http://doi.org/10.1007/978-3-662-54434-1_34, doi:10.1007/978-3-662-54434-1_34.

27 Peter Thiemann. Intrinsically-typed mechanized semantics for session types. In Ekaterina Komendantskaya, editor, *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal*, pages 19:1–19:15. ACM, 2019. URL: http://doi.org/10.1145/3354166.3354184, doi:10.1145/3354166.3354184.

28 Rob J. van Glabbeek. On the validity of encodings of the synchronous in the asynchronous π-calculus. *Inf. Process. Lett.*, 137:17–25, 2018. doi:10.1016/j.ipl.2018.04.015.

29 Vasco Thudichum Vasconcelos, Simon J. Gay, and António Ravara. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006. doi:10.1016/j.tcs.2006.06.028.

30 Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. *Proc. ACM Program. Lang.*, 4(POPL):51:1–51:32, 2020. doi:10.1145/3371119.

31 Nobuko Yoshida and Vasco Thudichum Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electron. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007. doi:10.1016/j.entcs.2007.02.056.

Deeper Shallow Embeddings

Jacob Prinz
University of Maryland, College Park, MD, USA

G. A. Kavvos
University of Bristol, UK

Leonidas Lampropoulos
University of Maryland, College Park, MD, USA

Abstract

Deep and shallow embeddings are two popular techniques for embedding a language in a host language with complementary strengths and weaknesses. In a deep embedding, embedded constructs are defined as data in the host: this allows for syntax manipulation and facilitates metatheoretic reasoning, but is challenging to implement – especially in the case of dependently typed embedded languages. In a shallow embedding, by contrast, constructs are encoded using features of the host: this makes them quite straightforward to implement, but limits their use in practice.

In this paper, we attempt to bridge the gap between the two, by presenting a general technique for extending a shallow embedding of a type theory with a deep embedding of its typing derivations. Such embeddings are almost as straightforward to implement as shallow ones, but come with capabilities traditionally associated with deep ones. We demonstrate these increased capabilities in a number of case studies; including a DSL that only holds affine terms, and a dependently typed core language with computational beta reduction that leverages function extensionality.

2012 ACM Subject Classification Software and its engineering; Software and its engineering → General programming languages; Social and professional topics → History of programming languages

Keywords and phrases type theory, shallow embedding, deep embedding, Agda

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.28

Funding This material is based upon work supported by NSF award #2107206, *Efficient and Trustworthy Proof Engineering* (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 NSF).

1 Introduction

Hosting a programming language inside another one is one of our favorite pastimes as programming language researchers. Such *embeddings* have proven useful in a variety of domains, ranging from particular applications (such as the design of hardware [9], and the writing of random property-based generators [10]), all the way to the mechanization of the metatheory of such applications (such as Kami [8] or Luck [19] respectively).

The complexity of such embeddings varies significantly depending on the features of both the *object language* and the *host language* involved. For example, embedding a simply-typed object language in a functional host language is a relatively straightforward task, yet one that has proved quite useful in practice, leading to a proliferation of libraries in mainstream ecosystems [26, 25, 17, 22]. On the other hand, embedding a dependently-typed language in another is a highly nontrivial task that gives rise to foundational challenges and that has received significant attention in recent years [1, 18].

Starting with Boulton et al. [5], language embeddings have been broadly classified as either *deep* or *shallow*. An embedding is deep when the terms of the object language are represented as inductive data in the host language. In a deep embedding terms may be

```
data Type : Set where
  _⇒_ : Type → Type → Type
  base : Type

data Ctx : Set where
  ∅ : Ctx
  _,_ : Ctx → Type → Ctx

data Var : (Γ : Ctx) → Type → Set where
  same : ∀{Γ T} → Var (Γ , T) T
  next : ∀{Γ T A} → Var Γ A → Var (Γ , T) A

data Exp : Ctx → Type → Set where
  var : ∀{Γ T} → Var Γ T → Exp Γ T
  lambda : ∀{Γ A B} → Exp (Γ , A) B
    → Exp Γ (A ⇒ B)
  app : ∀{Γ A B} → Exp Γ (A ⇒ B)
    → Exp Γ A → Exp Γ B
  tt : ∀{Γ} → Exp Γ base
```

Figure 1 A deep embedding of STLC in Agda.

arbitrarily manipulated and inspected via the usual mechanism of pattern matching. Deep embeddings for simply-typed languages are reasonably straightforward to construct, e.g. using an intrinsically-typed representation of terms as an inductive family [3, §3].

By contrast, shallow embeddings directly expand the constructs of the object language in terms of constructs of the host language. In the language of semantics, if a deep embedding can be thought of as defining the *initial* syntactic model of the object language, then a shallow embedding can be thought of as an *arbitrary* semantic model of the object language, but expressed and manipulated in the host language instead of mathematics [5, §5].

Deep Embeddings

For concreteness, consider the standard (intrinsic) deep embedding for the simply-typed lambda calculus (STLC) in Figure 1. We focus on the fragment consisting of a single base type, and functions. Contexts are lists of types. Variables are positions in that list, viz. de Bruijn indices. Terms are parameterized by a type and context. Every constructor of Term represents an STLC typing rule. For example, the app constructor takes a term of type $A \Rightarrow B$ and a term of type A, and produces produce a term of type B – parametrically in any context Γ. Functions over the syntax of STLC terms can then be defined using pattern-matching/induction, allowing a plethora of operations (e.g. substitution, reductions, optimizations) and metatheoretic proofs (e.g. admissibility of substitution).

In practice, Agda's type inference system plays very nicely with intrinsically-typed DSLs. Because the type and context are parameters of Term, Agda can infer them in the same way that it would for Agda programs. For example, when given the definition

```
lambda (var same) : Term (base → base)
```

Agda is able to infer that it is well-typed without additional information. It is thus not necessary for the user to specify the type of any part of this expression (e.g. the variable), thereby greatly increasing the usability of the DSL.

However, the same is not true for dependently-typed object languages: because of the presence of the *type conversion* rule

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A = B \text{ type}}{\Gamma \vdash M : B}$$

```
Ctx = Set
Type : Ctx → Set
Type Γ = Γ → Set

∅ : Ctx
∅ = ⊤
cons : (Γ : Ctx) → Type Γ → Ctx
cons Γ T = Σ Γ T

Var : (Γ : Ctx) → Type Γ → Set
Var Γ T = (γ : Γ) → T γ
same : ∀{Γ T} → Var (cons Γ T) (T ∘ proj₁)
same = λ (γ , t) → t
next : ∀{Γ T A} → Var Γ A
  → Var (cons Γ T) (A ∘ proj₁)
next x = λ (γ , t) → x γ

Term : (Γ : Ctx) → Type Γ → Set
Term Γ T = (γ : Γ) → T γ

Π : ∀{Γ}
  → (A : Type Γ) → Type (cons Γ A) → Type Γ
Π A B = λ γ → (a : A γ) → B (γ , a)

U : ∀{Γ} → Type Γ
U γ = Set

var : ∀{Γ T} → (icx : Var Γ T) → Term Γ T
var x = x

lambda : ∀{Γ A B} → Term (cons Γ A) B
  → Term Γ (Π A B)
lambda e = λ γ a → e (γ , a)

app : ∀{Γ A B} → Term Γ (Π A B)
  → (a : Term Γ A)→ Term Γ (λ γ → B (γ , a γ))
app e₁ e₂ = λ γ → (e₁ γ) (e₂ γ)
```

Figure 2 Shallow embedding of dependent type theory.

there is considerable overhead in defining a deep embedding of the object language, as we often have to somehow transport terms across type equalities. In practice this overhead is prohibitive, regardless of whether we are using the dependently-typed object language or proving things about it.

A number of researchers have attempted such deep embeddings of dependently-typed languages with varying degrees of success and completeness; see for example [12, 7, 1]. Each of these attempts represents a complex feat of proof engineering, often using various techniques (such as setoids), or assuming certain advanced features in the host language (such as quotient inductive types). In contrast to the simply-typed case, a simple, practical deep embedding of dependent type theory appears impossible with current technology. This invites a search for an easier alternative.

Shallow Embeddings

In contrast, shallow embeddings do not represent terms of the object language as inductive data, but rather directly interpret them as values in the host language.

For example, consider the shallow embedding of dependent type theory shown in Figure 2. Like with the deep embedding, we have definitions of types, contexts, variables, and terms. However, unlike the deep embedding, these are not datatypes. Instead, Ctx is defined as Agda's universe (called Set), and Type is the type of families of types over a given context. Finally, given a context Γ and type T, terms and variables are then defined as dependent functions $(\gamma : \Gamma) \to T\,\gamma$ over the family T. Moreover, for each term constructor that we had in the deep embedding, we have a corresponding definition in the shallow embedding of the corresponding type. For example, the lambda definition is defined using an Agda λ expression, and the app definition is defined using standard Agda function application.

Thus, each artifact of the object language is interpreted by its counterpart in the host language. Owing to its similarity to the set-theoretic model [16, §3] this is sometimes referred to as the *standard model* [1, §4], or even the *metacircular interpretation* of type theory [18].

This shallow embedding of dependent type theory obviates many of the difficulties one encounters when trying to define a deep embedding. The reason is that types themselves are no longer mere pieces of syntax, but mathematical objects that are subject to the definitional rules of Agda. Thus, the type equalities we had to transport over vanish [21]. As the shallow embedding inherits the definitional behavior of the host language, we find ourselves in a situation that enables quick and easy prototyping. Unfortunately, there is no free lunch: the fact that our terms are now semantic objects too means that we may no longer pattern-match on them.

A Middle Ground?

Given the complementary strengths and weaknesses of deep and shallow embeddings, it is natural to ask whether there is something in the middle: is there a form of embedding which is almost as easy to implement as a shallow embedding, but provides some of the extended, syntactic capabilities of a deep embedding?

In this paper we propose an answer to this question, which we call *deeper shallow embeddings*. To build a deeper shallow embedding we need a pre-existing shallow embedding of the object language. Then, the contexts and terms of the deeper shallow embedding are defined by "wrapping" the contexts and terms of the shallow embedding in an inductive data type. Following the technique of McBride [21], the types remain shallowly embedded, so that the host language takes care of type conversion for us.

We claim that deeper shallow embeddings preserve the mathematical simplicity of shallow embeddings, yet add extra capabilities which are useful for writing DSLs. This claim is substantiated by demonstrating these capabilities in practice. For example, we use them to (1) restrict the terms allowed in the DSL, (2) add metadata to terms (and perform computations dependent on this metadata), and (3) do a limited form of pattern-matching (which becomes more powerful in the presence of function extensionality).

Concretely, we make the following contributions:

- We present a way of deepening any shallow embedding, preserving its mathematical simplicity while also gaining additional capabilities that one might expect would require a deep embedding (Section 2). We show it by example on three shallow embeddings: the standard model (Section 2), a standard model for affine terms (Section 3), and a shallow embedding built from an inductive-recursive universe construction (Section 5).
- We demonstrate the usefulness of deeper shallow embeddings through a series of case studies showcasing different features that they exhibit over standard ones: adding metadata, restricting terms, and performing induction over terms (Section 3).
- We consider syntactic transformations such as substitution (Section 4), and show how to further increase the expressive power of a deeper shallow embedding by assuming function extensionality. This gives us the power to define – and compute – β-reduction (Section 5).

All of these results are formalized in Agda, available at `https://github.com/jeprinz/Deeper-Shallow-Embeddings`). Finally, we discuss related work in Section 6 and conclude by discussing limitations and future directions in Section 7.

```
data Context : S.Ctx → Set where
  ∅ : Context S.∅
  _,_ : Context sΓ → (T : S.Type sΓ) → Context (S.cons sΓ T)

data Var : {sΓ : S.Ctx} → (Γ : Context sΓ) → (T : S.Type sΓ)→ (S.Term sΓ T) → Set where
  same : Var (Γ , T) (T ∘ proj₁) S.same
  next : Var {sΓ} Γ A s → Var (Γ , T) (T ∘ proj₁) (S.next s)

data Term : {sΓ : S.Ctx} → (Γ : Context sΓ) → (T : S.Type sΓ)→ S.Term sΓ T → Set where
  lambda : Term (Γ , A) B s → Term Γ (S.Π A B) (S.lambda s)
  var : Var Γ T s → Term Γ T s
  app : Term Γ (S.Π A B) s₁ → (x : Term Γ A s₂)→ Term Γ (λ γ → B (γ , s₂ γ)) (S.app s₁ s₂)
  Π : (A : Term Γ S.U s₁) → (B : Term (Γ , s₁) S.U s₂)→ Term Γ S.U (S.Π₀ s₁ s₂)
  U : Term Γ S.U S.U
```

Figure 3 Deepening the shallow embedding of dependent type theory. Each constructor is a wrapper around a shallow constructor (prefixed by S).

2 Deeper Shallow Embeddings

As discussed before, a shallow embedding consists of a set of definitions in the host language. The valid object language programs in this embedding are exactly the well-typed terms built by combining these definitions. Thus, given a term t in the host language (here, Agda), the statement "t is a term of the shallow embedding" cannot be easily expressed in the host language – even if t is of a type that is evidently in the image of the shallow embedding. In fact, the ability to do this amounts to solving the *definability* problem in semantics.

By contrast, deeper shallow embeddings *internalize* this statement. By defining an inductive family of contexts and terms, we effectively *tag* the definable elements of the shallow embedding in a way that records their construction. This leaves us with something in between a deep and shallow embedding. On the one hand, our contexts, types, and terms carry elements of a shallow embedding. On the other hand, our typing derivations are deeply embedded in a datatype, so we may pattern-match on them.

To make the idea more concrete, let us revisit the shallow embedding of Figure 2. We will henceforth refer to this shallow embedding as S. By definition, to have a term of this shallow embedding means to have a shallow context Γ : Ctx, a shallow type T : Type Γ, and a dependent function (γ : Γ) → T γ. To deepen this embedding we must define a new datatype Term that is indexed by these three values. The constructors of the datatype encode the typing rules of the theory which it represents. The full definition may be found in Figure 3. All references to definitions in the shallow embedding start with the prefix S.

To understand this deepened term type better, we may for instance consider its lambda constructor. The λ constructor of the shallow embedding is referred to as S.lambda. If s is a term of the shallow embedding of the appropriate type and context, then S.lambda s represents the λ-abstraction of that term in the shallow embedding. Then, the lambda constructor of the deepened embedding inputs a term of a type parameterized by the shallow term s, and outputs an expression of type parameterized by the term S.lambda s. For clarity we write each deep constructor so that it refers to the corresponding shallow definition, but note that the code repetition could be eliminated by unfolding the definitions of the shallow embedding directly into the constructors of the datatype.

We can of course extend this idea to the construction of contexts and variables as well. The deepened context datatype Context : S.Ctx → Set is a family over the contexts S.Ctx of the shallow embedding. When we have an inhabitant of Context sΓ we know that (1) sΓ is a well-formed context in the shallow embedding, and (2) that sΓ is definable by starting from the empty context, and extending it by shallow types. Similarly, Var sΓ sT s is inhabited when s is a well-formed variable in the shallow embedding.

For a quick example of the deepened embedding, here is a definition of the identity function on the universe:

```
idU : Term ∅ (S.Π S.U S.U) _
idU = lambda (var same)
```

Notice that Agda is able to infer the corresponding shallow term, which has been elided and replaced by "_". We are always able to extract the shallow term from a deepened embedding:

```
extract : ∀{sΓ Γ T t} → Term {sΓ} Γ T t → S.Term sΓ T
extract {sΓ} {Γ} {T} {t} e = t
```

Hence, anything that can be proven about shallow embeddings can also be proven about deeper ones. For example, we can prove consistency (relative to the ambient type theory); that is, we can show that if we have an inhabitant of the shallow empty type, then we have an inhabitant of the empty type in the host language:

```
consistency : ∀{t} → Term {S.∅} ∅ (λ _ → ⊥) t → ⊥
consistency e = (extract e) tt
```

Syntax and Universe Level Simplifications

To facilitate readability, we have omitted certian details present in the formalization when presenting Agda code. In particular, we mostly leave out universally quantified parameters when they can easily be inferred. More importantly, we have also omitted all traces of universe levels. For example, in the formalization, the U constructor of Term looks like:

```
U₀ : ∀{sΓ Γ} → Term {sΓ} Γ S.U₁ S.U₀
```

Each universe level included in the deeper embedding needs it's own constructor, so for example U_1 is a separate constructor. Additionally, Term (and its shallow counterpart) needs some constructors to deal with universe level cumulativity. The full definition has three additional constructors: Lift, which raises a type to the next level; lift, which raises a term to a lifted type; and lower, which lowers a term from a lifted type. The corresponding shallow embedding definitions are implemented with the Agda terms of the same name.

3 Advantages of Deeper Shallow Embeddings, by Example

In this section we show that deeper shallow embeddings have usability advantages vis-a-vis both shallow and deep embeddings. We do so by example: we present deeper shallow embeddings which enable features that are not supported otherwise. Our discussion is focused around three examples: metadata on terms, pattern matching, and term restriction.

3.1 Metadata: Named Variables

Most embeddings of dependently-typed languages in proof assistants such as Agda or Coq rely on de Bruijn indices for representing and accessing variables: see e.g. [21, 18]. Evidently, this state of affairs is undesirable from a usability perspective.

We show that a deeper shallow embedding can be used to define a DSL with named variables. This is achieved without changing the standard shallow embedding of dependent types. Instead, the deepened embedding carries *metadata* about terms like a deep embedding could, in particular the names of variables. This way a user of the DSL is able to write lambda "x" (var "x") for the identity function, instead of the usual expression lambda (var same) .

To achieve this, we add a string along with each type in the deepened context:

```
data Context : S.Ctx → Set j where
  ∅ : Context S.∅
  _,_::_ : Context sΓ → String → (T : S.Type sΓ) → Context (S.cons sΓ T)
```

This allows us to write a function:

```
findVar : (Γ : Context sΓ) → String → ∑ Var Γ T t
                                      T,t
```

which searches for a variable name in the context. If the variables exists, it returns the shallow type and term of the variable as the Var value corresponding to it.

Unlike de Bruijn indices, this does not preclude the user from attempting to write a nonsensical term like lambda "x" (var "y") in an empty context. To deal with that eventuality we introduce an error type and term

```
data Error : Set where
  error : Error
```

We can now implement functions which, given a variable name, search the context for the corresponding type and term. If the variable is not found, the Error type is used.

```
resultType : (Γ : Context sΓ) → String → S.Type sΓ
resultType Γ name with findVar Γ name
... | nothing = λ _ → Error
... | just ((T , t) , x) = T


resultTerm : (Γ : Context sΓ)
  → (name : String) → S.Term sΓ (resultType Γ name)
resultTerm Γ name with findVar Γ name
... | nothing = λ _ → error
... | just ((T , t) , x) = t
```

We can now implement Term. The var constructor takes a string as an argument, and uses the aforementioned functions to compute its own type:

```
var : (name : String) → Term Γ (resultType Γ name) (resultTerm Γ name)
```

Finally, we let the lambda constructor take a string argument as well:

```
lambda : (name : String) → Term (Γ , name :: A) B s → Term Γ (S.Π A B) (S.lambda s)
```

Putting all of this together, we can now write terms with named variables! For example, the identity function can be written as:

```
id : Term ∅ (λ _ → (X : Set) → X → X) _
id = lambda "X" (lambda "x" (var "x"))
```

So, what if we try to use an unbound variable? Suppose we enter the definition

```
id : Term ∅ (λ _ → (X : Set) → X → X) _
id = lambda "X" (lambda "x" (var "y"))
```

This definition will not typecheck in Agda! By writing this definition we have communicated to Agda our expected type; we have also fixed the context to be empty. In the process of type-checking, Agda introduces the two variables `"X"` and `"x"` in the context, and then looks for `"y"`. It fails to find it, so var constructs an element of type `Term Γ ( λ _ → Error ) ( λ _ → error )` which does not match the stated type, causing a type error.

It is important to note that this technique is possible exactly because (a) `λ _ → Error` is a perfectly acceptable type of the shallow embedding, yet (b) none of our Term constructors create elements of `Term Γ ( λ _ → Error ) ( λ _ → error )` .

3.2 Pattern Matching and Induction: Compilation

Another drawback of shallow embeddings is that they do not provide the ability to induct on their terms. Given an arbitrary element `t : S.Term nil T`, there is not much we can really do: T is an arbitrary family over the empty context, so essentially an arbitrary Agda type. Hence, there is very little we can do, either with the term t or the type T.

In contrast, terms in the deeper shallow embedding are given by an inductive data type. It is therefore possible to write functions that pattern-match on them. For example, the following function compiles a dependently-typed term to JavaScript.

```
compileToJs : Term Γ T s → String
compileToJs {Γ} (lambda e) =
   "function(x" ++ (show (len Γ)) ++ ")"++ "{" ++ compileToJs e ++ "}"
compileToJs {Γ} (var x) =
  "x" ++ (show ((len Γ) - (index x)))
compileToJs (app e₁ e₂) =
   "(" ++ (compileToJs e₁) ++ "" ++ (compileToJs e₂) ++ ")"
```

The compilation proceeds in a fairly obvious way: a lambda is compiled to an anonymous function; and variable names are determined by their position in the context, using len and index to compute the relevant indices. Because of the lack of an induction principle, writing this simple function over a shallow embedding is impossible.

3.3 Syntactic Restrictions: Affine Terms

A final advantage of deeper shallow embeddings is that they provide the ability to restrict the terms that appear in the DSL. This has two advantages: (1) on an engineering level, it limits the interface exposed to the DSL programmer to a safer fragment; and (2) on a design level, it enables us to “carve out” a subset of interest of a shallow embedding.

When embedding a DSL in a shallow manner, the intention is that the user will build terms by composing the given definitions. However, shallow embeddings cannot stop a user from writing any term of the base language with a fortuitous type. For example, if one wants an element of S.Term S.∅ (S.Π S.U S.U) then instead of writing S.lambda (S.var S.same) , one could simply write λ $y\ x \to x$, circumventing the intent of the DSL designer. This example is fairly innocuous, but it is not difficult to imagine that this might become problematic in more complicated languages.

Kaposi, Kovács, and Kraus [18] propose a solution to this that uses Agda records for data hiding. The idea is that we wrap the contraptions of the shallow embedding in unary record types whose fields are private, and import only the wrapped model. That way the user cannot "access" the underlying representation. However, because of Agda's η-rule for record types, the wrapped model retains the definitional properties of the shallow embedding.

While this approach works, it relies heavily on features specific to Agda. If we use deeper shallow embeddings instead, only constructors of Term can be used to obtain terms. Therefore, the terms of the DSL can be restricted without making use of any data-hiding mechanism.

As an example, we can define a DSL which holds only affine terms, i.e. a DSL whose terms can use a variable in the context *at most once*. First, we need to augment contexts with some usage annotation. We define a family VarData : ContextsΓ → Set over deepened contexts, which stores a boolean flag for each type in the context. The flag records whether that variable has been used. Next, we inductively define a ternary relation

```
Check : VarData Γ → VarData Γ → VarData Γ → Set j
```

between flagged contexts, which holds if the first two contexts do not use the same variable, while the third context uses all variables used by either of the first two. This is reminiscent of ternary "context split" relations often used with linear types, e.g. session types [24, §3].

```
data VarData : Context sΓ → Set where
  ∅ : VarData ∅
  _,_ : VarData Γ → Bool → VarData (Γ , T)


data Check : VarData Γ → VarData Γ → VarData Γ→ Set j where
∅ : Check ∅ ∅ ∅
consLeft : (T : S.Type sΓ) → Check Γ₁ Γ₂ Γ₃ → Check (Γ₁ , true) (Γ₂ , false) (Γ₃ , true)
consRight : (T : S.Type sΓ) → Check Γ₁ Γ₂ Γ₃ → Check (Γ₁ , false) (Γ₂ , true) (Γ₃ , true)
consNeither : (T : S.Type sΓ) → Check Γ₁ Γ₂ Γ₃ → Check (Γ₁ , false) (Γ₂ , false) (Γ₃ , false)
```

Next, we define AffineTerm, which is a deepened type of terms that only holds affine terms. It is defined in much the same way as the standard deepened term type, but it incorporates elements of Check and VarData as evidence that the embedded terms are affine. Specifically, this evidence is used in the app constructor to ensure that the two subterms do not both use the same variable.

```
data AffineTerm : (Γ : Context sΓ) → VarData Γ → (T : S.Type sΓ) → S.Term sΓ  T → Set j where
app : AffineTerm Γ Γ₁ (S.Π A B) s₁ → (x : AffineTerm Γ Γ₂ A s₂) → Check Γ₁ Γ₂ Γ₃
    → AffineTerm Γ Γ₃ (λ γ → B (γ , s₂ γ)) (S.app s₁ s₂)
Π : AffineTerm Γ Γ₁ S.U s₁ → AffineTerm (Γ , s₁) (Γ₂ , b) S.U s₂ → Check Γ₁ Γ₂ Γ₃
  → AffineTerm Γ Γ₃ S.U (S.Π s₁ s₂)
_ ...
```

Finally, we are at a point where one may clearly witness the power of deeper shallow embeddings and the ability to pattern-match on them. We are able to define a function:

```
checkAffine : Term Γ T t → Maybe (Σ (VarData Γ) (λ vd → AffineTerm Γ vd T t))
```

which checks whether a given Term is affine, and – if so – reconstructs it as an AffineTerm. To achieve that we will need some helper functions, whose definitions we omit. Amongst these the most important one is check, which inputs two elements of VarData, and – if they do not conflict with each other – returns their combination alongside an element of Check. The "affinization" function itself recursively calculates the variables used by each expression. Whenever an app case is encountered, it checks that no variable is used twice.

4 Renamings and Substitutions

Having showcased some advantages of deeper shallow embeddings with small examples, we now turn to the more complicated operations of *renaming* and *substitution* that are central to dependent type theory.

As shallow embeddings are essentially semantic interpretations, substitutions are definable operations. In our running example of the "standard" interpretation, the type of all substitutions is the type of all functions between contexts; the action of a substitution on types and terms is then determined by function application:

```
Sub : Ctx → Ctx → Set
Sub Γ₂ Γ₁ = Γ₂ → Γ₁

extend : Sub Γ₂ Γ₁ → Term Γ₁ T → Sub Γ₂ (cons Γ₁ T)
extend sub e γ₂ = sub γ₂ , e (sub γ₂)

subType : Sub Γ₂ Γ₁ → Type Γ₁ → Type Γ₂
subType sub T = λ γ₂ → T (sub γ₂)

lift : (sub : Sub Γ₂ Γ₁) → (T : Type Γ₁) → Sub (cons Γ₂ (subType sub T)) (cons Γ₁ T)
lift sub T (γ , t) = sub γ , t

subTerm : (sub : Sub Γ₂ Γ₁) → Term Γ₁ T → Term Γ₂ (subType sub T)
subTerm sub e = λ γ₂ → e (sub γ₂)
```

There is no evident non-inductive way to isolate the renamings amongst these.

In contrast, in deep embeddings substitutions are usually given in an algebraic style, and defined inductively from the empty substitution, the identity, composition, weakening, and extension [12, §3.5] [7, §2] [1, §3]. Renamings may also be defined by induction-recursion – at the same time as a recursive function that interprets them as full substitutions [2, §5].

How is one to bridge this gap for deeper shallow embeddings? As before, the answer lies exactly in the middle: we are able to define a data type Ren of renamings, which is indexed in shallowly-embedded substitutions. Like with shallow embeddings but unlike with deep embeddings, renamings here have inherent computational content: they are actual functions mapping variables to variables.

```
Ren : S.Sub sΓ₂ sΓ₁ → Context sΓ₂ → Context sΓ₁ → Set
Ren sub Γ₂ Γ₁ = Var Γ₁ T t → Var Γ₂ (S.subType sub T) (S.subTerm sub t)

lift : Ren sub Γ₂ Γ₁ → Ren (S.lift sub T) (Γ₂ , S.subType sub T) (Γ₁ , T)

renTerm : Ren sub Γ₂ Γ₁ → Term Γ₁ T t → Term Γ₂ (S.subType sub T) (S.subTerm sub t)
renTerm ren (lambda e) = lambda (renTerm (lift ren) e)
renTerm ren (var x) = var (ren x)
renTerm ren (app e₁ e₂) = app (renTerm ren e₁) (renTerm ren e₂)
renTerm ren (Π A B) = Π (renTerm ren A) (renTerm (lift ren) B)
renTerm ren U = U
```

In many ways, our definition resembles the traditional definition of renaming in simply-typed λ-calculus [14, §II.1.1], i.e. a map from variables to variables that respects types. However, making that definition dependent involves quite a bit of tricky indexing [11, §5]. Fortunately, we have no need for that: our deepened renamings are also "tracked" by a corresponding substitution of the shallow embedding, so we can use that instead.

Substitutions work similarly: a deepened substitution Sub is parameterized by a shallow substitution, i.e. an element of S.Sub. Interestingly, renaming is used to define substitution.

```
Sub : S.Sub sΓ₂ sΓ₁ → Context sΓ₂ → Context sΓ₁ → Set₁
Sub sub Γ₂ Γ₁ = Var Γ₁ T t → Term Γ₂ (S.subType sub T) (S.subTerm sub t)

liftSub : Sub sub Γ₂ Γ₁ → Sub (S.lift sub) (Γ₂ , S.subType sub T) (Γ₁ , T)
liftSub sub same = var same
liftSub sub (next x) = renTerm next (sub x)

extend : Sub sub Γ₂ Γ₁ → Term Γ₁ T t → Sub (S.extend sub t) Γ₂ (Γ₁ , T)
extend sub e same = subTerm sub e
extend sub e (next x) = sub x

subTerm : Sub sub Γ₂ Γ₁ → Term Γ₁ T t → Term Γ₂ (S.subType sub T) (S.subTerm sub t)
subTerm sub (lambda e) = lambda (subTerm (liftSub sub) e)
subTerm sub (var x) = sub x
subTerm sub (app e₁ e₂) = app (subTerm sub e₁) (subTerm sub e₂)
subTerm sub (Π A B) = Π (subTerm sub A) (subTerm (liftSub sub) B)
subTerm sub U = U
```

To sum up, with deeper shallow embeddings we can perform complicated *syntactic* operations on embedded terms, such as substitution and renaming. Naturally, one wonders: how far can we take this? For example, could we encode a β-reduction step? We answer that in the next section.

5 β-reduction and Injectivity of Products

With induction under our belts, it might seem that we have all of the components needed to define one-step β-reduction. After all, we can certainly inductively traverse a term looking for a λ-abstraction to the left of an application, and – if we find one – apply substitution as described in the previous section. Here is a first attempt:

```
βreduce : Term Γ T t → Term Γ T t
βreduce (lambda e) = lambda (βreduce e)
βreduce (var x) = var x
βreduce (Π A B) = Π (βreduce A) (βreduce B)
βreduce U = U
βreduce (app e₁ e₂) = ?
```

In fact, the completed clauses of this definition are not one-step β-reduction. It will perform more reductions than necessary, as e.g. we recurse on both the left and right subtree of Π. We could rework this definition into a correct one, but this simpler variant suffices to illustrate the point.

Given our syntax, the only computationally non-trivial case is that of app. To complete it we would ideally like to check whether the expression is a redex, i.e. whether $\mathtt{e}_1$ is of the appropriate form lambda e. If not we can mindlessly recurse like in all other cases; but if we find a λ-abstraction, we would like to perform the substitution. Unfortunately, we are not able to pattern-match on $\mathtt{e}_1$. The reason is that its type is Term Γ (S.Π A B) $\mathtt{s}_1$. As Term is an inductive family indexed in shallow types, induction is only allowed when we have a term of general type Term Γ T $\mathtt{s}_1$ with all of Gamma, T and $\mathtt{s}_1$ free.

Being more precise about the problem we are trying to solve, we would like a function

```
castLam : Term Γ (S.Π A B) t → Maybe (Term (Γ , A) B (λ (γ , a) → t γ a))
```

that checks if its argument is a λ-abstraction, and if so returns its body. We can make some progress towards defining such a function by generalizing the Π type to an arbitrary type T. We may then induct on it; if it happens to be a λ-abstraction, we return a term of type Term (Γ , A′) B′ t' where A′, B′, and t′ have no particular relation to the input.

In fact, we can do better than that. We can define a fairly simple function castLamImpl which checks whether its argument is a λ-abstraction. If it is, it returns

- types A′ and B′, the latter depending on the former,
- the body t′ of the λ-abstraction,
- a proof that T ≡ S.Π A′ B′, and
- a proof that t is equal to the shallow-λ-abstraction of t′.

All of this is encoded in an ugly nested Σ type, so here we abbreviate the syntax to convey the meaning:

```
castLamImpl : Term Γ T t → Maybe ∑_{A,B,t'} λγ → ( T γ , t γ) ≡ λγ →
  ( ( S.Π A B ) γ , λ a → t' (γ, a) ) × Term (Γ , A) B t'
castLamImpl (lambda e) = just (_ , _ , _ , refl , e)
castLamImpl _ = nothing
```

One might expect that it would be easy to define castLam using the proofs of equality provided by castLamImpl. But that is not so! If we apply castLamImpl to something of type Term Γ (S.Π A B) t we would obtain, amongst other things, types A′ and B′ along with a proof of S.Π A B ≡ S.Π A′ B′. Hence, at the very least, we would have to show that p : A ≡ A′ and B $\equiv_{\mathtt{p}}$ B′, where the $\equiv_{\mathtt{p}}$ means that the two types are equal "over" the equality p of the types on which they depend. In more detail, the exact statement we need is

```
Π-injective :
  (λ γ → ((S^T.Π A B) γ , λ a → t (γ , a))) ≡ (λ γ → ((S^T.Π A' B') γ , λ a → t' (γ , a)))
  → (A , B , t) ≡ (A' , B' , t')
```

If we had a proof of this, it would be possible to derive castLam from castLamImpl:

```
castLam : Term Γ (S.Π A B) t → Maybe (Term (Γ , A) B (λ (γ , a) → t γ a))
castLam e with castLamImpl e
... | nothing = nothing
... | just (A , B , t' , p , e') with (Π-injective p)
... | refl = just e'
```

Using that we could complete the final case of our β-reduction function:

```
βreduce (app e₁ e₂) with castLam e₁
... | nothing = app (βreduce e₁) (βreduce e₂)
... | just e = subTerm (extend idSub e₂) e
```

Unfortunately, Π-injective is a very long way from being true. This is a well-known issue in the metatheory of dependent types, known as the *injectivity of type constructors*. To begin, notice that given any Agda types A, B, C, D, the following statement is not true:

$$(A \to B) \equiv (C \to D) \quad \to \quad A \equiv C \times B \equiv D$$

Taking A and C to be empty types, B to be empty, and D to be the unit type of one element, we see that in homotopy type theory [23] the antecedent is true by univalence, yet the consequent proves that the empty and the unit types are equal. Thus, neither this, nor the corresponding more general statement about dependent function types in Agda, can be true. But since S.Π was defined in terms of Agda function types, neither will Π-injective!

Additionally, our shallow types are functions from a context to an Agda type. In order to prove equality of shallow types, we will therefore also require function extensionality.

5.1 Type codes: an Inductive-Recursive Universe

The problem we faced above boils down to two facts: (1) the types of the shallow embedding S were elements of the Agda universe of types Set, and (2) Set itself is far too *open*. This means that we are not in general able to induct on elements of Set (i.e. Agda types) so as to prove the requisite injectivity lemma. This restriction is only natural: as Agda supports new data type definitions, which would change the induction principle of Set.

The solution is to change the shallow embedding S, so that its types come from a *closed* universe. This can be achieved using a classic construction by Martin-Löf [20], which is sometimes known as the *inductive-recursive universe* [13, §1]. The idea is simple: instead of having a universe of types, we construct a universe of *type codes*, i.e. an inductive data type whose elements represent types. At the same time we define a family over this universe, which interprets these codes as types of the host language.

The technique itself is best illustrated by example. In fact, as our object language contains a universe itself, we will generate an infinite hierarchy of universes by simply adjoining an additional $\mathbb{N}$ parameter. This family TypeCode : $\mathbb{N} \to$ Set of inductive-recursive universes is defined as follows. Technically, as written here this definition is not strictly positive. However, the full definition in our formalization includes the standard trick of performing induction on the $\mathbb{N}$ parameter so that it is admissible in Agda.

```
data TypeCode : ℕ → Set where
  'U : TypeCode (suc n)
  'Π : (A : TypeCode n) → (⟦ A ⟧ → TypeCode n) → TypeCode n
  'lift : TypeCode n → TypeCode (suc n)


⟦_⟧ : TypeCode n → Set
⟦ 'U ⟧ = TypeCode n
⟦ 'Π A B ⟧ = (a : ⟦ A ⟧) → ⟦ B a ⟧
⟦ 'lift T ⟧ = ⟦ T ⟧
```

Notice the distinctive use of induction-recursion in the constructor 'Π, which takes a type code and a family over the *interpretation* of that type code.

We can then build a new shallow embedding whose types are, in fact, type codes. The construction amounts to sprinkling the decoding function wherever it should be to turn type codes into bona-fide types:

```
Ctx = Set
Type : ℕ → Ctx → Set
Type n Γ = Γ → TypeCode n
Term : ∀{n} → (Γ : Ctx) → Type n Γ → Set
Term Γ T = (γ : Γ) → ⟦ T γ ⟧
```

```
U : ∀{n} → Type (suc n) Γ
U = λ _ → 'U

Π : (A : Type (suc n) Γ)
  → Type (suc n) (cons Γ A) → Type (suc n) Γ
Π A B = λ γ → 'Π (A γ) ((λ a → B (γ , a)))
```

The implementation of Type is now a function from a context to TypeCode, while the implementation of Term uses the decoding function. Moreover, notice that types and terms are now indexed by ℕ, so they live at various levels of the inductive-recursive universe hierarchy.

Finally, we can deepen this new shallow embedding, not forgetting to thread the new universe levels around in the process. The new definition of Context, Var, and Term looks nearly identical to our original one from Figure 3, except that the prefix S now refers to our new shallow embedding with type codes.

5.2 Completing the Definition

The benefit of inductive-recursive universes is that, as they are inductively defined, their constructors are injective. As a consequence we are now able to prove the injectivity of 'Π:

```
Π-injective-typecode : (('Π A B) , t) ≡ (('Π A' B') , t') → (A , B , t) ≡ (A' , B' , t')
Π-injective-typecode refl = refl
```

This is almost a proof of Π-injective, with the difference that we must somehow extract proofs S.Π A B ≡ S.Π A' B' and t ≡ t' from its premise. Looking at the premise more carefully, it roughly gives us what we want, but as an equality of functions:

```
(λ γ → ((S.Π A B) γ , λ a → t (γ , a))) ≡ (λ γ → ((S.Π A' B') γ , λ a → t' (γ , a)))
```

By post-composing these two functions with the first projection, we can obtain an equality of the form

```
(λ γ → (S.Π A B) γ a) ≡ (λ γ → (S.Π A' B') γ)
```

To turn that into the desired equality, we must also use the *function extensionality* axiom:

```
funExt : ((x : A) → f x ≡ g x) → f ≡ g
```

which is not natively available in Agda. However, it is well-known to be consistent with intentional Martin-Löf type theory [23, §2].

To make the behavior of funExt somewhat more computational, we add the following *rewrite rule* to Agda:

```
postulate
  funExtElim : funExt (λ x → refl) ≡ refl


{-# REWRITE funExtElim #-}
```

This definitional equality is known to hold in e.g. the set-theoretic model. With this machinery in place, we are able to complete the definition of β−reduction.

6 Related Work

Our approach to deepening shallow embeddings is closely related to McBride's "Outrageous but Meaningful Coincidences" [21]. In that work, McBride introduces a deeper shallow embedding for dependent type theory by wrapping it in a datatype. The embedding differs from ours in that instead of indexing deepened terms by their shallow counterparts, McBride writes an evaluator that targets the shallow embedding using induction-recursion. While the aesthetics of each approach are debatable, it is a fact that the inductive-recursive interpreter makes it difficult to define syntactic operations like renaming and substitution. This is because any operations on the terms must include a proof that they commute with the evaluation function. Our purely inductive definition makes such syntactic transformations much simpler to define. Our work focuses on the engineering aspects of this technique, and its various practical uses and advantages over traditional shallow embeddings.

Kaposi et al. [18] propose that shallow embeddings are a "morally correct" alternative to deep embeddings. They consider how one can be sure that a shallow embedding corresponds to the desired type theory; how does one know that no extra equalities or terms have been introduced? They prove several such correctness results *externally* to Agda. By contrast, our deeper shallow embeddings can be seen as bringing some of this work back inside of Agda. Kaposi et. al. also present a technique for term restriction in shallow embeddings, which we describe and compare to our techniques in (Section 3.3)

Of course, one way to build a maximally-expressive embedding of type theory in type theory is to actually use a deep embedding. Perhaps the most technically accomplished work in that direction is by Altenkirch and Kaposi [1], who use *quotient inductive-inductive types* (QIITs) to present a deep embedding of type theory (with explicit substitutions) that is already quotiented under its own definitional equality. The power of this technique has been shown by proving normalization results [2].

Previous work by Danielsson [12] and Chapman [7] encodes type theory (with explicit substitutions) in type theory itself. Because it lacks the technology of QIITs, this approach has to explicitly transport the representations of terms along type equalities using the type conversion rule.

The functional programming community has also explored various forms of embedding. Gibbons and Wu [15] present a unified approach through polymorphism and folding. Carette et al. [6] design a system using Haskell typeclasses to build shallow embeddings which simultaneously allow users to add new terms at any time but also to define new interpreters.

Another instance of an "intermediate" embedding that is between deep and shallow can be found in the work of Augustsson [4], where the author hints at the possibility of so-called *neritic* embeddings.

7 Conclusion

In this paper we presented a technique for *deepening* a shallow embedding by storing it in a data type. This allows us to inspect, analyze, and manipulate terms in a way that is usually associated with deep embeddings, while retaining the automation afforded by the shallow embedding. We demonstrated the practical uses of this technique in a series of small case studies, and showed how – assuming function extensionality – we can recover syntactic transformations, such as β-reduction.

One application of domain-specific languages is metaprogramming. In Lisp-like languages, code can be manipulated as data, quoted, and unquoted. Building a typed DSL can be seen as implementing a typed version of the sort of metaprogramming that Lisp can do. For example, quoting can be seen as simply writing an element of the DSL, and unquoting as corresponding to our extract functions. One could even imagine that the syntax of the DSL could look like the syntax of the host language, and therefore the host language would interpret code as either host code or DSL code depending on the type. Our hope is that ultimately dependently-typed embedding techniques will yield a typed (and therefore less error-prone) version of the sort of metaprogramming that one can do in Lisp.

Moreover, we explored the idea that there is more than just deep embeddings (initial models) and shallow embeddings (arbitrary models) in the design space. Unexplored constructions of shallow models harbor additional power, which might be of significant interest for particular applications. In the future, we are hoping to further explore this spectrum of "hybrid" embeddings, with the ultimate goal of a practical approach for encoding a dependent type theory inside a dependent type theory.

Of course, the technique of deepening a shallow embedding has limitations. For example, deeper shallow embeddings allow induction over terms, but not over types. Various steps can be taken to improve that; for example, switching to an inductive-recursive universe allowed us to prove that Π is injective. However, the same trick does not allow us to prove e.g. that U and Π are unequal.[1]

Finally, it is often the case that we would like to compute syntactic transformations on the terms of an embedded language – like the β-reduction function given in §5 – as they might prove useful in compilation, optimizations, etc. Thus, the question of expressivity arises: which syntactic transformations can be expressed over deeper shallow embeddings? We believe that most transformations that can be performed on a shallow embedding can be lifted to its deepened version: as shallow embeddings often "quotient away" numerous differences between terms, any transformation on that level is likely easy to extend to deepened terms, but may require some effort. For example, when we defined renaming and substitution in Section 4, we had to define substitution on the shallow embedding first; when defining β-reduction in Section 5, we had to find a shallow embedding which satisfied certain equalities first, which required some ingenuity. In short, the transformations that can be expressed are not quite as expressive as those over deep embeddings, and may also require additional work.

Because of these restrictions, we view the main benefits of a deeper shallow embedding to be for metaprogramming, rather than for studying the metatheory of type theory.

[1] As types are families of type codes over a context Γ, by funExt they are equal when $\Gamma = \bot$.

References

1 Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 18–29. Association for Computing Machinery, 2016. doi:10.1145/2837614.2837638.

2 Thorsten Altenkirch and Ambrus Kaposi. Normalisation by Evaluation for Type Theory, in Type Theory. *Logical Methods in Computer Science*, 13(4), 2017. doi:10.23638/LMCS-13(4:1)2017.

3 Thorsten Altenkirch and Bernhard Reus. Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In Jörg Flum and Mario Rodriguez-Artalejo, editors, *Computer Science Logic*, volume 1683 of *Lecture Notes in Computer Science*, pages 453–468, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. doi:10.1007/3-540-48168-0_32.

4 Lennart Augustsson. Making edsls fly. http://vimeo.com/73223479, 2012.

5 Richard J. Boulton, Andrew D. Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In Victoria Stavridou, Thomas F. Melham, and Raymond T. Boute, editors, *Theorem Provers in Circuit Design, Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience, Nijmegen, The Netherlands, 22-24 June 1992, Proceedings*, volume A-10 of *IFIP Transactions*, pages 129–156. North-Holland, 1992. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/11/EmbeddingPaper.pdf.

6 Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009. doi:10.1017/S0956796809007205.

7 James Chapman. Type Theory Should Eat Itself. *Electronic Notes in Theoretical Computer Science*, 228:21–36, 2009. doi:10.1016/j.entcs.2008.12.114.

8 Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: A platform for high-level parametric hardware specification and its modular verification. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017. doi:10.1145/3110268.

9 Koen Claessen. *Embedded Languages for Describing and Verifying Hardware*. PhD thesis, Chalmers University of Technology, Gothenburg, Sweden, 2001. URL: http://publications.lib.chalmers.se/publication/636-embedded-languages-for-describing-and-verifying-hardware.

10 Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP*, 46, January 2000. doi:10.1145/1988042.1988046.

11 Thierry Coquand. Canonicity and normalization for dependent type theory. *Theoretical Computer Science*, 777:184–191, 2019. doi:10.1016/j.tcs.2019.01.015.

12 Nils Anders Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, volume 4502 of *Lecture Notes in Computer Science*, pages 93–109. Springer, 2006. doi:10.1007/978-3-540-74464-1_7.

13 Peter Dybjer and Anton Setzer. Indexed induction–recursion. *The Journal of Logic and Algebraic Programming*, 66(1):1–49, 2006. doi:10.1016/j.jlap.2005.07.001.

14 Marcelo Fiore. Semantic analysis of normalisation by evaluation for typed lambda calculus. In *Proceedings of the 4th ACM SIGPLAN international conference on Principles and Practice of Declarative Programming - PPDP '02*, pages 26–37. ACM Press, 2002. doi:10.1145/571157.571161.

15 Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 339–347. ACM, 2014. doi:10.1145/2628136.2628138.

16 Martin Hofmann. Syntax and Semantics of Dependent Types. In Andrew M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, pages 79–130. Cambridge University Press, 1997. doi:10.1017/CBO9780511526619.004.

17 Paul Hudak and Donya Quick. *The Haskell School of Music: From Signals to Symphonies*. Cambridge University Press, 2018. doi:10.1017/9781108241861.

18 Ambrus Kaposi, András Kovács, and Nicolai Kraus. Shallow Embedding of Type Theory is Morally Correct. In Graham Hutton, editor, *Mathematics of Program Construction*, volume 11825 of *Lecture Notes in Computer Science*, pages 329–365, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-33636-3_12.

19 Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. Beginner's Luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 114–129, 2017. URL: http://dl.acm.org/citation.cfm?id=3009868, doi:10.1145/3009837.3009868.

20 Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, 1984.

21 Conor McBride. Outrageous but meaningful coincidences: Dependent type-safe syntax and evaluation. In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming*, WGP '10, pages 1–12, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1863495.1863497.

22 Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530, June 2013. doi:10.1145/2499370.2462176.

23 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

24 Vasco T. Vasconcelos. Fundamentals of session types. *Information and Computation*, 217:52–70, 2012. doi:10.1016/j.ic.2012.05.002.

25 Brent Yorgey. Diagrams: A declarative dsl for creating vector graphics. https://diagrams.github.io/doc/manual.html, 2013.

26 Christina Zeller and Ivan Perez. Mobile game programming in haskell. In Donya Quick and Daniel Winograd-Cort, editors, *Proceedings of the 7th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design, FARM@ICFP 2019, Berlin, Germany, August 18-23, 2019*, pages 37–48. ACM, 2019. doi:10.1145/3331543.3342580.

Reflexive Tactics for Algebra, Revisited

Kazuhiko Sakaguchi
Unaffiliated researcher, Tokyo, Japan

Abstract

Computational reflection allows us to turn verified decision procedures into efficient automated reasoning tools in proof assistants. The typical applications of such methodology include decidable algebraic theories such as equational theories of commutative rings and lattices. However, such existing tools are known not to cooperate with *packed classes*, a methodology to define mathematical structures in dependent type theory, that allows for the sharing of vocabulary across the inheritance hierarchy. Moreover, such tools do not support homomorphisms whose domain and codomain types may differ. This paper demonstrates how to implement reflexive tactics that support packed classes and homomorphisms. As applications of our methodology, we adapt the `ring` and `field` tactics of Coq to the commutative ring and field structures of the Mathematical Components library, and apply the resulting tactics to the formal proof of the irrationality of $\zeta(3)$ by Chyzak, Mahboubi, and Sibut-Pinote. As a result, the lines of code in the proof scripts have been reduced by 8%, and the time required for proof checking has been decreased by 27%.

2012 ACM Subject Classification Computing methodologies → Theorem proving algorithms; Theory of computation → Automated reasoning; Theory of computation → Type theory; Theory of computation → Constraint and logic programming

Keywords and phrases Coq, Elpi, λProlog, Mathematical Components, algebraic structures, packed classes, canonical structures, proof by reflection

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.29

Related Version *Full Version*: `https://arxiv.org/abs/2202.04330`

Supplementary Material *Software*: `https://doi.org/10.5281/zenodo.6487597`

Acknowledgements The author would like to thank Enrico Tassi for his help with Coq-Elpi, particularly for adding some features and fixing performance bottlenecks, and Assia Mahboubi for providing useful examples for identifying issues in earlier versions of our `ring` and `field` tactics. The comments on earlier versions of this paper by Anne Baanen, Assia Mahboubi, Enrico Tassi, and the anonymous reviewers have also been a great help.

1 Introduction

Computational reflection [2] makes it possible to replace proof steps with computations and has been widely used to automate proofs in some proof assistants such as Coq [54] and Agda [12]. For example, we can prove an integer equation $(a - b) + (b - a) = 0$ as follows.

1. We obtain a term $e \coloneqq \mathrm{Add}(\mathrm{Sub}(\mathrm{X}_0, \mathrm{X}_1), \mathrm{Sub}(\mathrm{X}_1, \mathrm{X}_0))$ from the LHS of the equation, where $\mathrm{Add}, \mathrm{Sub} : \mathrm{E} \to \mathrm{E} \to \mathrm{E}$ and $\mathrm{X} : \mathbb{N} \to \mathrm{E}$ are constructors of an inductive type E describing the syntax. This step is called *reification* (also called metaification or quotation).
2. We normalize e to a formal sum $0X_0 + 0X_1$ and check that all its coefficients are zero. This decision procedure is implemented and performed inside the proof assistant, and its validity is justified by a correctness lemma.

This process (detailed in Section 2.3 and 2.4) applies to any equation over an Abelian group, and this proof scheme can be adapted to other mathematical structures, e.g., commutative rings [11, 31], fields [60], lattices [34], and Kleene algebras [14].

Unfortunately, existing implementations of this proof methodology are known not to cooperate with *packed classes* very well [27, 33]. The packed classes discipline [26] is a methodology to define mathematical structures in dependent type theory, which allows for the sharing of vocabulary (definitions and lemmas) across the inheritance hierarchy of structures as well as multiple inheritance (Section 2.1 and 2.2). This methodology is used in the MathComp library [39] for Coq extensively, to provide more than 70 mathematical structures such as finite groups, rings, fields, as well as their homomorphisms.

The source of the incompatibility between proof by large-scale reflection and packed classes is twofold. Firstly, packed classes require the proof tools (e.g., the `rewrite` tactic) to compare overloaded operators (e.g., the multiplication of rings) modulo conversion to enable the sharing of vocabulary. This conversion is another kind of computational reflection, so-called *small-scale reflection*. Secondly, in most of the existing tactics based on large-scale reflection, their reification procedures recognize operators purely syntactically and do not take conversion into account. We briefly review an existing approach to matching modulo conversion, called the *keyed matching* discipline [28], used in the SSReflect plugin [58] and the Lean theorem prover [41, 42], and propose a reification scheme based on keyed matching to address this shortcoming (Section 3).

Another issue is that extending the above reflection scheme to support homomorphisms, whose domain and codomain types may differ, requires a more involved data type describing the syntax, another decision procedure, and correctness proof. In this paper, instead of redefining the syntax and the decision procedure, we propose a solution based on two reflection steps. The first step, which we call *preprocessing*, pushes down homomorphisms in the input terms to leaves using the structure preservation laws, e.g., $f(x+y) = f(x) + f(y)$. Although preprocessing requires a heterogeneous syntax that can express a term that has subterms of different types, it remains quite simple since we do not have to replace variables with numbers in preprocessing as in X_n above. In the second step, we apply the reflexive decision procedure that uses a homogeneous syntax, as explained at the beginning of this section. While some generic goal preprocessing methodologies [7, 9, 36] have been implemented as standalone tactics, the characteristic of our approach is that those two steps are strongly tied and the preprocessor is not intended to be called solely. This approach allows us to avoid performing reification twice and thus has a performance advantage. Moreover, the preprocessing step allows us to adapt an existing reflexive tactic to operators not directly supported by its syntax (e.g., opposite which can be expressed as a combination of zero and subtraction) without modifying the existing syntax, procedures, and correctness proofs for the second step (Section 4).

As an application of our methodology, we adapt the `ring` and `field` tactics [31, 60] of Coq to the commutative ring and field structures of MathComp, with support for homomorphisms and some operators that cannot be directly described by the provided syntax (Section 5.1). Furthermore, we apply the resulting tactics to the formal proof of Apéry's theorem [16, 17, 37] (Section 5.3).

▶ **Theorem 1** (Apéry's theorem [3, 62]). *The following constant, the evaluation of the Riemann zeta function at 3, is irrational:*

$$\zeta(3) = \sum_{i=1}^{\infty} \frac{1}{i^3}.$$

For this purpose, we also reimplemented a technique [17, Section 4.3][37, Section 2.4] to automatically prove proof obligations generated by the `field` tactic using the `lia` (linear integer arithmetic) tactic [6, 59] of Coq (Section 5.2). This reimplementation is extensible by

declaring canonical structure instances and supports a broader range of problems, thanks to the approach of Gonthier et al. [29] to use canonical structures (Section 2.1) for proof automation. As a result, the lines of code in the formal proof of Apéry's theorem have been reduced by 8% and the time required for proof checking has been reduced by 27%.

Our reification procedures are written in Coq-Elpi [53] (Section 2.4). Elpi [24, 45] is a dialect of λProlog [40], a higher-order logic programming language. The Coq-Elpi plugin lets us write Coq commands and tactics in Elpi, and provides a higher-order abstract syntax (HOAS) [44] embedding of Coq terms in Elpi, to manipulate syntax trees with binders in a comfortable way.

2 Background

This preliminary section briefly reviews the main ingredients of this paper, namely, canonical structures (Section 2.1), the hierarchy of mathematical structures in MathComp (Section 2.2), large-scale reflection (Section 2.3), and reification in Coq-Elpi (Section 2.4).

2.1 Canonical structures

Canonical structures [38, 47, 57] make it possible to implement ad-hoc inference mechanisms in Coq by giving a particular form of hints [4] to the unification engine [63]. An interface to trigger such an inference is expressed as a record. For example, a record type declaration

```
Structure eqType := { eq_sort : Type; eq_op : eq_sort -> eq_sort -> bool }.
```

represents a type (`eq_sort`) equipped with a comparison function (`eq_op`). At the same time, `eqType` is an interface to relate a type to its canonical comparison function. `Structure` is just a synonym of `Record`, but we reserve the former for interfaces for canonical structure resolution. A hint can be given as a record instance. For example, an instance

```
Canonical nat_eqType : eqType := {| eq_sort := nat; eq_op := eqn |}.
```

allows us to type check `@eq_op _ 0%N 1%N`, where `0%N` and `1%N` are Peano natural numbers of type `nat` and `eqn` is the comparison function of type `nat -> nat -> bool`. Since `eq_op` is a record projection of type `(forall e : eqType, eq_sort e -> ...)`, supplying `0%N` as the second argument of `eq_op` requires solving a type equation `eq_sort ?e ≡ nat` to type check, where `?e` is a unification variable of type `eqType`. For a `Canonical` declaration, the system synthesizes a unification hint between the projections (`eq_sort` and `eq_op`) and the head symbols of the fields (`nat` and `eqn`), respectively. Therefore, the above equation is solved by instantiating `?e` with `nat_eqType`.

Additionally, declaring the `eq_sort` projection as an implicit coercion [46, 47, 56] allows us to use `T : eqType` in the context that expects a term of type `Type`, so that one may write `x : T` rather than `x : eq_sort T`.

```
Coercion eq_sort : eqType >-> Sortclass.
```

2.2 The hierarchy of mathematical structures in MathComp

We summarize some mathematical structures provided by the MathComp library below and illustrate the inheritance hierarchy they form in Figure 1. Note that `comRingType` and later structures are used only in Section 5. Each structure is defined as a record bundling a `Type` with operators and axioms as in `eqType` of Section 2.1. More details on the structures and their operators can be found in [18, Chapter and 4].

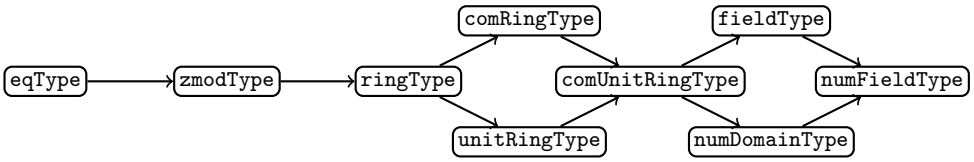


Figure 1 An excerpt of the hierarchy of mathematical structures in MathComp, where an arrow from `aType` to `bType` means that `bType` inherits from `aType`, e.g., `ringType` inherits from `zmodType`.

- `T : eqType` is a type whose propositional equality is decidable. The `eqType` record in Section 2.1 is a simplified version of this structure. For any `x` and `y` of type `T`, `x == y` (:= `eq_op x y`) tests if `x` is equal to `y`. Its negation can be expressed as `x != y`.
- `V : zmodType` is a $\mathbb{Z}$-module (additive Abelian group). For any `x` and `y` of type `V`, `x + y` (:= `GRing.add x y`), `- x` (:= `GRing.opp x`), and `0` (:= `GRing.zero V`) denotes the sum of `x` and `y`, the opposite of `x`, and zero, respectively.
- `R : ringType` is a ring. For any `x` and `y` of type `R`, `x * y` (:= `GRing.mul x y`) and `1` (:= `GRing.one R`) denotes the product of `x` and `y`, and one, respectively.
- `R : comRingType` is a commutative ring.
- `R : unitRingType` is a ring structure with computable inverses. For any `x` of type `R`, `x^-1` (:= `GRing.inv x`) denotes the multiplicative inverse of `x`, which is equal to `x` itself if `x` is not a unit, i.e., has no multiplicative inverse.
- `R : comUnitRingType` is a commutative ring with computable inverses.
- `F : fieldType` is a field.
- `R : numDomainType` is a partially ordered integral domain.
- `F : numFieldType` is a partially ordered field.

where "$\mathtt{E}_1$ (:= $\mathtt{E}_2$)" means that $\mathtt{E}_1$ is a notation [55] for $\mathtt{E}_2$, and they are syntactically equal. Each operator above takes a structure instance as its first argument, which is implicit except for `GRing.zero` and `GRing.one`.

These structures are defined by following packed classes, advocated by Garillot et al. [26] and also detailed in [1, 25, 39, 48]. For example, the `ringType` structure is defined as follows.

```
1  (* in Module GRing: *)
2  Module Ring.
3
4  Record mixin_of (R : zmodType) : Type :=
5    Mixin { one : R; mul : R -> R -> R; ... (* properties of one and mul *) }.
6
7  Record class_of (R : Type) : Type :=
8    Class { base : Zmodule.class_of R; mixin : mixin_of (Zmodule.Pack base) }.
9
10 Structure type : Type := Pack { sort : Type; class : class_of sort }.
11
12 Definition zmodType (cT : type) : zmodType :=
13   @Zmodule.Pack (sort cT) (base (class cT)).
14
15 End Ring.
16 Notation ringType := Ring.type.
```

The `GRing.Ring` module serves as a namespace qualifying the definitions inside the module, which are internals to define the `ringType` structure. Each structure has such a module, e.g., `GRing.Zmodule` is for `zmodType`. The structure is divided into three kinds of records: mixin (Line 4), class (Line 7), and structure (Line 10). The mixin record gathers operators and

axioms newly introduced by the structure, e.g., the multiplication, multiplicative identity, and their properties are required to define rings by extending $\mathbb{Z}$-modules. The class record assembles the mixins of the superclasses. The structure record is the actual interface of the structure that bundles a `Type` with its class instance.

`GRing.Ring.zmodType` is an explicit subtyping function that takes a `ringType` and returns its underlying `zmodType`, which can be made implicit by declaring it as a coercion.

```
Coercion Ring.sort : Ring.type >-> Sortclass.
Coercion Ring.zmodType : Ring.type >-> Zmodule.type.
```

Furthermore, declaring this subtyping function as a canonical instance allows us to write a term that mixes $\mathbb{Z}$-module and ring operators, e.g., `0 + 1`, by solving type equation of the form `GRing.Zmodule.sort ?V ≡ GRing.Ring.sort ?R`. In general, solving an equation `GRing.Ring.sort ?R ≡ T` gives us the ring instance `?R` of type `T`.

```
Canonical Ring.zmodType.
```

The ring operators are defined by lifting the projections of the mixin record to the structure record, as follows.

```
Definition one (R : ringType) : R := Ring.one (Ring.mixin (Ring.class R)).
Definition mul (R : ringType) : R -> R -> R := Ring.mul (Ring.mixin (Ring.class R)).
```

Packed classes can also express the hierarchy of morphisms. For example, the MathComp library provides the structure `{additive U -> V}` of additive functions ($\mathbb{Z}$-module homomorphisms) from `U` to `V`. Its record projection `GRing.Additive.apply` returns a function of type `U -> V` and is used for triggering instance resolution (e.g., Section 4.1) in the same way as `GRing.Ring.sort` above. Similarly, there is a structure of ring homomorphisms `{rmorphism R -> S}` which inherits from additive functions.

2.3 Large-scale reflection

This section demonstrates how to prove $\mathbb{Z}$-module equations by reflection. Firstly, we define the data type describing the syntax as follows:

```
Inductive AGExpr : Type :=
  | AGX : nat -> AGExpr
  | AGO : AGExpr                        (* zero *)
  | AGOpp : AGExpr -> AGExpr            (* opposite *)
  | AGAdd : AGExpr -> AGExpr -> AGExpr. (* addition *)
```

where `AGX j` means $\mathtt{j}^{\mathrm{th}}$ variable. This inductive data type allows us to write a Coq function manipulating the syntax. For example, we can interpret a syntax tree as follows:

```
Fixpoint AGeval (V : Type) (zero : V) (opp : V -> V) (add : V -> V -> V)
                (varmap : list V) (e : AGExpr) : V :=
  match e with
    | AGX j => nth zero varmap j
    | AGO => zero
    | AGOpp e1 => opp (AGeval zero opp add varmap e1)
    | AGAdd e1 e2 =>
      add (AGeval zero opp add varmap e1) (AGeval zero opp add varmap e2)
  end.
```

where the first four arguments are the carrier type and operators of a $\mathbb{Z}$-module, and `varmap` is a list whose $\mathtt{j}^{\text{th}}$ item gives the interpretation of the $\mathtt{j}^{\text{th}}$ variable. Such an object representing variable assignments is called a *variable map*.

Similarly, we can define a function `AGnorm` of type `AGExpr -> list int` that normalizes a syntax tree to a list of integers representing a formal sum, e.g., $[1; -2]$ represents $X_0 - 2X_1$, where `int` is the type of integers defined in MathComp. Their correctness specialized for the case that `V` is `int` can be stated as follows.

```
Lemma int_correct (varmap : list int) (e1 e2 : AGExpr) :
  (* if all the coefficients of the normal form of e1 - e2 is equal to 0, *)
  all (fun i => i == zeroz) (AGnorm (AGAdd e1 (AGOpp e2))) = true ->
  (* e1 and e2 evaluated to integers by AGeval are equal. *)
  AGeval zeroz oppz addz varmap e1 = AGeval zeroz oppz addz varmap e2.
```

where `zeroz`, `oppz`, and `addz` are $\mathbb{Z}$-module operators for `int`.

Suppose we want to prove a goal

```
(x + (- y)) + x = (- y) + (x + x)
```

for some `x y : int` where `+` and `-` here mean `addz` and `oppz`, respectively. Thanks to the above reflection lemma, the proof can be done by the following proof term.

```
let e1 := AGAdd (AGAdd (AGX 0) (AGOpp (AGX 1))) (AGX 0) in
let e2 := AGAdd (AGOpp (AGX 1)) (AGAdd (AGX 0) (AGX 0)) in
@int_correct [:: x; y] e1 e2 erefl.
```

Here we used computational reflection twice. Firstly, `e1` and `e2` are the reified terms representing the LHS and RHS of the goal. These terms interpreted by `(AGeval ... [:: x; y])` are convertible to the LHS and RHS, respectively. This conversion is triggered by applying the proof term `(@int_correct ...)` to the goal. Secondly, the nullity conditions `(all ... = true)` required by the reflection lemma `int_correct` is checked by reducing its LHS to `true`. This conversion is triggered by supplying `erefl` as the last argument of `int_correct` because `erefl` is the reflexivity proof of type `(forall x, x = x)` where the argument `x` can be left implicit. In the former case, unfolding too many constants may lead to performance issues, and conversion should be performed carefully. In the latter case, we can simply reduce the LHS to `true`, and this is the case where optimized reduction procedures such as `vm_compute` [30] and `native_compute` [10] can be useful.

2.4 Implementing reification in Coq-Elpi

To turn the above method into an automated proof tool, the reified terms and the variable map must automatically be obtained from the goal. Since we cannot pattern match on the operators such as `oppz` in the object level, this reification has to be done in the meta level.

In this section, we implement reification in Coq-Elpi. An example of an Elpi program follows.

```
1 pred mem o:list term, o:term, o:term.
2 mem [X|_]  X {{ O }}      :- !.
3 mem [_|XS] X {{ S lp:N }} :- !, mem XS X N.
```

In this code, we define a *predicate* `mem`. Line 1 is the type signature of `mem`, meaning that `mem` has three arguments of type `list term`, `term`, and `term`, respectively, where `term` is the type of Coq terms embedded in Elpi. Line 2 and 3 are two *rules* that define the meaning of `mem`. Capital identifiers such as `X`, `XS`, and `N` are unification variables. The syntax `[X|XS]` is a

cons cell of lists whose head and tail are `X` and `XS`, respectively. The syntaxes `{{ ... }}` and `lp:` are the quotation from Elpi to Coq and the antiquotation from Coq to Elpi, respectively. Therefore, these two rules are equivalent to the following:

```
mem [X|_]  X (global (indc «O»))          :- !.
mem [_|XS] X (app [global (indc «S»), N]) :- !, mem XS X N.
```

where `app` of type `list term -> term` is a constructor of `term` meaning an n-ary function application of Coq, and `global (indc _)` means a constructor of Coq.[1]

Actually, the proposition `mem XS X N` asserts that the `N + 1`$^{\text{th}}$ element of `XS` is `X`, where `N` is a Coq term of type `nat`. Let us consider an example `mem [Y, Z] Z M`, where `Y` and `Z` are distinct Coq terms and `M` remains unknown. The LHS of the first rule requires that the head of `XS` is `X`, but this does not apply to our example. Thus, it attempts matching with the second rule by solving equations `[_|XS] = [Y, Z]`, `X = Z`, and `{{ S lp:N }} = M`. Then we get `XS = [Z]` from the first equation, and proceed to execute its RHS `(!, mem [Z] Z N)`, which is the conjunction of the cut (`!`) operator and `mem [Z] Z N`. The cut operator prevents backtracking, i.e., trying other rules of `mem` when the later items of the conjunction fails. Since `mem [Z] Z N` matches with the first rule, `N` is instantiated with `{{ O }}`. In the end, our example `mem [Y, Z] Z M` succeeds with `S O` substituted to the variable `M`. Indeed, `Z` is the second element of `[Y, Z]`.

If the first argument `XS` is an open-ended list `[X0, ..., XN | XS']` where `XS'` remains unknown, and the given item `X` is none of the known elements, `mem XS X _` instantiates `XS'` with `[X | _]` and `X` becomes the `N + 2`$^{\text{th}}$ element of `XS`.

We implement reification as a predicate `quote`, such that `quote In Out VarMap` reifies `In` of type `int` to `Out` of type `AGExpr` and produces an open-ended variable map `VarMap`:

```
pred quote i:term, o:term, o:list term.
quote {{ zeroz }} {{ AGO }} _ :- !.
quote {{ oppz lp:In1 }} {{ AGOpp lp:Out1 }} VarMap :- !,
  quote In1 Out1 VarMap.
quote {{ addz lp:In1 lp:In2 }} {{ AGAdd lp:Out1 lp:Out2 }} VarMap :- !,
  quote In1 Out1 VarMap, quote In2 Out2 VarMap.
quote In {{ AGX lp:N }} VarMap :- !, mem VarMap In N.
```

where `i:` and `o:` stand for input and output, respectively. Marking an argument as input avoids instantiation of that argument. The first three rules of `quote` are just simple syntactic translation rules for the operators. If the input does not match with any of those, it should be treated as a variable by the last rule, which is implemented using the `mem` predicate above.

3 Large-scale reflection for packed classes

Thanks to the techniques reviewed in Section 2.3 and 2.4, we can implement a tactic `int_zmodule` for solving any integer equation that holds for any $\mathbb{Z}$-module. However, its generalization `poly_zmodule` to arbitrary $\mathbb{Z}$-modules, declared as instances of the `zmodType` structure, is actually not trivial. First, we describe a naive implementation that fails and analyze the source of the failure in Section 3.1. Then, we propose a solution to this issue based on the keyed matching discipline [28] in Section 3.2.

[1] Note that the actual Elpi syntax does not support «...» in input. This is just for illustration purposes.

3.1 Purely syntactic reification does not work for packed classes

We first generalize the correctness and reflection lemmas `int_correct` to any ℤ-module:

```
Lemma AG_norm_subst (V : zmodType) (varmap : list V) (e : AGExpr) :
  AGsubst 0 -%R +%R varmap (AGnorm e) = AGeval 0 -%R +%R varmap e.

Lemma AG_correct (V : zmodType) (varmap : list V) (e1 e2 : AGExpr) :
  all (fun i => i == 0) (AGnorm (AGAdd e1 (AGOpp e2))) = true ->
  AGeval 0 -%R +%R varmap e1 = AGeval 0 -%R +%R varmap e2.
```

where `AG_norm_subst` is the key lemma to prove `AG_correct`, `AGsubst` is the function to substitute a variable map to a formal sum, and `-%R` and `+%R` are 0-ary notations for `GRing.opp` and `GRing.add` implicitly applied to `V`, respectively.

To reimplement the `quote` predicate, we add a new argument `V` which is the `zmodType` instance for the type of the input term, and replace operators `zeroz`, `oppz`, and `addz` with `@GRing.zero V`, `@GRing.opp V`, and `@GRing.add V`, respectively.

```
pred quote i:term, i:term, o:term, o:list term.
quote V {{ @GRing.zero lp:V }} {{ AGO }} _ :- !.
quote V {{ @GRing.opp lp:V lp:In1 }} {{ AGOpp lp:Out1 }} VarMap :- !,
  quote V In1 Out1 VarMap.
quote V {{ @GRing.add lp:V lp:In1 lp:In2 }} {{ AGAdd lp:Out1 lp:Out2 }} VarMap :- !,
  quote V In1 Out1 VarMap, quote V In2 Out2 VarMap.
quote _ In {{ AGX lp:N }} VarMap :- !, mem VarMap In N.
```

However, this `quote` predicate fails to reify at least one addition operator in the goal:

```
forall x : int, x + 1 = 1 + x.
```

Let us take a closer look at it by `Set Printing All`:

```
forall x : int,
@eq (GRing.Zmodule.sort int_ZmodType)
  (@GRing.add int_ZmodType x (GRing.one int_Ring))
  (@GRing.add (GRing.Ring.zmodType int_Ring) (GRing.one int_Ring) x)
```

where `int_ZmodType` and `int_Ring` are the canonical `zmodType` and `ringType` instances of `int`, respectively.

The root of the issue is that the two occurrences of `GRing.add` take syntactically different `zmodType` instances as highlighted. The former instance is inferred from the type of `x`, by solving the type equation `GRing.Zmodule.sort ?V ≡ int`. The latter instance is inferred from the type of `GRing.one ?R` where `?R` is eventually instantiated with `int_Ring`, by solving the type equation `GRing.Zmodule.sort ?V ≡ GRing.Ring.sort ?R` whose solution is `?V := GRing.Ring.zmodType ?R`. The `quote` predicate above requires that all the `zmodType` instances occurring as the first argument of the operators are syntactically equal to each other. However, the above goal does not respect this restriction. In the presence of the inheritance mechanism of packed classes, such syntactically different instances for the same type and structure coexist [1, Section 3.1][26, Section 2.4][48, Section 3], and canonical structure resolution may infer them simultaneously. Nevertheless, definitional equality of those instances is ensured by *forgetful inheritance* [1], that is, the practice of implementing inheritance and subtyping functions by record inclusion and erasure of some record fields, respectively.

3.2 Reification by small-scale reflection

Reification recognizing operators by conversion or unification rather than purely syntactic matching would address the above issue. However, using full unification for term matching, e.g., triggering unification `@GRing.opp V ?t' ≜ t` to check if `t` is the opposite of an unknown term `?t'`, can make reification too costly. Thus, we propose a solution that mixes syntactic matching and conversion as in the keyed matching discipline [28]. The idea of keyed matching is to find a subterm that matches with a pattern `(f t1 ... tn)` by attempting the matching operation only on subterms of the form `(f t1' ... tn')`. While the head constant (the key) `f` has to be the same constant,[2] its arguments can be compared by conversion or unification.

In our case, the keys are the $\mathbb{Z}$-module operators `GRing.zero`, `GRing.opp`, and `GRing.add`. The `quote` predicate can be reimplemented as follows:

```
1 pred quote i:term, i:term, o:term, o:list term.
2 quote V {{ @GRing.zero lp:V' }} {{ AG0 }} _ :- coq.unify-eq V V' ok, !.
3 quote V {{ @GRing.opp lp:V' lp:In1 }} {{ AGOpp lp:Out1 }} VarMap :-
4   coq.unify-eq V V' ok, !, quote V In1 Out1 VarMap.
5 quote V {{ @GRing.add lp:V' lp:In1 lp:In2 }} {{ AGAdd lp:Out1 lp:Out2 }} VarMap :-
6   coq.unify-eq V V' ok, !, quote V In1 Out1 VarMap, quote V In2 Out2 VarMap.
7 quote _ In {{ AGX lp:N }} VarMap :- !, mem VarMap In N.
```

where `coq.unify-eq V V' ok` asserts that `V` unifies with `V'`. Since the first argument `V` and the input term do not have any unification variable under normal use of `quote`, this unification problem falls in a conversion problem that is generally easier and less costly to solve than unification. For example, the second rule of `quote` (Line 3) does not require `V'` in the input term `@GRing.opp V' In1` to be syntactically equal to the first argument `V`, but it compares `V'` with `V` by conversion after syntactic matching of the opposite operator `GRing.opp`. Since this conversion is a part of term matching, the cut operator to prevent backtracking comes after conversion.

The `zmodType` instance to use as the first argument of `quote` can be obtained by canonical structure resolution. This inference is implemented as follows.

```
1 pred solve i:goal, o:list sealed-goal.
2 solve (goal _ _ {{ @eq lp:Ty lp:T1 lp:T2 }} _ _ as G) GS :-
3   std.assert-ok! (coq.unify-eq {{ GRing.Zmodule.sort lp:V }} Ty)
4                  "Cannot find a declared Z-module", !,
5   quote V T1 ZE1 VarMap, !, quote V T2 ZE2 VarMap, !,
6   ...
```

The `solve` predicate is the entry point of a tactic in Coq-Elpi. The above rule matches the goal proposition with a pattern `T1 = T2` where `T1` and `T2` have type `Ty` (Line 2), triggers unification `GRing.Zmodule.sort V ≜ Ty` to find the canonical `zmodType` instance `V` of `Ty` (Line 3), and then reifies `T1` to `ZE1` and `T2` to `ZE2` using `V` obtained in the second step (Line 5). Note that if unification by `coq.unify-eq` fails, its third argument of type `diagnostic` carries the error message. The `std.assert-ok!` predicate of Line 3 asserts that unification given as the first argument succeeds, but if it fails, it prints the carried error message with the string given as the second argument.

[2] In the actual implementation of keyed matching in the SSReflect plugin, there are some exceptions such that a projection of a canonical structure can match with its canonical instances. See [28, Section 3.1] for details.

4 Extending the syntax with homomorphisms and more operators

In this section, we implement a new tactic `morph_zmodule`, that extends the syntax supported by `poly_zmodule` with $\mathbb{Z}$-module homomorphisms (Section 4.1) and subtraction (Section 4.2) which is not directly supported the syntax `AGExpr`. These extensions are achieved by adding another layer of reflection which we call *preprocessing*. This twofold reflection scheme allows us to reuse the syntax `AGExpr`, interpretation and normalization procedures `AGeval` and `AGnorm`, and the reflection lemma `AG_correct` presented in Section 2 and 3 as is.

4.1 Homomorphisms

Firstly, we define another inductive type describing the syntax involving homomorphisms.

```
1 Implicit Types (U V : zmodType).
2
3 Inductive MExpr : zmodType -> Type :=
4   | MX V : V -> MExpr V
5   | MO V : MExpr V
6   | MOpp V : MExpr V -> MExpr V
7   | MAdd V : MExpr V -> MExpr V -> MExpr V
8   | MMorph U V : {additive U -> V} -> MExpr U -> MExpr V.
```

The main difference of this type compared with `AGExpr` is that: `MExpr` (Line 3) is parameterized by a `zmodType` instance `V`, the constructor `MX` (Line 4) representing a variable takes a term of type `V` instead of an index of type `nat`, and the constructor `MMorph` (Line 8), representing a homomorphism application, allows for changing the parameter `V`. Therefore, one shall interpret a reified term of this type without a variable map provided.

```
1 Fixpoint Meval V (e : MExpr V) : V :=
2   match e with
3     | MX _ x => x
4     | MO _ => 0
5     | MOpp _ e1 => - Meval e1
6     | MAdd _ e1 e2 => Meval e1 + Meval e2
7     | MMorph _ _ f e1 => f (Meval e1)
8   end.
```

The normalization procedure we need for `MExpr` is just pushing down homomorphisms appearing as the `MMorph` constructor to the leaves of the syntax tree:

```
1 Fixpoint Mnorm U V (f : {additive U -> V}) (e : MExpr U) : V :=
2   match e in MExpr U return {additive U -> V} -> V with
3     | MX _ x => fun f => f x
4     | MO _ => fun _ => 0
5     | MOpp _ e1 => fun f => - Mnorm f e1
6     | MAdd _ e1 e2 => fun f => Mnorm f e1 + Mnorm f e2
7     | MMorph _ _ g e1 => fun f => Mnorm [additive of f \o g] e1
8   end f.
```

where the third argument `(f : {additive U -> V})` accumulates homomorphisms applied to `e`. Therefore, the case for `e := MMorph _ _ g e1` (Line 7) constructs a homomorphism `[additive of f \o g]` that is the function composition of `f` and `g`, and passes it to the recursive call for normalizing `e1`. On the other hand, the case for `e := MX _ x` (Line 3) applies `f` to the variable `x`. Since dependent pattern matching on `e : MExpr U` forces instantiation of `U` in type checking of each clause, defining `Mnorm` that type checks requires the so-called *convoy pattern* [15] to propagate this instantiation to the type of `f`.

Thanks to the structure preservation laws of homomorphisms, a result of normalization `Mnorm f e` should be equal to `f` applied to `Meval e`. That is to say, the following correctness lemma holds:

```
1  Lemma M_correct V (e : MExpr V) : Meval e = Mnorm [additive of idfun] e.
```

where `[additive of idfun]` is the identity homomorphism.

The reification procedure for the `morph_zmodule` tactic should take an input term `In` of type `V : zmodType`, and obtain a variable map `varmap` and two reified terms `OutM` and `Out` of types `MExpr V` and `AGExpr`, respectively. For any such Coq terms, the following chain of equations should hold to justify the completeness of the tactic:

$$
\begin{aligned}
\texttt{In} &\equiv \texttt{Meval OutM} && \text{(a meta property)}\\
&= \texttt{Mnorm [additive of idfun] OutM} && \text{(Lemma M\_correct)}\\
&\equiv \texttt{AGeval ... varmap Out} && \text{(a meta property)}\\
&= \texttt{AGsubst ... varmap (AGnorm Out)} && \text{(Lemma AG\_norm\_subst)}
\end{aligned}
$$

where $\equiv$ and $=$ respectively mean definitional equality and propositional equality. Although these meta properties of reification cannot be proved inside Coq, the kernel of Coq will check them for every invocation of the tactic, as explained in Section 2.3.

Considering the above requirements, reification can be reimplemented as follows.

```
1  pred quote i:term, i:(term -> term), i:term, o:term, o:term, o:list term.
2  quote V _ {{ @GRing.zero lp:V' }} {{ MO lp:V }} {{ AGO }} _ :-
3    coq.unify-eq V V' ok, !.
4  quote V F {{ @GRing.opp lp:V' lp:In1 }}
5        {{ @MOpp lp:V lp:OutM1 }} {{ AGOpp lp:Out1 }} VarMap :-
6    coq.unify-eq V V' ok, !, quote V F In1 OutM1 Out1 VarMap.
7  quote V F {{ @GRing.add lp:V' lp:In1 lp:In2 }}
8        {{ @MAdd lp:V lp:OutM1 lp:OutM2 }} {{ AGAdd lp:Out1 lp:Out2 }} VarMap :-
9    coq.unify-eq V V' ok, !,
10   quote V F In1 OutM1 Out1 VarMap, quote V F In2 OutM2 Out2 VarMap.
11 quote V F In {{ @MMorph lp:U lp:V lp:G lp:OutM }} Out VarMap :-
12   coq.unify-eq {{ @GRing.Additive.apply lp:U lp:V lp:Ph lp:G lp:In1 }} In ok, !,
13   quote U (x\ F {{ @GRing.Additive.apply lp:U lp:V lp:Ph lp:G lp:x }})
14         In1 OutM Out VarMap.
15 quote V F In {{ @MX lp:V lp:In }} {{ AGX lp:N }} VarMap :- !,
16   mem VarMap (F In) N.
```

Let the six arguments of the new `quote` predicate be `V`, `F`, `In`, `OutM`, `Out`, and `VarMap`. The first three arguments are input: `V` is a `zmodType` instance, `F` is a homomorphism from `V` to another `zmodType` instance, and `In` is the input term of type `V`. Then, the last three arguments are output: `OutM` and `Out` are the reified terms of types `MExpr V` and `AGExpr`, respectively, and `VarMap` is the variable map. The second argument `F` is required to make recursion of `quote` work and accumulates homomorphisms as in the third argument `f` of `Mnorm`. Note that `F` is represented as an Elpi function from `term` to `term`, which lets us compose functions without leaving a beta redex in the Coq level. While the first reified term `OutM` exactly corresponds to `In`, the second reified term `Out` and the variable map corresponds to `F In`.

The most crucial part of the new `quote` predicate is its fourth rule (Line 11), which handles the case that the input `In` is a homomorphism application. It first triggers unification `@GRing.Additive.apply U V _ G In1 ≜ In` to decompose the input into the homomorphism instance `G` and its argument `In1`. Then, since `G` is a homomorphism from `U` to `V`,

it invokes the recursive call of `quote` on `In1` with `U` as the first argument (the `zmodType` instance) and the composition of `F` and `G` as the second argument (the homomorphism). This composition is written as `(x\ ...)` which means an abstraction ($\lambda x.\ \ldots$).

4.2 More operators

Based on our twofold reflection scheme, we can add support for operators not directly supported by the syntax `AGExpr`. For example, let `subr` be an opaque subtraction operator of type `(forall U : zmodType, U -> U -> U)`. By opaque we mean that `subr` does not reduce and thus we cannot rely on its definitional behavior, but we can reason about it through a lemma:

```
subrE : @subr = (fun (U : zmodType) (x y : U) => x + (- y)).
```

Firstly, we add the following constructor to `MExpr` representing `subr`.

```
(* in Inductive MExpr: *)
  | MSub V : MExpr V -> MExpr V -> MExpr V
```

Then, the interpretation and normalization function have to be adapted to the new definition of `MExpr`, by adding the following cases.

```
(* in Fixpoint Meval: *)
    | MSub _ e1 e2 => subr (Meval e1) (Meval e2)
```

```
(* in Fixpoint Mnorm: *)
    | MSub _ e1 e2 => fun f => Mnorm f e1 + (- Mnorm f e2)
```

The point is that `Meval` interprets `MSub V e1 e2` using `subr`, but `Mnorm` normalizes it using `GRing.add` and `GRing.opp`. Proving the correctness lemma `M_correct` based on these new definitions can be done using the lemma `subrE`. These definitions and the correctness lemma let us replace subexpressions of the form `subr e1 e2` with `e1' + (- e2')` by preprocessing, and make it possible to generate a corresponding reified term of type `AGExpr`. Therefore, an input term of the form `subr _ _` has to be reified to `@MSub _ _ _` and `AGAdd _ (AGOpp _)`, as follows (see Line 2).

```
quote V F {{ @subr lp:V' lp:In1 lp:In2 }}
      {{ @MSub lp:V lp:OutM1 lp:OutM2 }} {{ AGAdd lp:Out1 (AGOpp lp:Out2) }}
      VarMap :-
  coq.unify-eq V V' ok, !,
  quote V F In1 OutM1 Out1 VarMap, quote V F In2 OutM2 Out2 VarMap.
```

In fact, even if an operator can be supported by relying on its definitional behavior, our methodology is sometimes performance-wise better than doing so. For example, `n%:R` (:= `1 *+ n`) and `n%:~R` (:= `1 *~ n`) are generic embeddings of `n : nat` and `n : int` to a ring, respectively, where `x *+ n` (:= `GRing.natmul x n`) and `x *~ n` (:= `intmul x n`) are `n` times addition of `x : V` defined for any `V : zmodType`. For any `n` of type `nat`, `n%:R` and `(Posz n)%:~R` are convertible since the latter unfolds to the former, where `Posz` is a constructor of `int` that embeds `nat` to `int`. Therefore, if a reflexive tactic supports `n%:~R`, it is possible to support `n%:R` by reifying it in the same way as `(Posz n)%:~R`. However, it may lead to performance issues by triggering conversions such as:

```
Time Check erefl : (Posz 6 * 6)%:~R = 36%:R :> rat. (* 36.364s *)
```

where `:> rat` means that the LHS and RHS are rational numbers of type `rat`.[3]

[3] Note that this particular performance issue reproduces only with MathComp 1.12.0 or earlier.

The source of this inefficiency is that conversion unfolds too many constants. Computations involving rational numbers are particularly inefficient because `rat` is defined as a dependent pair of the numerator and denominator that are coprime [18, Section 4.4.2] and every `rat` operator performs GCD calculation to ensure the canonicity of representations. In our reflection scheme, conversion between `GRing.natmul` and `intmul` can be hidden in preprocessing. It makes conversion performing only a small number of unfolding and thus more efficient.

5 Applications: `ring`, `field`, and the irrationality of $\zeta(3)$

As an application of the methodology presented in Section 3 and 4, we briefly report our effort to adapt the `ring` and `field` tactics [31, 60] of Coq to the commutative ring and field structures of MathComp in Section 5.1. The `field` tactic generates proof obligations describing the non-nullity of the denominators in the given equation. Those conditions can often be simplified to equivalent integer disequations and solved by the `lia` tactic. In Section 5.2, we implement this simplification based on the approach of Gonthier et al. [29] to use canonical structures for proof automation. In Section 5.3, we apply the above proof tools to the formal proof of Apéry's theorem by Chyzak, Mahboubi, and Sibut-Pinote [16, 17, 37] to bring more proof automation. Our `ring` and `field` tactics for MathComp are available as a Coq library called *Algebra Tactics* [49].

5.1 The `ring` and `field` tactics

The `ring` and `field` tactics [31, 60] of Coq respectively solve polynomial and rational equations by computational reflection. Their reflexive decision procedures are based on normalization to the *sparse Horner form* [31], a multivariate, computationally efficient version of the Horner normal form of polynomials.

The following inductive type describes the syntax supported by the `ring` tactic:

```
Inductive PExpr (C : Type) : Type :=
  | PEO : PExpr C                          (* zero:           GRing.zero *)
  | PEI : PExpr C                          (* one:            GRing.one  *)
  | PEc : C -> PExpr C                     (* constant:       _%:~R      *)
  | PEX : positive -> PExpr C              (* variable                   *)
  | PEadd : PExpr C -> PExpr C -> PExpr C  (* addition:       GRing.add  *)
  | PEsub : PExpr C -> PExpr C -> PExpr C  (* subtraction:    _ - _      *)
  | PEmul : PExpr C -> PExpr C -> PExpr C  (* multiplication: GRing.mul  *)
  | PEopp : PExpr C -> PExpr C             (* opposite:       GRing.opp  *)
  | PEpow : PExpr C -> N -> PExpr C.       (* power:          GRing.exp  *)
```

where `C` is the type of coefficients and fixed to the binary integer type `Z` of the Coq standard library in our usage. For each constructor, its meaning and the corresponding operator in MathComp are indicated in the code comment left. Note that `n%:~R` is the generic embedding of `n : int` to a ring explained in Section 4.2, and `x ^+ n` (:= `GRing.exp x n`) is the n^{th} power of `x` with `n : nat`. There is also `x ^ n` (:= `exprz x n`) operator, namely, n^{th} power of `x` with `n : int`, which works only for `unitRingType`.

In addition to the above constructs, the `field` tactic supports the following two operators.

```
(* in Inductive FExpr: *)
  | FEinv : FExpr C -> FExpr C             (* inverse:        GRing.inv  *)
  | FEdiv : FExpr C -> FExpr C -> FExpr C  (* division:       _ / _      *)
```

On top of these syntaxes, we implemented preprocessors to support homomorphisms and more operators such as `GRing.natmul`, `intmul`, and `exprz`. Since rings and fields have poorer structures such as $\mathbb{Z}$-modules, subexpressions of these structures may appear under homomorphism applications. For example, let `f : V -> R` be an additive function whose codomain `R` is a ring, and we want to perform the following equational reasoning in preprocessing:

```
  f (x *~ (n * m))
= f x * (n * m)%:~R
= f x * (n%:~R * m%:~R).
```

This example indicates that ring multiplication may appear in a $\mathbb{Z}$-module subexpression of a ring expression, and homomorphisms can be pushed down through it.

Therefore, we defined three inductive types describing the syntax: `NExpr` for expressions of type `nat`, `RExpr` for ring expressions, and `ZMExpr` for $\mathbb{Z}$-module expressions. The latter two are defined as mutually inductive types. `RExpr` contains constructors for field operators and is used for both `ring` and `field` tactics. Since a ring homomorphism can be pushed down through these operators only if the codomain of the homomorphism is a field, we define normalization functions for `RExpr` and `ZMExpr` for each of the `ring` and `field` tactics separately.

5.2 Automating proofs of non-nullity conditions for `field`

The `field` tactic can now solve a goal

```
((n ^+ 2)%:~R - 1) / (n%:~R - 1) = (n%:~R + 1) :> F
```

where `F` is a field. The `field` tactic then generates a proof obligation `n%:~R - 1 != 0 :> F` describing the non-nullity of the denominator in the equation. If `F` is a partially ordered field (`numFieldType`), this obligation can be simplified to `n != 1 :> int` because `_%:~R` for any partially ordered integral domain (`numDomainType`) is injective. The simplified obligation can sometimes be solved by other automated tactics such as `lia` [6, 59], which can solve linear goals over integers. This combination of the `field` and `lia` tactics is extensively used in the formal proof of Apéry's theorem [17, Section 4.3][37, Section 2.4].

Note that the `lia` tactic has another preprocessing tactic, called `zify`, that canonizes the goal by mapping numeric types such as `nat` to the binary integer type `Z`. The `zify` tactic is extensible by declaring type class instances explaining how to map types and operators in preprocessing. We implemented another small library called Mczify [50] to use this feature to support the arithmetic operators of MathComp in the `lia` tactic.

In this section, we reimplement this simplification based on the approach of Gonthier et al. [29]. Firstly, we define a canonical structure `zifyRing` that relates a ring expression that is an element of the integer subring (`rval : R`), to the corresponding integer expression (`zval : int`) such that `rval = zval%:~R`.

```
Section ZifyRing.

Variable R : ringType.

Structure zifyRing :=
  ZifyRing { rval : R; zval : int; zifyRingE : rval = zval%:~R }.
```

For instance, the `zifyRing` record allows us to relate `0` and `1` of type `R` to `0` and `1` of type `int`, respectively, as follows.

```
Canonical zify_zero := @ZifyRing 0 0 (erefl : 0 = 0%:~R).
Canonical zify_one  := @ZifyRing 1 1 (erefl : 1 = 1%:~R).
```

Since the integer subring is closed under opposite, `- x = (- n)%:~R` holds if `x = n%:~R`. This implication can be encoded as a canonical instance that takes another instance as an argument, as follows.

```
Lemma zify_opp_subproof (e1 : zifyRing) : - rval e1 = (- zval e1)%:~R.

Canonical zify_opp (e1 : zifyRing) :=
  @ZifyRing (- rval e1) (- zval e1) (zify_opp_subproof e1).
```

Similarly, the closure properties under `GRing.add`, `GRing.mul`, and `intmul` can be implemented as the following instances.

```
Canonical zify_add e1 e2 := @ZifyRing (rval e1 + rval e2) (zval e1 + zval e2) ...
Canonical zify_mul e1 e2 := @ZifyRing (rval e1 * rval e2) (zval e1 * zval e2) ...
Canonical zify_mulrz e1 n := @ZifyRing (rval e1 *~ n) (zval e1 *~ n) ...
```

In general, solving an equation `rval ?e1 ≜ x` gives us an integer expression `n` and its correctness proof of `x = n%:~R` from a ring expression `x`. Let us consider an example `x := 1 + n%:~R *~ 2`. Solving the equation `rval ?e1 ≜ x` proceeds by instantiating `?e1` with `zify_add ?e2 ?e3` since the head symbol of `x` is `GRing.add`, and then the problem is divided into two sub-problems `rval ?e2 ≜ 1` and `rval ?e3 ≜ n%:~R *~ 2`. Solving the former sub-problem is done by instantiating `?e2` with `zify_one`, solving the latter proceeds by instantiating `?e3` with `zify_mulrz ?e4 2`, and we get another sub-problem `rval ?e4 ≜ n%:~R`. By repeating this recursive process, we eventually get the canonical solution `?e4 := zify_mulrz zify_one n`. The `zval` and `zifyRingE` fields of the solution `?e1 := zify_add zify_one (zify_mulrz (zify_mulrz zify_one n) 2)` give us the integer expression and the proof, respectively.

Reducing a ring (dis)equation to an integer (dis)equation is performed by rewriting the ring equation by the following lemma.

```
End ZifyRing.

Lemma zify_eqb (R : numDomainType) (e1 e2 : zifyRing R) :
  (rval e1 == rval e2) = (zval e1 == zval e2).
```

For example, combining the above lemma and the `lia` tactic allows us to solve the following goal. We use a small Ltac [22] script to perform this proof automation in practice.

```
Goal forall n : int, n%:~R *~ 2 + 1 != 0 :> rat.
Proof. move=> n; rewrite zify_eqb /=; lia. Qed.
```

5.3 The irrationality of $\zeta(3)$

This section briefly reports the result of applying the proof tools presented in the previous sections to the formal proof of Apéry's theorem [16, 17, 37]. This proof involves various collections of numbers such as integers, rational numbers `rat`, their real closure `realalg`, algebraic numbers `algC`, and Cauchy sequences. These types are equipped with ring instances except for Cauchy sequences and also with field instances except for integers. Therefore, the embedding functions corresponding to their inclusion, e.g., $\mathbb{Z} \subset \mathbb{Q}$, are ring homomorphisms. Since the type of integers `int` of MathComp is defined based on Peano natural numbers `nat`, it is well suited for proofs but prevents us from performing computation involving large integer constants in a reasonable time. Therefore, this proof also uses the binary representation of

Table 1 Performance comparison of the `rat_field` tactic and our `field` tactic in `ops_for_b.v` of the formal proof of Apéry's theorem, which proves a recurrence equation satisfied by a sequence called b [17, Section 4]. The size of a problem is the number of constructors of reified terms of type `FExpr`. Note that Problem #1 has many relatively large coefficients greater than 10^{12}. Therefore, the ratio of time spent for its normalization is higher than those of the other problems, and thus its improvement in execution time is relatively minor.

Lemma	# Problem	Size	`rat_field` Time (s)	`field` Time (s)
`P_eq_Delta_Q`	1	14,690	91.807	87.013
`recAperyB`	2	8,407	4.170	2.068
`recAperyB`	3	113,657	39.676	26.963

integers `Z` for computation purposes and defines a function that embeds `Z` to rational numbers `rat_of_Z : Z -> rat`. This embedding function is made opaque to prevent computing in `rat`.

We managed to replace two tactics in this proof with our tools: `rat_field` adapting the `field` tactic to MathComp, and `goal_to_lia` implementing the reduction of Section 5.2, both of which are implemented in Ltac and specific to rational numbers `rat`. This replacement is done by adding the support for `Z` constants and operators to our `field` tactic and by making `rat_of_Z` canonically a ring homomorphism. That is to say, we did not have to implement any treatment specific to this proof to our tools, since supporting large integer constants is considered to be of general interest. Moreover, our `ring` and `field` tactics can reason about any ring and field instances and ring homomorphisms. Thus, they can solve a broader range of subgoals, and some manual labor before or after invoking them, e.g., tweaking ring homomorphisms, has been handed off to our tools.

Our tools not only automate more proofs but also, directly and indirectly, make proofs more concise and faster to check. To give some figures, Table 1 summarizes the performance of the invocations of `rat_field` and `field` that take more than 1 second in the proof. In those cases, `field` is consistently faster than `rat_field`. Moreover, by extensively refactoring proofs using our tools, we could reduce 442 lines of specifications and proofs out of 5829 lines excluding code for proof automation, and checking the entire proof became 27% faster (6 min 52 s) than before (9 min 23 s).

On the other hand, we still see some room for improvement in this refactoring work. For example, our `field` tactic cannot directly solve an equation that has rational exponents, e.g., $x^{\frac{3}{2}}$, or variables in exponents [5], e.g., $x^{n+m} = x^n x^m$. However, they require reimplementing reflexive decision procedures and are pretty orthogonal to the present work, except that it might be possible to implement incomplete support for the latter case in preprocessing.

6 Conclusion

We proposed a methodology for building reflexive tactics and their concrete implementations in Coq-Elpi that cooperate with algebraic structures (Section 3) and their homomorphisms (Section 4.1) represented by packed classes. The issue we solved in Section 3 is not specific to packed classes, as the issue solved by forgetful inheritance [1] also appears in semi-bundled [61, Section 4.1] type classes [51]. On the other hand, purely syntactic reification works fine with unbundled [52] type classes, where operators appear as parameters of interfaces, as is the case in [13, 14]. However, this approach does not scale up to larger hierarchies, e.g., as noted in [14, Section 6.1]. Reification by small-scale reflection (Section 3.2) has also been adopted

in the `zify` tactic [8], which is the key ingredient to enable support for overloaded arithmetic operators of MathComp in the `lia` tactic (Section 5.2). Reification by small-scale reflection is also applicable to reification by parametricity [32], although it does not deal with variable maps and thus does not fit our purpose. Such implementation can be done by using the `ssrpattern` tactic in place of the `pattern` tactic, but it may not preserve the efficiency of reification by parametricity.

We argue that Coq-Elpi turned out to be a practical tool to implement our methodology, and in particular, provides features that made our reification procedures concise, although we could reimplement our tactics with other meta-languages such as OCaml, Ltac [22], Ltac2 [43], and Mtac2 [35]. For example, the cut operator, which is unavailable in type classes [52, Section 9] and Ltac, offers a pretty intuitive way to control backtracking and is a key to achieve efficiency. Quotation and antiquotation allow us to embed Coq terms with holes in our Elpi code in a readable way. Moreover, the results in Section 5.3 show that our resulting tactics run in reasonable times.

Our twofold reflection scheme and its preprocessing step allow us to adapt an existing reflexive tactic to homomorphisms (Section 4.1) and new operators (Section 4.2) without reimplementing the whole tactic. While other preprocessing tools such as `zify`, `ppsimpl` [7], and `trakt` [9] have been implemented as standalone tactics, our approach avoids performing reification twice by binding the preprocessor tightly with the reflexive decision procedure and thus has a performance advantage. This idea has been mentioned in [7, Section 3]:

> As future work, we consider binding `ppsimpl` more tightly with other reflective tactic such as `ring` or `lia` thus avoiding to perform a reification twice.

On the other hand, the downside of our approach is that it is less modular and does not let users extend an existing preprocessor with new rules. Since Coq-Elpi provides the abilities to generate inductive data types, Coq constants, and Elpi rules, we could improve this situation by writing an Elpi program that produces a reflexive preprocessor and reification rules from their high-level descriptions. Furthermore, we could integrate such an enhancement to Hierarchy Builder [21] to utilize metadata about the hierarchy of structures in reification.

As an application of our methodology, we adapted the `ring` and `field` tactics of Coq to the commutative rings and fields of MathComp (Section 5.1). We demonstrated their practicality and scalability by applying them to the formal proof of Apéry's theorem (Section 5.3). Although their reflexive decision procedures are not our contribution, we found room for improvement on this point. For example, the `ring_exp` tactic [5] of the Lean [42] mathematical library [61] solves ring equations with variables in exponents, which is one of the cases we wished to solve in Section 5.3. The `ring_exp` tactic does not directly support homomorphisms, but the `simp` and `norm_cast` tactics [36] serve as preprocessors for pushing down and up homomorphisms as in Section 4.1 and 5.2. Since those Lean tactics can be performed alone as in the `ppsimpl` tactic, the above comparison of our approach and other preprocessing tools also applies here. Also, those Lean tactics are not reflexive and produce proof terms explaining rewriting steps. While such implementation does not require proving the procedure correct and is regarded as performance-wise better than reflection in Lean, it would not scale up to large equations and expressions that have large normal forms. On the other hand, implementing efficient tactics using computational reflection requires verified and efficient procedures involving computation-oriented data structures such as sparse Horner form [31]. Cohen and Rouhling [20] proposed a modular approach to define and reason about efficient decision procedures using CoqEAL refinement framework [19, 23], which is a suitable candidate method for extensively developing reflexive tactics for mathematical structures of MathComp.

References

1 Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, and Kazuhiko Sakaguchi. Competing inheritance paths in dependent type theory: A case study in functional analysis. In *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2020. doi:10.1007/978-3-030-51054-1_1.

2 Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William E. Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*, pages 95–105. IEEE Computer Society, 1990. doi:10.1109/LICS.1990.113737.

3 Roger Apéry. Irrationalité de $\zeta(2)$ et $\zeta(3)$. *Astérisque*, 61, 1979. Société Mathématique de France.

4 Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in unification. In *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 84–98. Springer, 2009. doi:10.1007/978-3-642-03359-9_8.

5 Anne Baanen. A Lean tactic for normalising ring expressions with exponents (short paper). In *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 21–27. Springer, 2020. doi:10.1007/978-3-030-51054-1_2.

6 Frédéric Besson. Fast reflexive arithmetic tactics the linear case and beyond. In *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, volume 4502 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2006. doi:10.1007/978-3-540-74464-1_4.

7 Frédéric Besson. ppsimpl: a reflexive Coq tactic for canonising goals. In *The Third International Workshop on Coq for Programming Languages, CoqPL 2017*, 2017. URL: https://popl17.sigplan.org/details/main/3/ppsimpl-a-reflexive-Coq-tactic-for-canonising-goals.

8 Frédéric Besson. Fix #10779 (hnf normalisation of instance + reification of overloaded operators), 2019. Accessed: 2022-04-27. URL: https://github.com/coq/coq/pull/10787.

9 Valentin Blot, Denis Cousineau, Enzo Crance, Louise Dubois de Prisque, Chantal Keller, Assia Mahboubi, and Pierre Vial. Modular pre-processing for automated reasoning in dependent type theory, 2022. doi:10.48550/ARXIV.2204.02643.

10 Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full reduction at full throttle. In *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.

11 Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *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.

12 Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda - a functional language with dependent types. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer, 2009. doi:10.1007/978-3-642-03359-9_6.

13 Thomas Braibant and Damien Pous. Tactics for reasoning modulo AC in Coq. In *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 167–182. Springer, 2011. doi:10.1007/978-3-642-25379-9_14.

14 Thomas Braibant and Damien Pous. Deciding Kleene algebras in Coq. *Logical Methods in Computer Science*, 8(1), 2012. doi:10.2168/LMCS-8(1:16)2012.

15 Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013. URL: http://mitpress.mit.edu/books/certified-programming-dependent-types.

16 Frédéric Chyzak, Assia Mahboubi, and Thomas Sibut-Pinote. Apery: A formal proof of the irrationality of zeta(3), the Apéry constant, 2020. Accessed: 2022-02-08. URL: https://github.com/coq-community/apery.

17 Frédéric Chyzak, Assia Mahboubi, Thomas Sibut-Pinote, and Enrico Tassi. A computer-algebra-based formal proof of the irrationality of $\zeta(3)$. In *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 160–176. Springer, 2014. doi:10.1007/978-3-319-08970-6_11.

18 Cyril Cohen. *Formalized algebraic numbers: construction and first-order theory. (Formalisation des nombres algébriques : construction et théorie du premier ordre)*. PhD thesis, École Polytechnique, Palaiseau, France, 2012. URL: https://tel.archives-ouvertes.fr/pastel-00780446.

19 Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In *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 147–162. Springer, 2013. doi:10.1007/978-3-319-03545-1_10.

20 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*, 2017. URL: https://hal.inria.fr/hal-01414881.

21 Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy builder: Algebraic hierarchies made easy in Coq with Elpi (system description). In *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPIcs*, pages 34:1–34:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.FSCD.2020.34.

22 David Delahaye. A tactic language for the system Coq. In *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95. Springer, 2000. doi:10.1007/3-540-44404-1_7.

23 Maxime Dénès, Anders Mörtberg, and Vincent Siles. A refinement-based approach to computational algebra in Coq. In *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, volume 7406 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2012. doi:10.1007/978-3-642-32347-8_7.

24 Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast, embeddable, λProlog interpreter. In *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 460–468. Springer, 2015. doi:10.1007/978-3-662-48899-7_32.

25 François Garillot. *Generic Proof Tools and Finite Group Theory. (Outils génériques de preuve et théorie des groupes finis)*. PhD thesis, École Polytechnique, Palaiseau, France, 2011. URL: https://tel.archives-ouvertes.fr/pastel-00649586.

26 François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In *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 327–342. Springer, 2009. doi:10.1007/978-3-642-03359-9_23.

27 Paolo G. Giarrusso. ring tactic breaks on mathcomp: it requires all ring operations are defined on *syntactically* equal types, not definitionally equal ones, 2020. Accessed: 2021-12-07. URL: https://github.com/coq/coq/issues/11998.

28 Georges Gonthier and Enrico Tassi. A language of patterns for subterm selection. In *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, volume 7406 of *Lecture Notes in Computer Science*, pages 361–376. Springer, 2012. doi:10.1007/978-3-642-32347-8_25.

29 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. doi:10.1017/S0956796813000051.

30 Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *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.

31 Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In *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.

32 Jason Gross, Andres Erbsen, and Adam Chlipala. Reification by parametricity - fast setup for proof by reflection, in two lines of Ltac. In *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 289–305. Springer, 2018. doi:10.1007/978-3-319-94821-8_17.

33 Benjamin S. Hvass. ring tactic for math-comp integers, 2019. Accessed: 2021-12-07. URL: https://github.com/math-comp/math-comp/issues/401.

34 Daniel W. H. James and Ralf Hinze. A reflection-based proof tactic for lattices in Coq. In *Proceedings of the Tenth Symposium on Trends in Functional Programming, TFP 2009, Komárno, Slovakia, June 2-4, 2009*, volume 10 of *Trends in Functional Programming*, pages 97–112. Intellect, 2009.

35 Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. Mtac2: typed tactics for backward reasoning in Coq. *Proceedings of the ACM on Programming Languages*, 2(ICFP):78:1–78:31, 2018. doi:10.1145/3236773.

36 Robert Y. Lewis and Paul-Nicolas Madelaine. Simplifying casts and coercions (extended abstract). In *Joint Proceedings of the 7th Workshop on Practical Aspects of Automated Reasoning (PAAR) and the 5th Satisfiability Checking and Symbolic Computation Workshop (SC-Square) Workshop, 2020 co-located with the 10th International Joint Conference on Automated Reasoning (IJCAR 2020), Paris, France, June-July, 2020 (Virtual)*, volume 2752 of *CEUR Workshop Proceedings*, pages 53–62, 2020. URL: http://ceur-ws.org/Vol-2752/paper4.pdf.

37 Assia Mahboubi and Thomas Sibut-Pinote. A formal proof of the irrationality of $\zeta(3)$. *Logical Methods in Computer Science*, 17(1), 2021. doi:10.23638/LMCS-17(1:16)2021.

38 Assia Mahboubi and Enrico Tassi. Canonical structures for the working Coq user. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2013. doi:10.1007/978-3-642-39634-2_5.

39 Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, 2021. doi:10.5281/zenodo.4457887.

40 Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012. doi:10.1017/CBO9781139021326.

41 Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. doi:10.1007/978-3-030-79876-5_37.

42 Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015. doi:10.1007/978-3-319-21401-6_26.

43 Pierre-Marie Pédrot. Ltac2: Tactical warfare. In *The Fifth International Workshop on Coq for Programming Languages, CoqPL 2019*, 2019. URL: https://popl19.sigplan.org/details/CoqPL-2019/8/Ltac2-Tactical-Warfare.

44 Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 199–208. ACM, 1988. doi:10.1145/53990.54010.

45 Claudio Sacerdoti Coen and Enrico Tassi. ELPI - embeddable λProlog interpreter. Accessed: 2022-02-08. URL: https://github.com/LPCIC/elpi.

46 Amokrane Saïbi. Typing algorithm in type theory with inheritance. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 292–301. ACM Press, 1997. doi:10.1145/263699.263742.

47 Amokrane Saïbi. *Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types. Application à la Théorie des Catégories. (Formalization of Mathematics in Type Theory. Generic tools of Modelisation and Demonstration. Application to Category Theory)*. PhD thesis, Pierre and Marie Curie University, Paris, France, 1999. URL: https://tel.archives-ouvertes.fr/tel-00523810.

48 Kazuhiko Sakaguchi. Validating mathematical structures. In *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 138–157. Springer, 2020. doi:10.1007/978-3-030-51054-1_8.

49 Kazuhiko Sakaguchi. Algebra tactics, 2021. Accessed: 2022-02-08. URL: https://github.com/math-comp/algebra-tactics.

50 Kazuhiko Sakaguchi. Mczify, 2021. Accessed: 2022-02-08. URL: https://github.com/math-comp/mczify.

51 Matthieu Sozeau and Nicolas Oury. First-class type classes. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2008. doi:10.1007/978-3-540-71067-7_23.

52 Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21(4):795–825, 2011. doi:10.1017/S0960129511000119.

53 Enrico Tassi. Coq-Elpi. Accessed: 2022-02-08. URL: https://github.com/LPCIC/coq-elpi.

54 The Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2022. the PDF version with numbered sections is available at https://doi.org/10.5281/zenodo.1003420. URL: https://coq.inria.fr/distrib/V8.15.0/refman/.

55 The Coq Development Team. Section 2.2.4 in [54]: "Syntax extensions and notation scopes", 2022. URL: https://coq.inria.fr/distrib/V8.15.0/refman/user-extensions/syntax-extensions.

56 The Coq Development Team. Section 2.2.6 in [54]: "Implicit coercions", 2022. URL: https://coq.inria.fr/distrib/V8.15.0/refman/addendum/implicit-coercions.

57 The Coq Development Team. Section 2.2.8 in [54]: "Canonical structures", 2022. URL: https://coq.inria.fr/distrib/V8.15.0/refman/language/extensions/canonical.

58 The Coq Development Team. Section 3.1.5 in [54]: "The ssreflect proof language", 2022. URL: https://coq.inria.fr/distrib/V8.15.0/refman/proof-engine/ssreflect-proof-language.

59 The Coq Development Team. Section 3.2.2 in [54]: "Micromega: solvers for arithmetic goals over ordered rings", 2022. URL: https://coq.inria.fr/distrib/V8.15.0/refman/addendum/micromega.

60 The Coq Development Team. Section 3.2.3 in [54]: "Ring and field: solvers for polynomial and rational equations", 2022. URL: https://coq.inria.fr/distrib/V8.15.0/refman/addendum/ring.

61 The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 367–381. ACM, 2020. doi:10.1145/3372885.3373824.

62 Alfred van der Poorten. A proof that Euler missed: Apéry's proof of the irrationality of $\zeta(3)$. *Math. Intelligencer*, 1(4):195–203, 1979. An informal report. doi:10.1007/BF03028234.

63 Beta Ziliani and Matthieu Sozeau. A comprehensible guide to a new unifier for CIC including universe polymorphism and overloading. *Journal of Functional Programming*, 27:e10, 2017. doi:10.1017/S0956796817000028.

Formalizing Algorithmic Bounds in the Query Model in EasyCrypt

Alley Stoughton
Boston University, MA, USA

Carol Chen
Stuyvesant High School, New York, NY, USA

Marco Gaboardi
Boston University, MA, USA

Weihao Qu
Boston University, MA, USA

Abstract

We use the EasyCrypt proof assistant to formalize the adversarial approach to proving lower bounds for computational problems in the query model. This is done using a lower bound game between an algorithm and adversary, in which the adversary answers the algorithm's queries in a way that makes the algorithm issue at least the desired number of queries. A complementary upper bound game is used for proving upper bounds of algorithms; here the adversary incrementally and adaptively realizes an algorithm's input. We prove a natural connection between the lower and upper bound games, and apply our framework to three computational problems, including searching in an ordered list and comparison-based sorting, giving evidence for the generality of our notion of algorithm and the usefulness of our framework.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Models of computation; Theory of computation → Design and analysis of algorithms

Keywords and phrases query model, lower bound, upper bound, adversary argument, EasyCrypt

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.30

Supplementary Material EasyCrypt Frameworks for Proving Algorithmic Bounds:
Software (Source Code): `https://github.com/alleystoughton/AlgorithmicBounds`

Funding *Alley Stoughton*: National Science Foundation Grant No. 1801564.
Marco Gaboardi: National Science Foundation Grant No. 2040249.

Acknowledgements We would like to thank Mark Bun for numerous helpful discussions. The anonymous referees provided very helpful feedback that helped us improve the paper.

1 Introduction

1.1 Algorithmic Background

The establishment of lower bounds for computational problems, and upper bounds for algorithms solving those problems, are synergistic activities. As algorithms and their upper bounds become better, that encourages the search for tighter lower bounds, and vice versa. One of the richest sources of non-trivial lower bounds is the query model,[1] in which an algorithm doesn't have direct access to its input, but may only issue queries about it. A lower bound theorem in the query model shows a lower bound, as a function of the input size, n, of the number of queries any algorithm must issue in the worst case, in order to solve a size-n instance of the given computational problem.

[1] We include comparison-based sorting in this category.

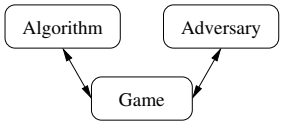


Figure 1 Game between Algorithm and Adversary.

Adversary arguments (the adversarial method) are one of the most common ways of establishing such lower bounds [3, 2, 13]. When Knuth first published Volume 3 of *The Art of Computer Programming* [15], he used the term "oracular lower bounds" for this method. In Section 5.3.2, when discussing minimum comparison merging, he talked of constructing a "suitable oracle" that answers an algorithm's queries about which of two elements is greater:

> If we can construct a suitable oracle ... we can ensure that every valid merging algorithm will have to ask a rather large number of questions.

In the 1998 Second Edition of Volume 3 [16], he has adopted the modern terminology of "adversary" instead of oracle.

An adversary argument is structured as a two-party game between the algorithm and adversary, as illustrated in Figure 1. The algorithm is allowed to query parts of the input (e.g., bits, or cells), and the adversary is programmed to answer in a way that delays the game long enough to achieve the desired lower bound. The duration of the game is measured by steps, where one step consists of a query by the algorithm combined with its answer by the adversary. The game[2] plays the role of referee, keeping track of the inputs that are consistent with the adversary's answers so far, and declaring the game to be over when all the remaining inputs have the same answer. Neither the algorithm nor the adversary have to report anything. When proving a lower bound theorem, one exhibits an adversary, and then proves that, for all algorithms (correct or not), the game continues for long enough to achieve the lower bound – in which case we say the adversary has "won" the game.

Perhaps the simplest example of an adversary argument is the proof that an algorithm computing the or (disjunction) function of a list of booleans of size n must query, in the worst case, the value of every index of the list, and so must make n queries. The adversary can be stateless, and when asked for the value of the ith element of the list it can always answer false, keeping the algorithm uncertain as to whether the result of the or function will be true or false. Initially the set of input lists maintained by the game consists of all 2^n lists of booleans of size n. As the game progresses, the set of input lists will always consist of all lists of booleans in which all the elements at the indices queried so far are false. Thus as long as not every index has been queried, there is at least one input list including at least one occurrence of true, and so where the answer is true, and exactly one input list where all the elements are false, and thus the answer is false. In other words, the game is not over until all indices have been queried.

In our work, we are interested in establishing concrete bounds (like n in the preceding example), rather than asymptotic ones. Of course concrete bounds can by abstracted to asymptotic bounds after the initial proof, as desired.

[2] We consider a slight variation of the framework of [2] that is well-suited to formalization.

1.2 Formalization in EasyCrypt

When we set out to formalize adversary arguments using a proof assistant, we opted to work in EasyCrypt because of our experience (e.g., [21]) formalizing cryptographic games in EasyCrypt. EasyCrypt (see Section 2) has a module system allowing one to implement algorithms and adversaries as modules – collections of procedures operating on private variables. This allows our *lower bound game* (see Section 3.3) to be expressed as a parameterized module – parameterized by an abstract algorithm and adversary. Procedures in EasyCrypt are allowed to employ random assignments, choosing values from probability distributions. Consequently, lower bound theorems in our formalization must be proved against potentially probabilistic algorithms. Making use of randomness doesn't give an algorithm an advantage, though, if we consider worst-case efficiency and expect it to produce the correct result with probability 1. An adversary "wins" a run of the lower bound game against an algorithm and for a bound lb iff the game runs at least lb steps before the queries/answers uniquely determine the computational problem's answer and the game ends. When proving a lower bound theorem, one exhibits an adversary and shows that, for all algorithms, the adversary wins the lower bound game with probability 1. The lower bound game itself isn't probabilistic, and neither are the adversaries we have used in our work to date. Thus when an algorithm is also non-probabilistic, winning with probability 1 just means the only possible run results in a win.

In our framework, we have a general method (see Section 3.1) for expressing computational problems in the query model over inputs consisting of lists of a fixed size, n. Our bounds are then expressions in terms of n. We accommodate restrictions on input lists, e.g., that they are sorted. Our notion of query is general enough to encompass comparisons, i.e., queries asking not for the *value* of the input at a given index, but asking how the values at two indices are *related*. A computational problem is parameterized by an auxiliary value picked by the adversary, but made available to the algorithm. E.g., this is used in searching problems to say which element should be searched for in an ordered list.

Upper bound theorems can also be naturally expressed in terms of a game: an *upper bound game*, parameterized by an algorithm and an adversary. This time (see Section 3.4), the algorithm reports an answer to the computational problem, in addition to issuing queries.[3] The adversary adaptively answers the algorithm's queries, incrementally realizing more and more of the input list (or the relative order of the input list's elements, in the case of comparison queries). This notion of adversary includes ones based on hard-coded auxiliary values and input lists. The upper bound game plays the role of referee, keeping track of the inputs that are consistent with the answers to the queries issued so far. An algorithm "wins" a run of the upper bound game for a bound ub and against an adversary iff, in no more than ub steps, either the algorithm reports the correct answer (the same, for all remaining consistent inputs), or the adversary answers a query inconsistently, causing the game to end early. When proving an upper bound theorem for an algorithm, we prove that for all (even probabilistic) adversaries, the algorithm wins the upper bound game against the adversary with probability 1. The upper bound game itself isn't probabilistic, and so when both the algorithm and adversary are also non-probabilistic, winning with probability 1 just means the only possible run results in a win.

We connect the lower and upper bound games via the following theorem (see Section 3.5 for the formal statement): If an adversary wins the lower bound game against an algorithm for bound lb with probability 1, and $ub < lb$, then with probability 1, the algorithm loses the upper bound game against the adversary for bound ub.

[3] Such an algorithm can be converted to the kind expected by the lower bound game.

We use examples to informally make the case that our model of algorithm – in which a game asks the algorithm for its next query, and later tells it the answer – is general enough to model all algorithms. In a recursive algorithm, e.g., the stack of recursive calls may be arbitrarily deep at the point where an answer is needed to a query. We must suspend the computation, storing the suspension in a global variable of the algorithm. Once the answer is received, this suspended computation can then be resumed and supplied the query's answer. We realize this using terms of an ad hoc functional language as the suspensions. We leave to future work the design of, and development of a meta-theory for, a general purpose functional language for expressing suspendable algorithms in the query model.

In our current work, the probabilistic nature of EasyCrypt is something of an impediment. Because we are modeling the worst-case query complexity of algorithms that must always produce correct results, randomization is not an advantage when expressing algorithms. Fortunately, we have a generic method for reducing lower and upper bound theorems for probability 1 to non-probabilistic Hoare logic judgments. This method still requires us to prove termination with probability 1 of the procedures of our adversary (resp., algorithm) in a lower bound (resp., upper bound) proof. But doing this is not significantly different from proving termination, as we don't use randomness in our adversaries (resp., algorithms). In future work, though, we would like to work with randomized algorithms that are allowed to produce incorrect results with small probability. Thus continuing to work in EasyCrypt may be an advantage.

By using our EasyCrypt framework for lower and upper bound results, we eliminate errors that could lurk in ad hoc formalizations. The lower and upper bound games act as referees, and EasyCrypt guarantees that algorithms and adversaries cannot interfere with the states of the games or their opponents. Assuming we are happy with the formalization of a computational problem, the definition of an adversary is simply part of the proof of a lower bound theorem; if it has "bugs", as long as the proof goes through, they are irrelevant. Furthermore, when proving an upper bound theorem for a given algorithm, the framework ensures that we are counting queries correctly. And any algorithm, even a "buggy one", shows the existence of an algorithm with the proved upper bound.

1.3 Our Contributions

Here are the paper's contributions, all of which were formalized using the EasyCrypt proof assistant:

- We formalize a general notion of list-based computational problems in the query model.
- We give a generic formalization of the adversarial method, expressing this as a two-party lower bound game between an algorithm and adversary.
- We formalize a two-party upper bound game for establishing upper bounds of algorithms, where the adversary adaptively and incrementally realizes an input to the algorithm.
- We prove a natural connection between the lower and upper bound games.
- We provide an axiom-free EasyCrypt theory for working with bounds involving logarithms. We also provide an axiom-free theory formalizing well-founded relations, induction and recursion.
- We demonstrate the utility and generality of our framework by applying it to three computational problems, giving evidence for why our apparently "passive" notion of algorithm is general enough to express all algorithms, including recursive ones.

1.4 Paper Outline

The rest of the paper is structured as follows. In Section 2, we give an introduction to EasyCrypt and detail the proof practices we follow. In Section 3: we describe our framework for formalizing computational problems, as well as the lower and upper bound games; we consider the proof of the theorem connecting these games; and we consider the proof of the lower bound theorem for the or function. In Section 4, we apply our framework to the problem of searching in an ordered list, proving identical lower and upper bounds (the upper bound is for the binary search algorithm). We also (Section 4.1) consider a general EasyCrypt theory for working with bounds involving logarithms. In Section 5, we apply our framework to the problem of sorting a list of distinct elements, proving lower and upper bounds (the upper bound is for the merge sort algorithm) that are close, but with a slight gap. We also mention our EasyCrypt theory of well-founded relations, induction and recursion. In Section 6, we consider related work. Finally, in Section 7 we consider directions for future work.

2 The EasyCrypt Proof Assistant

EasyCrypt [6] is a tactic-based proof assistant with several program logics allowing one to state and prove lemmas about a simple programming language, pWhile, with non-recursive procedures, while loops, and probabilistic assignments, and in which programs are structured using a simple module system designed for expressing games. Modules consist of procedures along with global variables that can be manipulated by those procedures. They can be parameterized by abstract modules implementing module types, and can later be applied to concrete modules implementing those types. EasyCrypt has:

- a Hoare logic for proving partial correctness of procedures;
- a probabilistic Hoare logic, pHL, for proving probabilistic facts about procedures;
- a probabilistic relational Hoare logic, pRHL, for proving relations between procedures;
- an "ambient" higher-order logic, which is used for connecting judgments of the program logics, as well as for giving mathematical definitions and proving lemmas about them.

Simple ambient logic goals may be solved using SMT solvers. Finally, EasyCrypt has theories, which collect definitions, modules and module types, axioms and lemmas. Theories may be parameterized by types and operators, and instantiating a theory with values for those types and operators is done by a process called cloning. All axioms involving those types and operators must then be proved to hold.[4] E.g., if a theory T includes the parameters

```
op n : int. (* n is an integer *)
axiom gt0_n : 0 < n. (* n is positive *)
```

and we clone T instantiating n with the expression k * 2, then we must prove that k * 2 is positive.

To explain the meaning of judgments in EasyCrypt's logics, suppose M.f (M is a module) is a procedure with parameter x : int returning a value of type bool, and N.g is a procedure with parameter y : int returning a value of type bool.

The Hoare logic judgment

```
hoare [M.f : 0 < x ⟹ res]
```

[4] EasyCrypt provides a facility for cloning but leaving some types and operators abstract, so axioms involving them are not proved, but we do not use this facility in our work.

says that if we start M.f in a memory &m in which the value of the parameter field x is positive, and this execution terminates with a memory &n, then the result value field res of &n will be true (this says nothing when the execution does not terminate). If, instead, we wrote

```
hoare [M.f : 0 < x ==> !res]
```

then we would be saying that the result value field will be false, not true (! is negation).

The executions of M.f from &m can be modeled as the branches of a tree that splits upon random assignments, where each arc is labeled with the (non-zero) probability of the selected value from the support of the distribution. EasyCrypt allows sub-distributions, and so the sum of these probabilities may be less than 1. In addition, each iteration of a while loop is marked by a single arc labeled with probability 1. These trees can be infinitely splitting, even though EasyCrypt's distributions are discrete (a distribution on an infinite type will be non-uniform), and they can have infinite branches, corresponding to non-termination. Each branch has an associated probability (the limit of the multiplication of all its probabilities), and the sum of the probabilities of all the branches may be less than 1. A *run* is a branch with non-zero probability. When M.f is non-probabilistic, its tree has a single run, which either terminates in a final memory or is infinite.

The pHL judgment

```
phoare [M.f : 0 < x ==> res] = p
```

says that if we execute M.f from an initial memory &m in which the value of x is positive, the probability that the execution terminates in a memory with a result value field that is true is p – i.e., p is the sum of the probabilities of all the branches of the running of M.f from &m terminating with memories whose result value fields are true. If we prove this, we can then use some ambient logic tactics to conclude

```
forall &m (a : int), 0 < a ⇒ Pr[M.f(a) @ &m : res] = p
```

which says that for all memories &m and positive integers a, the probability that running M.f in the updating of &m that gives x the value a terminates in a memory whose result value field is true is exactly p. If we set p to 1%r (the integer 1 converted to type real) this means every run of M.f(a) from &m terminates in a memory whose result value field is true, and the sum of the probabilities of those runs is 1. And if M.f isn't probabilistic, its single run will terminate in a memory with a true result value field. This allows us to express total correctness. A judgment

```
phoare [M.f : true ==> true] = 1%r
```

says that M.f is *lossless*: no matter what memory M.f is started in (the memory includes its argument), it terminates with probability 1. This mean that any infinite branch (if there are any) has probability 0, i.e., all runs terminate, and the sum of the runs' probabilities is 1. If M.f is non-probabilistic, losslessness means that the single branch of M.f from any initial memory terminates.

The pRHL judgment

```
equiv [M.f ~ N.g : x{1} = y{2} ==> res{1} ⇒ res{2}]
```

says that if &1 and &2 are memories in which the value of x in &1 is equal to the value of y in &2, then the distributions on memories corresponding to running M.f and N.g on &1 and &2, respectively, satisfy the lifting of the formula res{1} ⇒ res{2} (if the result of the first memory is true, then the result of the second memory is true) to pairs of memory distributions. Given a proof of this judgment, we can then prove the ambient logic formula

Listing 1 Framework Parameters.

```
type inp, out, aux. op univ : inp list. op arity : {int | 0 ≤ arity} as ge0_arity.
axiom univ_uniq : uniq univ.
op good : aux → inp list → bool. op f : aux → inp list → out option.
axiom good (aux : aux, inps : inp list) :
  size inps = arity ⇒ all (mem univ) inps ⇒ good aux inps ⇒ exists (y : out), f aux inps = Some y.
axiom bad (aux : aux, inps : inp list) :
  size inps ≠ arity ∨ ! (all (mem univ) inps) ∨ ! good aux inps ⇒ f aux inps = None.
```

```
forall &m (a : int), Pr[M.f(a) @ &m : res] ≤ Pr [N.g(a) @ &m : res]
```

Alternatively, if we prove the judgment

```
equiv [M.f ~ N.g : x{1} = y{2} ⟹ res{1} = res{2}]
```

we can conclude

```
forall &m (a : int), Pr[M.f(a) @ &m : res] = Pr [N.g(a) @ &m : res]
```

2.1 Our Proof Practices

Unrestricted use of SMT solvers can lead to poorly documented, unstable proofs. We counter this as follows. Sometimes we avoid using SMT solvers entirely, using the EasyCrypt directive that disallows the smt tactic. More commonly, we use the directive requiring that every goal solved using the smt tactic be solved by *both* the Z3 and Alt-Ergo solvers. And in every use of smt, we explicitly list the lemmas the solvers may use (in addition to their internal theories). smt() means to only use the solvers' internal theories.

3 EasyCrypt Framework for Lower and Upper Bounds

In this section we describe our framework – defined as a theory Bounds in EasyCrypt (788 lines of code) – for describing computational problems and proving lower and upper bound theorems, using lower and upper bound games. We also consider the proof of a theorem connecting the lower and upper bound games. And we consider the proof of the lower bound theorem for the or function that was sketched in the introduction.

3.1 Expressing Computational Problems

The Bounds theory is parameterized by several types and operators (functions) that allow us to express computational problems. They are described in Listing 1. An *input list* is a list of values of size arity (a non-negative integer; ge0_arity is introduced as the name of an axiom saying that 0 ≤ arity) where each value has type inp (the *input type*) and is an element of a finite universe univ (a list of distinct (unique) elements). The type aux is a type of *auxiliary values*, and a value of type aux is the first parameter of the good and f operators. When inps is an input list, good aux tests whether it is a *good* input list for our computational problem, *according to* the auxiliary value aux. The operator f represents our *computational problem*; f aux inps returns an optional value of the *output type*, out. We axiomatize that:

- (good) If inps is an input list that is good according to aux, then f aux inps returns Some of some output value.
- (bad) Otherwise, f aux inps returns None.

When we instantiate the above types and operators, we will typically leave arity *abstract*, so that algorithms and adversaries will have to work for all input list sizes.

Below are three examples of how computational problems can be formalized using our framework. We believe many more problems can be easily encoded using variations of these techniques.

3.1.1 Or Function

If our computational problem is the or (disjunction) function, we can instantiate inp, univ, out and aux to bool, [true; false], bool and unit (whose only value is ()), where good () always returns true (all input lists satisfy it). When inps is an input list, f inps returns Some true, if at least one of the elements of inps is true, and Some false, otherwise. We use this instantiation of Bounds in Section 3.6.

3.1.2 Searching in an Ordered List

To see why we parameterize f and good by an auxiliary value, consider the problem of searching in an ordered list inps of elements from a finite range of at least two integers for a given value k that is guaranteed to occur at least once in inps, returning the first index (indices are in the range $0, \ldots,$ arity $- 1$) into inps where k is found. Here we can instantiate inp and out to int, and instantiate univ to the finite range of integers. But both good and f need to know what k is, and so we set aux to int, so the first arguments to good and f can be k. Then an input list inps is satisfied by good k iff it is sorted in (not necessarily strictly) ascending order and contains at least one occurrence of k, and f k inps returns Some of the first index into inps where k is found, when inps is a good input list relative to k, and returns None, otherwise. We use this instantiation of Bounds in Section 4.

3.1.3 Sorting

We can encode the problem of sorting a list of distinct elements of size len (at least 1) according to some total ordering, as follows. First, we instantiate inp and univ to bool and [true; false], and instantiate arity to len $*$ len. Thus an index into an input list inps can be thought of as encoding a pair (i, j), where i and j are indices into the list of distinct elements, and asking for the boolean corresponding to (i, j) can be thought of as querying whether the ith element of the list of distinct elements is less-than-or-equal to the jth element. In this example, we don't need an auxiliary value, and so aux can be unit. The predicate good, though, should test whether an inps satisfies the properties of a total ordering. Finally, the operator f should transform a list of booleans representing a total ordering to Some of the permutation of the indices of the list of distinct elements that is sorted according to the total ordering, and return None when not given a total ordering. Thus out can be int list (even though the len! permutations on the indices $0, \ldots,$ len $- 1$ are a small subset of this type). We use this instantiation of Bounds in Section 5, when our lower and upper bounds will be expressed in terms of len (not arity).

3.2 Adversaries

In the following two sections, we will consider two sub-theories of Bounds: LB for lower bounds, and UB, for upper bounds. Both of these theories share the same kind of *adversary*, defined by the following module type (interface):

```
module type ADV = { proc *init() : aux   proc ans_query(i : int) : inp }.
```

Listing 2 Lower Bound Game.

```
1  module G(Alg : ALG, Adv : ADV) = {
2    proc main() : bool * int = {
3      var inpss : inp list list; var don : bool; var error : bool;
4      var stage : int; var queries : int fset; var aux : aux;
5      var i : int; var inp : inp;
6      aux <@ Adv.init(); Alg.init(aux);
7      inpss ← init_inpss aux; error ← false; don ← inpss_done aux inpss; stage ← 0; queries ← fset0;
8      while (!don ∧ !error) {
9        i <@ Alg.make_query();
10       if (0 ≤ i < arity ∧ ! i \in queries) {
11         queries ← queries `|` fset1 i; stage ← stage + 1;
12         inp <@ Adv.ans_query(i);
13         Alg.query_result(inp);
14         inpss ← filter_nth inpss i inp; don ← inpss_done aux inpss;
15       }
16       else { error ← true; }
17     }
18     return (error, stage);
19   }
20 }.
```

A module of this type may have global (private) variables, and implements at least the two procedures of the module type. There may be auxiliary procedures, and all procedures may access the global variables. The procedure init takes no arguments and returns an auxiliary value – because in both the lower and upper bound games we let the adversary choose the auxiliary value. The asterisk before its name specifies that init must initialize the module's global variables. The procedure ans_query will only be called with an index i into an input list, i.e., an integer between 0 and arity − 1. It is a request for the value of the ith element of the input list. In addition to returning this value, a stateful adversary will update some of its global variables, so it knows how to properly respond to subsequent queries.

3.3 Lower Bounds Subtheory

In this subsection, we consider the lower bounds subtheory, LB. A *lower bound algorithm* is a module satisfying the interface:

```
module type ALG = { proc *init(aux : aux) : unit   proc make_query() : int
  proc query_result(x : inp) : unit }.
```

Its init procedure is given the auxiliary value chosen by the adversary, and initializes its global variables. The procedure make_query asks the algorithm to issue a query, which should be an index into an input list, i.e., an integer in the range $0, \ldots,$ arity $- 1$. And the procedure query_result is called to tell the algorithm the adversary's answer to the algorithm's last query. Note that the algorithm does not report an answer to the computational problem.

The *lower bound game*, G, is defined in Listing 2. G is parameterized by an algorithm Alg and adversary Adv, i.e., modules with the module types ALG and ADV, respectively. It defines a procedure main that takes no arguments, and returns a value of type bool * int. The boolean indicates whether the game ended with the algorithm committing an *error* by issuing an illegal query, and the integer is the *stage* (number of steps executed) at the game's termination. main uses two groups of local variables (which are not accessible to Alg and Adv). The ones on lines 3–4 are persistent; they are initialized before the game's while loop, and are updated by each loop iteration. The ones on line 5 are temporary variables; their values are reset at each iteration. main begins (line 6) by asking Adv to initialize itself and choose the auxiliary value, which is stored in aux, and then asking Alg to initialize itself, letting it know the auxiliary value. On line 7, main initializes the rest of the persistent variables:

- inpss records (as a list of lists with no duplicates) the good inputs lists (relative to aux) that are consistent with the queries issued so far and their answers. It is initialized to all the good input lists, using an operator init_inpss.
- error records whether the algorithm has issued an illegal query.
- don records whether all elements of inpss have the same answer to the computational problem (relative to aux), i.e., whether all the elements of the result of mapping f aux over inpss are the same (this is what the operator inpss_done tests). Depending upon good, f and aux, this may be true initially; and it's true whenever inpss is empty.
- stage records the current stage, and is initialized to 0.
- queries records the set of queries that have been issued by the algorithm, and is initialized to the empty set.

The while loop of main runs as long as both error and don are false, i.e., as long as the algorithm has not committed an error and at least two elements of inpss have different answers. An iteration of the while loop begins by asking (line 9) Alg to issue a query, which is stored in i. If (test on line 10) i is not a good index into an input list or repeats an earlier query, this causes error to be set to true. Otherwise (line 11) the set of queries is updated to include i and the stage is incremented. Adv (line 12) is then asked to answer the query i, producing inp. On line 13, Alg is informed of the adversary's answer. On line 14, inpss is filtered to only retain input lists whose ith elements are equal to inp, and then don is recomputed based on the updated inpss, seeing if the game is now done. Once the while loop has terminated (line 18), because either error or don became true, main returns the final error status and stage.

If Adv answers a query in a way that is inconsistent with inpss, i.e., so that after filtering, inpss becomes empty, then don will be set to true, and so the game will end without error. It doesn't matter whether Alg notices this inconsistency upon the final call to Alg.query_result. But at every call to Alg.make_query, Alg can be assured that all of the answers it has received so far are consistent.

When proving a lower bound theorem for a given computational problem, we program a concrete adversary Adv whose procedures are provably lossless, and then prove that for all algorithms Alg whose procedures are lossless and don't read or write the global variables of Adv (or vice versa),

```
Pr[G(Alg, Adv).main() @ &m : res.`1 ∨ ϕ(arity) ≤ res.`2] = 1%r.
```

holds, where ϕ is the desired function of arity. We can read the conclusion as saying that, when started in a memory &m (which the game doesn't depend on, because G has no global variables, and Adv.init and Alg.init initialize the adversary's and algorithm's global variables), with probability 1, the game terminates either with the algorithm having committed an error (by asking an out of range query or the same query twice), or with the final stage being at least ϕ(arity).

An algorithm that issues duplicate queries can be converted into a more efficient one by caching query answers, and so from the point of view of lower bounds, treating query-repeating algorithms as erroneous is sound. Our motivation for signaling an error when a query is repeated is to ensure the game terminates – which is technically useful.

When proving that the procedures of an algorithm (or adversary) are lossless, it is sometimes necessary to condition this on a *termination invariant* on the global variables of the algorithm (or adversary). When doing this, we show that the initialization procedures establish this invariant with probability 1, and that the other procedures preserve the invariant with probability 1. We will see this in action when we consider the proof of the upper bound theorem for the merge-sort algorithm (Section 5.2).

Because our conclusion is with probability 1 and we assume the procedures of the algorithm are lossless, we can apply a generic lemma that we have proved using pHL to show that G(Alg, Adv).main is lossless. And we can use this fact to reduce our theorem to a "main lemma" whose conclusion is an ordinary Hoare (partial correctness) judgment:

```
hoare [G(Alg, Adv).main : true ==> res.`1 \/ φ(arity) <= res.`2].
```

The challenging part of proving the main lemma is handling the game's loop. In Hoare logic, this is done using a loop invariant, and we need a loop invariant that is true when the loop is first entered, is preserved by the loop, and where the conjunction of the loop invariant and the fact that the game has ended – and so either the algorithm has committed an error, or f aux agrees on all elements of inpss – tells us that if the algorithm hasn't committed an error, then the game has run long enough to give us the desired lower bound. In Sections 3.6, 4.2 and 5.1, we will see three rather different examples of loop invariants supporting lower bound proofs.

3.4 Upper Bounds Subtheory

In this subsection, we consider the upper bounds subtheory, UB. In order to define the module type of upper bound algorithms, we need a datatype response of algorithm responses, along with some associated operators:

```
type response = [ Response_Query of int | Response_Report of out ].
op dec_response_query (resp : response) : int option =
  with resp = Response_Query i ⇒ Some i  with resp = Response_Report _ ⇒ None.
op dec_response_report (resp : response) : out option =
  with resp = Response_Query _ ⇒ None  with resp = Response_Report x ⇒ Some x.
op is_response_query (resp : response) : bool = dec_response_query resp ≠ None.
op is_response_report (resp : response) : bool = dec_response_report resp ≠ None.
```

A value of type response either has the form Response_Query i, for an integer i, representing a query by the algorithm, or the form Response_Report x, for a value x of type out, representing the algorithm's reporting of an answer, x, to a computational problem. The operator dec_response_query tries to decode a value of type response as a query, yielding Some i when given Response_Query i, and yielding None when given a value with constructor Response_Report. And dec_response_report is similar, but with the constructors swapped, and so producing an optional value of type out. The operators is_response_query and is_response_report use these operators to test whether a value of type response has constructor Response_Query or Response_Report.

An *upper bound algorithm* is a module satisfying the interface:

```
module type ALG = { proc *init(aux : aux) : unit   proc make_query_or_report_output() : response
  proc query_result(x : inp) : unit }.
```

The procedures init and query_result are just like the procedures of the same names of a lower bound algorithm. And make_query_or_report_output asks the algorithm to either issue another query or report the answer to the computational problem.

The *upper bound game*, G, is defined in Listing 3. This game is similar to the lower bound game (Listing 2), and so we focus on the differences. The interpretations of error and don are different:

- error can still become true because Alg issues an illegal query, but also because the final answer it reports is incorrect.
- don can become true because Alg reports the correct answer, but also because Adv chooses aux or answers queries in a way that causes inpss to become empty.

Listing 3 Upper Bound Game.

```
1  module G(Alg : ALG, Adv : ADV) = {
2    proc main() : bool * int = {
3      var inpss : inp list list; var aux : aux; var error : bool;
4      var don : bool; var stage : int; var queries : int fset;
5      var resp : response; var i : int; var inp : inp; var out : out;
6      aux <@ Adv.init(); Alg.init(aux);
7      inpss ← init_inpss aux; error ← false; don ← inpss = []; stage ← 0; queries ← fset0;
8      while (!don ∧ !error) {
9        resp <@ Alg.make_query_or_report_output();
10       if (is_response_query resp) {
11         i ← oget (dec_response_query resp);
12         if (0 ≤ i < arity ∧ ! i \in queries) {
13           queries ← queries `|` fset1 i; stage ← stage + 1;
14           inp <@ Adv.ans_query(i);
15           Alg.query_result(inp);
16           inpss ← filter_nth inpss i inp; if (inpss = []) { don ← true; }
17         }
18         else { error ← true; }
19       }
20       else {
21         out ← oget (dec_response_report resp);
22         if (inpss_answer aux inpss out) { don ← true; }
23         else { error ← true; }
24       }
25     }
26     return (error, stage);
27   }
28 }.
```

On line 7, don is initialized to be true iff inpss is empty. On line 9, Alg is asked to either issue a query or report the final answer, encoded in a value resp of type response. If resp encodes a query, then it is decoded on line 11, resulting in the query i. Lines 12–15 and line 18 are the same as lines 10–13 and line 16 of the lower bound game. But line 16 is different: it sets don to true if inpss has become empty. Alternatively, if resp encodes the reporting of an output, then line 21 decodes resp to out. Line 22 then checks (via the operator inpss_answer) whether out is the answer (relative to aux) for all the elements of inpss, i.e., whether every element of the result of mapping f aux over inpss is equal to Some out. If the answer is "yes", then don is set to true; otherwise error is set to true.

When proving an upper bound theorem for algorithm Alg, we prove that Alg's procedures are lossless, and that, for all adversaries Adv whose procedures are lossless and don't read or write the global variables of Alg, that

```
Pr[G(Alg, Adv).main() @ &m : ! res.`1 ∧ res.`2 ≤ φ(arity)] = 1%r.
```

holds, where ϕ is the desired function of arity. We can read the conclusion as saying that, when started in a memory &m, with probability 1, the game terminates without the algorithm having committed an error (and so with Alg having reported the correct answer to the computational problem, unless Adv caused inpss to become empty) and with the final stage being no more than ϕ(arity).

Because our conclusion is with probability 1 and we assume the procedures of the adversary are lossless, we can apply a generic lemma that we have proved using pHL to show that G(Alg, Adv).main is lossless. And we can use this fact to reduce our theorem to a "main lemma" whose conclusion is an ordinary Hoare judgment:

```
hoare [G(Alg, Adv).main : true ⟹ ! res.`1 ∧ res.`2 ≤ φ(arity)].
```

Listing 4 Conversion from Upper to Lower Bound Algorithm.

```
module UBAlg_to_LBAlg (UBAlg : UB.ALG) : LB.ALG = {
  proc init(aux : aux) : unit = { UBAlg.init(aux); }
  proc make_query() : int = {
    var resp : UB.response; var i : int; resp <@ UBAlg.make_query_or_report_output();
    if (UB.is_response_query resp) { i ← oget (UB.dec_response_query resp); } else { i ← −1; }
    return i;
  }
  proc query_result(x : inp) : unit = { UBAlg.query_result(x); }
}.
```

In Sections 4.3 and 5.2 we will see two rather different examples of loop invariants supporting upper bound proofs.

If Adv answers a query inconsistently, so that after filtering, inpss becomes empty, and thus don is set to true and the game ends without error, the final call to Alg.query_result must still update the state of Alg in such a way that the loop invariant is preserved. But at every call to Alg.make_query_or_report_output, Alg can be assured that all of the answers it has received so far are consistent.

A meta-level analysis using the semantics of EasyCrypt shows that if Alg loses a run of the upper bound game against Adv and for bound ub (the only run if both Alg and Adv are non-probabilistic), there is an auxiliary value aux, and an inps that is good relative to aux such that there is a run (the only run if Alg is non-probabilistic) of Alg against the non-probabilistic, non-adaptive adversary that picks aux and then answers queries according to inps in which Alg loses. Thus if we are able to prove that Alg wins the upper bound game against all such *hard-coded* adversaries with probability 1, it will actually win the game against all adversaries with probability 1. We haven't tried to prove this meta result in EasyCrypt, and we don't assume it in our proofs.

3.5 Connections between Frameworks

Listing 4 defines a parameterized module UBAlg_to_LBAlg that converts an upper bound algorithm to a lower bound one. The only issue is what to do when the upper bound algorithm reports an answer to the computational problem – an action that isn't allowed for a lower bound algorithm. UBAlg_to_LBAlg translates the reporting of an answer into the illegal query −1. (This is so that, if the upper bound algorithm reports an answer before there is a unique answer according to the remaining consistent input lists, and so the upper bound game terminates with an error, the same thing will happen in the lower bound game.)

Through a sequence of lemmas using pRHL, Hoare logic, pHL and the ambient logic, we were able to prove the theorem saying that for all integers lb and ub, upper bound algorithms Alg whose procedures are lossless, adversaries Adv whose procedures are lossless and don't read/write the global variables of Alg, and memories &m:

```
Pr[LB.G(UBAlg_to_LBAlg(Alg), Adv).main() @ &m : res.`1 ∨ lb ≤ res.`2] = 1%r ⇒ ub < lb ⇒
Pr[UB.G(Alg, Adv).main() @ &m : res.`1 ∨ ub < res.`2] = 1%r.
```

Thus if we have proved a lower bound theorem involving Adv and lb, and we know ub < lb, we can conclude that every run of the upper bound game between Alg and Adv (when both Alg and Adv are non-probabilistic there is only one run) ends with either Alg committing an error or the game having run strictly more than ub steps, and that the sum of the probabilities of those runs is 1. This of course implies that we cannot prove an upper bound theorem for Alg with bound ub.

3.6 Lower Bound for Or Function

As a warm up exercise, we formalized the proof from Section 1.1 that any algorithm computing the or function on a list of booleans of size arity must query every element of the list, in the worst case. We clone our general Bounds framework as described in Section 3.1.1. Our adversary is stateless and answers all queries with false. Our loop invariant for the lower bound game's while loop simply says (in addition to some housekeeping properties) that inpss is all the input lists in which all the queried indices are false. The formalization of the computational problem and the subsequent lower bound proof took 324 of code.

4 Application to Searching in an Ordered List

In this section, we consider proofs of lower and upper bound theorems for the computational problem of searching in an ordered list of integers (coming from a finite range of at least two elements) of size arity in which an element k occurs at least once, returning the first index into the list where k can be found. We clone our general Bounds theory as described in Section 3.1.2.

4.1 Theory for Reasoning about Bounds Involving Logarithms

We have developed a reusable (also used in Section 5) EasyCrypt theory IntLog (1440 lines of code) for reasoning about bounds involving integer logarithms. We have the operators

```
op (%/) : int → int → int. op (%%/) : int → int → int.
```

%/ is EasyCrypt's integer division operator, and we define %%/ to add one to the result of n %/ b when n is not divisible by b. Then we can prove that, for all integers n and $b \geq 1$, n %/ $b = \lfloor n/b \rfloor$[5] and n %%/ $b = \lceil n/b \rceil$. We define integer logarithm operators rounding down and up

```
op int_log : int → int → int. op int_log_up : int → int → int.
```

Then we prove that, for all $b \geq 2$ and $n \geq 1$, int_log b $n = \lfloor \log b\ n \rfloor$ and int_log_up b $n = \lceil \log b\ n \rceil$, where log is the real number logarithm operator. For use in lower and upper bound proofs, it is convenient to define operators

```
op divpow2 (n k : int) : int = n %/ (2 ^ k). op divpow2up (n k : int) : int = n %%/ (2 ^ k).
```

where ^ is exponentiation. We will mention some of the many lemmas we have proved about these operators in the following sections.

4.2 Lower Bound Proof

Our adversary is expressed in terms of the minimum element of the universe, a, and a + 1, which we call b. Its init procedure chooses b as the auxiliary value – the value the algorithm must search for. The adversary has global variables win_beg and win_end of type int and win_empty of type bool that determine the current *window of uncertainty*, where win_beg and win_end are indices into an input list such that win_beg ≤ win_end, and if win_empty holds (the window is empty), then win_beg = win_end and win_end < arity − 1 (the window does not end at the end of the input list). The *window size* (computed by operator win_size)

[5] In EasyCrypt's syntax, one actually must write floor (n%r / b%r).

is defined to be: 0, if the window is empty; and win_end − win_beg + 1, otherwise. The procedure init initializes win_beg to 0, win_end to arity − 1 and win_empty to false. When ans_query is called with query i, it acts as follows:

- if the window is empty (win_size returns 0), it returns witness – some unknown but fixed value[6];
- else, if i is strictly smaller than win_beg, it returns a;
- else, if i is strictly greater than win_end, it returns b;
- else, if the window has size 1 and win_end is arity − 1, it returns b;
- else, if the window has size 1 and win_end is strictly less than arity − 1, it sets win_empty to true and returns witness;
- else, if $i <$ (win_beg + win_end) %%/ 2, it sets win_beg to $i + 1$ and returns a;
- else it sets win_end to $i - 1$ and returns b.

The part of the game's loop invariant relating to inpss says that, if the window is not empty, then:

- for all i between win_beg and win_end, inclusive, the input list consisting of a's up to but not including position i, and then b's thereafter, is in inpss; and
- if win_end < arity − 1, the input list consisting of a's up to and including position win_end, and then b's thereafter, is in inpss.

The bound part of the loop invariant says that:

```
(win_end = arity − 1 ⇒
 divpow2up arity stage ≤ win_size win_empty win_beg win_end) ∧
(win_end < arity − 1 ⇒
 divpow2 arity stage ≤ win_size win_empty win_beg win_end)
```

We are able to prove that:

- if the window is not empty and the game is finished, then the window size is 1; and
- if win_end < arity − 1 and the game is finished, then the window is empty.

At the game's beginning, win_end = arity − 1. If this is still true at the game's end, it follows that the window is not empty and divpow2up arity stage is 1, and thus – by one of the lemmas of IntLog – that int_log_up 2 arity ≤ stage. Otherwise, there is a point at which win_end is set to one less than the algorithm's query, i. At this point, we *don't* know that the new window size is at least the old window size divided by two, rounding *up*, but it *is* at least the old window size divided by two, rounding *down*. And thus we switch to only knowing divpow2 arity stage ≤ win_size win_empty win_beg win_end. Then, when the game ends, we have that the window is empty, so that divpow2 arity stage is 0. Because it is 0, we can conclude – using another of our IntLog lemmas – that int_log_up 2 arity ≤ stage. Consequently, our lower bound theorem has bound int_log_up 2 arity.

4.3 Upper Bound Proof

Because binary search can be expressed iteratively, it was straightforward to define the binary search algorithm in our framework. Its state consists of the element aux to be searched for, along with input list indices low and high, where low ≤ high and the following invariant holds:

- every element of inpss is sorted and has at least one occurrence of aux between positions low and high inclusive;
- in every element of inpss, there are no occurrences of aux at positions strictly less than low;

[6] Used to document that the proof doesn't depend on the value returned.

- no indices between low and high inclusive have been asked as queries.

low and high are initialized to 0 and arity − 1. The *window size* is defined to be high − low + 1. When make_query_or_report_output is called:

- if the window size is 1, the algorithm reports its answer, low;
- else, the algorithm queries the midpoint mid ← (low + high) %/ 2.

When query_result is called with the answer, x:

- if $x <$ aux, it sets low to mid + 1;
- else, it sets high to mid.

In either case, the new window size is no more than the old window size divided by 2, rounding *up*. Thus the bound part of the loop invariant can be

```
stage ≤ int_log_up 2 arity ∧ win_size low high ≤ divpow2up arity stage
```

When the window size is 2 or more, one of our IntLog lemmas tells us that stage < int_log_up 2 arity, ensuring the bound invariant will be preserved as stage is incremented. Our overall upper bound is thus int_log_up 2 arity.

4.4 Conclusions

The formalization of the searching problem and some associated lemmas took 148 lines of code. And the proof of the lower bound (resp., upper bound) theorem took 724 (resp., 353) lines of code.

Our upper and lower bounds are identical, proving that the binary search algorithm is optimal. Our first version of the lower bound theorem used a simpler approach but only achieved a bound of int_log 2 arity. We noticed this was not optimal for the arity of 3, and then wrote an OCaml program[7] to generate optimal strategies for larger arities and small universes. The program's results helped us develop an adversarial strategy supporting our tighter lower bound theorem.

5 Application to Sorting

In this section, we consider proofs of lower and upper bound theorems for the computational problem of sorting a nonempty list of distinct elements of size len. We clone our general Bounds theory as described in Section 3.1.3.

5.1 Lower Bound Proof

Our adversary has a single global variable inpss of type inp list list consisting of its own copy of the list of consistent input lists maintained by the lower bound game. Initially, this is init_inpss (), which has len! elements – because the set of all permutations of the indices $0, \ldots,$ len $- 1$ is in one-to-one correspondence with all the total orderings on those indices. When its ans_query procedure is called with a query q encoding a comparison query (i, j) (and so asking if element i of the list of distinct elements is less-than-or-equal-to element j), it partitions inpss into two lists:

- inpss_t is all the elements of inpss in which position q is true; and
- inpss_f is all the elements of inpss in which position q is false.

[7] All OCaml programs are included in our repository.

If the size of inpss_f is at least as big as the size of inpss_t, it sets inpss to inpss_f and returns false; otherwise it sets inpss to inpss_t, and returns true. Because the new size of inpss in each step is at least the old size divided by 2, rounded *up*, the initial size of inpss is fact len, and the game isn't over until inpss contains a single element, it is straightforward to prove a lower bound of int_log_up 2 (fact len). We lower-approximate this in two ways: (1) (len ∗ int_log 2 len) %/ 2; and (2) len ∗ (int_log 2 len) − 2 ∗ 2^(int_log 2 len). When len is at least 11, we prove (2) ≥ (1). For example, when len is 20,000, (2) evaluates to 247,232 whereas int_log_up 2 (fact len) evaluates to 256,909, and so the gap is only 9,677 comparisons.

5.2 Upper Bound Proof

Our upper bound result is for the merge sort algorithm. First, we defined a recurrence wc returning (what we later prove is) an upper-approximation of the worst-case number of comparisons needed by merge sort when sorting a nonempty list of distinct elements of size n:

$$\mathsf{wc}\, n = \begin{cases} 0, & \text{if } n = 1, \\ \mathsf{wc}\,(n\,\%/\,2) + \mathsf{wc}\,(n\,\%\%/\,2) + n - 1, & \text{otherwise.} \end{cases}$$

If we had made a mistake in defining wc, our upper bound proof would have failed. We were able to upper-approximate wc len as len ∗ int_log 2 len.

In order to define wc in EasyCrypt and prove the necessary properties about it, we used our axiom-free theory WF (503 of code) for well-founded relations, induction and recursion.[8] In this theory, we mirrored the set theoretic formalization of these concepts within EasyCrypt's higher-order logic, using the type

```
type 'a rel = 'a → 'a → bool.
```

to represent relations. This theory has now been added to the official EasyCrypt library.

Because merge sort is a recursive algorithm, we opted to let Alg have a single global variable term of type term where term is the inductive datatype

```
type term = [ Sort of int list | List of int list | Cons of int & term | Merge of term & term
  | Cond of int & int & int list & int list ].
```

We think of the elements of term as *terms* of an ad hoc functional programming language. A term is evaluated relative to a total ordering on indices. Sort xs evaluates to the result of sorting xs. List xs evaluates to xs. Cons i t is evaluated by evaluating t to xs, and then returning $i :: xs$. Merge t u is evaluated by first evaluating t to xs, next evaluating u to ys, and then returning the result of merging xs and ys. Finally, Cond i j us vs queries whether $i \leq j$ in the total ordering, returning: $i :: zs$, where zs is the result of merging us and $j :: vs$, when the answer is "yes"; and $j :: zs$, where zs is the result of merging $i :: us$ and vs, when the answer is "no".

The procedure init initializes term to Sort $[0; \ldots; \mathsf{len} - 1]$. We implemented a single-step operational semantics that runs a term until it either produces an answer (List xs) which can be reported to the algorithm (as the result of make_query_or_report_output) or asks a query (Cond), which can be returned to the game, recording the blocked term in the global variable term. Once the answer to the query is supplied via query_result, the algorithm applies the answer to term, so execution can continue upon the next call to make_query_or_report_output. To ensure termination of make_query_or_report_output, we make use of a well-formedness invariant on terms together with a termination metric.

[8] We developed this theory for another purpose, but it has not been referenced in the literature, to-date.

Our loop invariant says that:

- term is well-formed;
- for each remaining consistent input list inps (representing a total ordering), the evaluation of term relative to inps is the same as the result of sorting $[0; \ldots; \mathsf{len} - 1]$ using inps;
- wc_term term + stage $\leq$ wc len, where wc_term recursively upper-bounds the numbers of comparisons term can make in the worst case; and
- all the comparisons term could potentially make have not yet been issued as queries.

At the start of the game, wc_term term is wc len and stage is 0. As the game progresses, wc_term term becomes smaller as stage becomes bigger. When the game ends, wc_term term is 0, and so we get that stage is no more than wc len, which we know is no more than len $*$ int_log 2 len.

5.3 Conclusions

The formalization of the sorting problem took 557 lines of code (this included the formalization of total orderings as lists of booleans). The lower bound proof took 547 lines of code, which included 59 lines of generic code for showing that two lists of unique elements have the same size using the existence of a bijection between them. And the upper bound proof took 1740 lines of code, much of which consisted of meta-theoretic results about the ad hoc functional language used to model suspendable, recursive computations.

Our lower and upper bounds are close, but there is a small gap. E.g., if we compare int_log_up 2 (fact len) (the tightest form of the lower bound) with wc len (the tightest form of the upper bound) when len is equal to 20,000, we obtain 256,909 and 267,233, respectively, for a gap of only 10,324. We are considering how the gap could be closed.

6 Related Work

Formalization of lower and upper bounds. There is existing work [14, 20, 11, 1, 17] formalizing specific lower bounds using proof assistants in computational models other than the query model. Some of these works include reusable libraries applicable to certain classes of problems. General frameworks for certifying upper bounds were developed in [12, 18], but again not in the query model. Eberl [11] and Azevedo de Amorim [1] have formalized lower bounds on comparison-based sorting problems in Isabelle and Coq, respectively. Both works utilize decision trees, even though the two recursive datatypes representing them are rather different. Their proofs give a lower bound, in terms of the number of permutations on list indices, on the heights of decision trees, and then lower-approximate this bound in terms of list size. In our work we formalize a similar lower bound as an instance of our framework, showing that lower bound games and adversarial reasoning are good and flexible abstractions. In fact, the decision tree model and the query model are closely related, and we expect that a formalization of the adversarial approach could also be based on decision trees. We leave the study of the relations between the two formalizations to future work. In contrast to the formalizations of Eberl and Azevedo de Amorim, our work is much more general. Our generic notions of algorithms and adversaries, and generic definitions of lower and upper bound games apply uniformly to all computational problems. Any algorithm can give rise to an upper bound theorem, assuming the proof goes through; similarly, any adversary supports a lower bound theorem, assuming the proof goes through. We are not aware of other works that have formalized the query model directly. Recent work by Tassarotti et al. [22] formalizes probably approximately correct (PAC) learnability in Lean. Classifiers, in the PAC learning model, are similar to queries in the query model. It would be interesting to explore if our framework could be used in this model.

Game-based formal reasoning. Extensive research has employed formal reasoning based on games, often in the context of cryptographic security. Novak [19] developed a framework for proving the security of cryptographic schemes based on probabilistic games in Coq. Barthe et al. [7] formalized a framework in Coq with similar goals where cryptographic schemes and protocols are modeled by probabilistic games, parameterized by adversarial entities. Subsequent work in this direction led to the design of EasyCrypt [8, 6], which provides better support for this kind of reasoning. We use EasyCrypt in our work, demonstrating its utility for game-based reasoning beyond security. An Isabelle/HOL framework for proving the security of cryptographic schemes based on probabilistic games has recently been developed [9]. Recent work [5] has extended EasyCrypt with a cost model, allowing proofs about execution time and numbers of oracle calls and restrictions on the cost of adversaries. It would be interesting to investigate if this extension could be used to reason about lower bounds in the query model in a more natural way. Game-based reasoning supported by a proof assistant has also been used for certifying results useful in computer networking [4]. The recent work of [10] has employed a two-player game between an adversary and an algorithm to certify lower bounds for the online bin stretching problem. The proposed game, used to establish lower bounds for varying numbers of bins, is formalized in Coq. Interestingly, this work also proves some previously unknown lower bounds. As a comparison, our framework targets a larger class of computational problems in the query model. Moreover, it is not limited to lower bounds, but also explores upper bounds and the connection between them.

7 Future Work

In future work, we would like to generalize our framework for defining computational problems in various ways. It would be nice to be able to directly model problems whose inputs are not lists, e.g., graphs, instead of having to encode them using lists. We would also like to be able to model problems where answers are not required to be unique, e.g., searching for any element of a list satisfying some predicate.

The formalization of the ad hoc functional language we used to model suspendable, recursive query-model algorithms in our upper bound proof for merge sort (Section 5.2) could be generalized so as to be applicable to recursive algorithms in general, instead of just merge sort. Proving the meta-theoretic results once and for all in a more general form would save a great deal of effort when formalizing upper bound theorems for other algorithms.

We would like to extend our framework to be able to handle randomized algorithms that are allowed to produce incorrect results with small probabilities. In such an extended framework, the probabilistic nature of EasyCrypt could be an asset.

Finally, we would like to use our existing framework or future versions of our framework to discover new lower or upper bounds, as opposed to simply formalizing variations of standard results in our framework(s), as we have done in the current work.

References

1 Arthur Azevedo De Amorim. Poleiro: Analyzing Sorting Algorithms, 2015. URL: `http://poleiro.info/posts/2015-03-25-analyzing-sorting-algorithms.html`.

2 Sepehr Assadi. Advanced Algorithms II: Sublinear Algorithms: Lecture 3, 2020. URL: `https://people.cs.rutgers.edu/~sa1497/courses/cs514-s20/lec3.pdf`.

3 Sara Baase and Allen Van Gelder. *Computer Algorithms: Introduction to Design and Analysis (third edition)*. Pearson/Prentice Hall, 2000.

4 Alexander Bagnall, Samuel Merten, and Gordon Stewart. A library for algorithmic game theory in Ssreflect/Coq. *J. Formaliz. Reason.*, 10(1):67–95, 2017.

5 Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, and Pierre-Yves Strub. Mechanized proofs of adversarial complexity and application to universal composability. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15–19, 2021*, pages 2541–2563. ACM, 2021.

6 Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. EasyCrypt: A tutorial. In *Foundations of Security Analysis and Design VII*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer International Publishing, 2014.

7 Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21–23, 2009*, pages 90–101. ACM, 2009.

8 Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Proceedings of the 31st Annual Conference on Advances in Cryptology*, CRYPTO 2011, pages 71–90. Springer-Verlag, 2011.

9 David A. Basin, Andreas Lochbihler, and S. Reza Sefidgar. CryptHOL: Game-based proofs in higher-order logic. *J. Cryptol.*, 33(2):494–566, 2020.

10 Martin Böhm and Bertrand Simon. Discovering and certifying lower bounds for the online bin stretching problem. *CoRR*, abs/2001.01125, 2020. `arXiv:2001.01125`.

11 Manuel Eberl. Lower bound on comparison-based sorting algorithms. *Arch. Formal Proofs*, 2017, 2017. URL: `https://www.isa-afp.org/entries/Comparison_Sort_Lower_Bound.shtml`.

12 Manuel Eberl. Proving divide and conquer complexities in Isabelle/HOL. *J. Autom. Reason.*, 58(4):483–508, 2017.

13 Jeff Erickson. Lecture Notes: Adversary Arguments, 2013. URL: `https://jeffe.cs.illinois.edu/teaching/algorithms/notes/13-adversary.pdf`.

14 Ruben Gamboa and John Cowles. A mechanical proof of the Cook-Levin theorem. In *International Conference on Theorem Proving in Higher Order Logics*, pages 99–116. Springer-Verlag, 2004.

15 Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

16 Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching, 2nd Edition*. Addison-Wesley, 1998.

17 Laura Kovács, Hanna Lachnitt, and Stefan Szeider. Formalizing graph trail properties in Isabelle/HOL. In Christoph Benzmüller and Bruce R. Miller, editors, *Intelligent Computer Mathematics: 13th International Conference, CICM 2020, Bertinoro, Italy, July 26–31, 2020, Proceedings*, volume 12236 of *Lecture Notes in Computer Science*, pages 190–205. Springer-Verlag, 2020.

18 Shiri Morshtein, Ran Ettinger, and Shmuel S. Tyszberowicz. Verifying time complexity of binary search using Dafny. In José Proença and Andrei Paskevich, editors, *Proceedings of the 6th Workshop on Formal Integrated Development Environment, F-IDE@NFM 2021, Held online, 24–25th May 2021*, volume 338 of *EPTCS*, pages 68–81, 2021.

19 David Nowak. A framework for game-based security proofs. In Sihan Qing, Hideki Imai, and Guilin Wang, editors, *Information and Communications Security, 9th International Conference, ICICS 2007, Zhengzhou, China, December 12–15, 2007*, volume 4861 of *Lecture Notes in Computer Science*, pages 319–333. Springer-Verlag, 2007.

20 Ulrich Schöpp. A formalised lower bound on undirected graph reachability. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008, Doha, Qatar, November 22–27, 2008. Proceedings*, volume 5330 of *Lecture Notes in Computer Science*, pages 621–635. Springer-Verlag, 2008.

21 Alley Stoughton and Mayank Varia. Mechanizing the proof of adaptive, information-theoretic security of cryptographic protocols in the random oracle model. In *30th IEEE Computer Security Foundations Symposium*, pages 83–99, Santa Barbara, CA, USA, 2017. IEEE Computer Society. URL: `https://github.com/alleystoughton/PCR`.

22 Joseph Tassarotti, Koundinya Vajjha, Anindya Banerjee, and Jean-Baptiste Tristan. A formal proof of PAC learnability for decision stumps. In Catalin Hritcu and Andrei Popescu, editors, *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17–19, 2021*, pages 5–17. ACM, 2021.

Formalization of a Stochastic Approximation Theorem

Koundinya Vajjha[1]
University of Pittsburgh, PA, US

Barry Trager
IBM Research, Yorktown Heights, NY, US

Avraham Shinnar
IBM Research, Yorktown Heights, NY, US

Vasily Pestun
IBM Research, Yorktown Heights, NY, US
Institut des Hautes Études Scientifiques, Bures sur Yvette, France

Abstract

Stochastic approximation algorithms are iterative procedures which are used to approximate a target value in an environment where the target is unknown and direct observations are corrupted by noise. These algorithms are useful, for instance, for root-finding and function minimization when the target function or model is not directly known. Originally introduced in a 1951 paper by Robbins and Monro, the field of Stochastic approximation has grown enormously and has come to influence application domains from adaptive signal processing to artificial intelligence. As an example, the Stochastic Gradient Descent algorithm which is ubiquitous in various subdomains of Machine Learning is based on stochastic approximation theory. In this paper, we give a formal proof (in the Coq proof assistant) of a general convergence theorem due to Aryeh Dvoretzky [21] (proven in 1956) which implies the convergence of important classical methods such as the Robbins-Monro and the Kiefer-Wolfowitz algorithms. In the process, we build a comprehensive Coq library of measure-theoretic probability theory and stochastic processes.

2012 ACM Subject Classification Mathematics of computing → Stochastic processes; Mathematics of computing → Nonlinear equations; Computing methodologies → Optimization algorithms

Keywords and phrases Formal Verification, Stochastic Approximation, Stochastic Processes, Probability Theory, Optimization Algorithms

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.31

Supplementary Material *Software*: `https://github.com/IBM/FormalML/releases/tag/ITP2022`

Funding *Koundinya Vajjha*: Vajjha acknowledges support from the Alfred P. Sloan Foundation under grant number G-2018-10067 and the Andrew W. Mellon Foundation.

1 Introduction

This paper presents a formal proof of Aryeh Dvoretzky's 1956 result on stochastic approximation.

To motivate this result, let us consider a problem frequently occurring in various contexts of statistical learning: Let Y be a real-valued random variable that depends on a parameter x. We may say that $P(Y|x)$ is the probability distribution of Y conditioned or dependent on a parameter x. Next, suppose we are given a function $f(y, x)$ and we want to find x that solves the equation

$$\mathbb{E}_P f(Y, x) = 0 \tag{1}$$

[1] Corresponding author.

Moreover, assume that $P(Y|x)$ is not available to us explicitly, but only in an implicit or sampling form, that is, we are provided a sampling oracle which takes a parameter x and returns a sample of Y drawn from x-dependent probability distribution $P(Y|x)$.

▶ **Example 1** (Kolmogorov's Strong Law of Large Numbers)**.** Let $f(y, x) = y - x$, and let Y be independent of x. To solve equation (1) in this situation means to solve $\mathbb{E}[Y] = x$, that is to find the expected value x of a random variable Y given an oracle from which we can sample Y, in other words to construct a statistical estimator of $\mathbb{E}[Y]$ given a series of samples $y_0, y_1, \ldots$. The following iterative algorithm does the job

$$x_{n+1} := x_n + a_n(y_n - x_n) \tag{2}$$

for $n = 0, 1, 2, \ldots$ where $x_0 := 0$ and $a_n = \frac{1}{n+1}$. Indeed, the iterations (2) are equivalent to the standard sample mean estimator $x_n = \frac{1}{n}\sum_{k=0}^{n-1} y_k$. Notice that the iterations (2) have the form $x_{n+1} = x_n + a_n f(y_n, x_n)$. The theorem that the estimator x_n converges almost surely (with probability one) to the true expectation value is famously known as the "Kolmogorov's Strong Law of Large Numbers" (SLLN).

▶ **Example 2** (Banach's fixed point and optimal control)**.** Now consider the opposite example, where Y that depends on x in a deterministic way, say $Y = g(x)$ where $g : \mathbb{R} \to \mathbb{R}$ is a certain function (and event space is a single point). In this case, when we pass x to the oracle, the oracle deterministically returns to us the value of the function g evaluated at x. In this case, solving the equation (1) for the function $f(y, x) = y - x = g(x) - x$ means solving the equation

$$g(x) = x \tag{3}$$

If g is a γ-contraction map[2] the standard proof of the Banach fixed point theorem tells us that the iterations

$$x_{n+1} := x_n + a_n(g(x_n) - x_n) \tag{4}$$

for a suitable choice of a_n, for example $a_n = \frac{1}{n+1}$, form a Cauchy sequence $x_1, x_2, \ldots$ that converges to the fixed point of the map $g : \mathbb{R} \to \mathbb{R}$. A variation of this process is applied to solve Bellman's equation for optimal control of Markov Decision Process (MDP) where γ-contraction map g comes from Bellman's optimality operator for MDPs with discount parameter $0 < \gamma < 1$.

▶ **Example 3** (Stochastic gradient descent)**.** Now, as a variation of (1), suppose that we want to find x that minimizes the expectation value $\mathbb{E}[L(Y, x)]$ of a certain loss function $L(Y, x)$ in a context where Y is sampled by an oracle from an x-dependent probability distribution. Assuming that $\mathbb{E}[L(Y, x)]$ is a locally convex analytic function, finding a local minimum is equivalent to solving the stationary point equation

$$\nabla_x \mathbb{E}[L(Y, x)] = 0 \tag{5}$$

Since ∇_x is a linear operator, the above equation is equivalent to $\mathbb{E}[\nabla_x L(Y, x)] = 0$ and therefore is again an example of the equation (1) with $f(y, x) := -\nabla_x L(y, x)$. The iterative sequence

$$x_{n+1} := x_n - a_n \nabla_x L(y_n, x_n) \tag{6}$$

---

[2] this means that in some norm $||\bullet||$ it holds that $||g(x) - g(x')|| < \gamma||x - x'||$ for all x, x' with $\gamma < 1$

is known as stochastic gradient descent. This algorithm is a typical component of most of machine learning algorithms that search for a parameter x that minimizes the expected value of the loss function $L(Y, x)$ given samples of Y.[3] Under suitable conditions on $f(y, x) = -\nabla_x L(y, x)$ and the parameters a_n (for example, $a_n = \frac{1}{n+1}$ would satisfy all required assumptions) one can prove convergence of (6) to the critical point of the loss function $L(Y, x)$.

These three examples demonstrate the ubiquity of the problem (1), and many more applications could be mentioned in a longer report.

In all these cases the solution of the problem (1) has the form

$$x_{n+1} := x_n + a_n f(y_n, x_n) \tag{7}$$

and is called *a stochastic approximation algorithm.*

A large body of literature explored different versions of assumptions on the domain of variables, on the function $f(y, x)$ and on the step-sizes (learning rates) a_n, under which the convergence of x_n could be proven in various senses: as convergence in L^2, as convergence in probability, as convergence with probability 1.[4]

Robbins and Monro introduced in [30] the field of Stochastic Approximation by proving the L^2 convergence of the process (7) for $f(y, x) = b - y$ to the value x that solves the equation $\mathbb{E}[Y](x) = b$. Note that we write $\mathbb{E}[Y](x)$ to indicate that x occurs as a parameter in the distribution of Y. For this theorem, Robbins and Monro assumed

$$a_n \to 0, \qquad \sum_{n=1}^{\infty} a_n = \infty, \qquad \sum_{n=1}^{\infty} a_n^2 < \infty, \tag{8}$$

that Y is bounded with probability 1, and that the function $M(x) := \mathbb{E}[Y](x)$ is (i) non-decreasing, (ii) the solution x_* of $M(x) = b$ exists, and (iii) the derivative at the solution is positive $M'(x)|_{x=x_*} > 0$.

Kiefer and Wolfowitz [26] took a similar approach but considered the problem of estimating the parameter x where the function $M(x)$ has a maximum, and proved convergence in probability.

Wolfowitz [42] weakened the assumption of Robbins-Monro about boundedness of Y: instead his version assumes only that the variance of Y is bounded uniformly over x, and $M(x)$ is bounded, and with those assumptions Wolfowitz proves convergence in probability.

Blum [12] weakened further the assumptions of Robbins-Monro and Wolfowitz and proved a substantially stronger result, namely that the iterative sequence (7) (with $f(x_n, y_n) = b - y_n$) converges with probability 1. Blum requires the variance of Y be uniformly bounded over x, but he allows the expectation value $M(x) = \mathbb{E}[Y](x)$ to be bounded by a linear function of x

$$|M(x)| \leq A|x| + B \quad A, B \geq 0 \tag{9}$$

instead of a constant. Blum's proof is based on a version of Kolmogorov's inequality adopted in a suitable way by Loève [28] where instead of series of independent random variables, a certain dependence was allowed but constrained by a conditional expectation value. This

[3] In the context of supervised learning, y will stand for (y_{in}, y_{out}) tuples sampled from training data, and x stands for the model parameters, e.g. neural network weights. If $N_x : y_{in} \mapsto y_{out}$ is a neural network, then with a quadratic supervised loss one normally takes $L(y, x) := (y_{out} - N_x(y_{in}))^2$ where $y = (y_{in}, y_{out})$

[4] The notion of "convergence with probability 1" is the same as the notion of "convergence almost surely", but different from the notion of "convergence in probability", which is much weaker.

extension of Kolomogorov's inequality to the conditional situation was related to earlier works of Borel, Lévy and Doob about convergence with probability 1 of certain stochastic processes.

Finally, the most general form of stochastic approximation was formulated by Dvoretzky [21]. In the original Robbins-Monro stochastic approximation (7), the next value x_{n+1} is determined through the previous value x_n and the sample y_n. Dvoretzky allowed more general estimator algorithms in which x_{n+1} is determined through a certain function that can take as arguments complete history of all previous values $x_1, \ldots, x_n$ and the current sample y_n.

Concretely, let $T_n : \mathbb{R}^n \to \mathbb{R}$ be a real-valued function of n-variables. Consider the stochastic process

$$x_{n+1} := T_n(x_1, \ldots, x_n) + W_n \tag{10}$$

where $W_1, W_2, \ldots$ are random variables, with W_n dependent on the previous history $X_1, \ldots, X_n$ such that

$$\mathbb{E}(W_n |\ x_1, \ldots, x_n) = 0 \tag{11}$$

Another way to formulate Dvoretzky's setup is to say that for any sequence of random variables $X_1, X_2, \ldots$ where we have conditional probability distribution of X_{n+1} dependent on the complete history $x_1, \ldots, x_n$, and then *define*

$$\begin{aligned} T(x_1, \ldots, x_n) &\stackrel{def}{=} \mathbb{E}[X_{n+1} | x_1, \ldots, x_n] \\ W_n &\stackrel{def}{=} X_{n+1} - \mathbb{E}[X_{n+1} | x_1, \ldots, x_n] \end{aligned} \tag{12}$$

in this way we automatically get the relation (10) with noise terms W_n that satisfy (11).

For example, in the Robbins-Monro version we take (7) with $f(y, x) = b - y$ which gives $X_{n+1} := X_n + a_n(b - Y_n)$ and hence the Robbins-Monro process is a specialization of Dvoretzky's process with

$$\begin{aligned} &T(x_1, \ldots, x_n) := x_n + a_n(b - M(x_n)) \\ &W_n := a_n(M(x_n) - Y_n) \end{aligned} \tag{13}$$

whereas before in the context of Robbins-Monro we had $M(x_n) := \mathbb{E}(Y_n | x_n)$.

To prove his result, Dvoretzky assumed that:

1. there exists a point x_* such that

$$|T_n(x_1, \ldots, x_n) - x_*| \leq \max(\alpha_n, (1 + \beta_n)|x_n - x_*| - \gamma_n) \tag{14}$$

where $\alpha_n, \beta_n, \gamma_n$ are sequences of non-negative real numbers with

$$\alpha_n \to 0 \tag{15}$$

$$\sum_n \beta_n < \infty \tag{16}$$

$$\sum_n \gamma_n = \infty \tag{17}$$

2. The cumulative variance of the noise terms W_n is bounded

$$\sum_{n=1}^{\infty} \mathbb{E}[W_n^2] < \infty, \qquad \mathbb{E}[W_n | X_1, \ldots, X_n] = 0 \tag{18}$$

and proved that the iterative sequence (10) converges with probability 1 to the fixed point x_*.

The Robbins-Monro theorem in its strongest form (that is, under the weakest assumptions of Blum (9)) becomes an easy consequence of Dvoretzky theorem. We only have to check that given the assumptions of Blum we can apply Dvoretzky. First, Blum's assumption that the variance of Y_n is bounded by, say, σ^2 for all x and n, given the relation (13), implies $\sum_{n=1}^{\infty} \mathbb{E}[W_n^2] = \sum_{n=1}^{\infty} a_n^2 \sigma^2 < \infty$, and therefore Dvoretzky's assumption (18) about limited cumulative variance of his noise terms holds. Second, given Blum's $M(x)$ in equation (9), we will construct $\alpha_n, \beta_n, \gamma_n$ that satisfy (15) and such that the bound on the operator T in (14) holds. To do that, first choose a real-valued series $\{\rho_n\}$ with $\rho_n > 0$ and $\rho_n \to 0$ such that[5] $\sum_n \rho_n a_n = \infty$. For simplicity assume, by a change of coordinates, that the fixed point $x_* = 0$. Assuming that $M(x)$ is regular and monotonic in a neighborhood of x_* there is an inverse map M^{-1} and then we define the sequence $\{\eta_n\} = \{M^{-1}(\rho_n)\}$ for sufficiently small ρ_n. Next, define (for sufficiently large n)

$$\begin{aligned} \alpha_n &:= \max(\eta_n, B a_n) \\ \beta_n &:= 0 \\ \gamma_n &:= a_n \rho_n \end{aligned} \tag{19}$$

A case-by-case check for $|x| \leq \eta_n$ and for $|x| > \eta_n$ shows that Dvoretzky's bound on (14) holds given the relation (13).

One universal theme passing through the various versions of stochastic approximation convergence theorems is the choice of the scheduling of the step-sizes (or learning rates) a_n.

In the Robbins-Monro scheduling assumption (8), it is clear that the step-sizes have to converge to zero (otherwise the model would fluctuate and never converge to the exact solution). The second assumption $\sum_{n=1}^{\infty} a_n = \infty$ that says that the rates should not converge to zero too fast is also sensible, as otherwise it is easy to imagine a learning schedule with a_n dropping to zero so fast that the iterative process does not reach the fixed point x_* from an initial point x_0 (for a concrete example, see [17, pp. 5]). The third assumption, $\sum_{n=1}^{\infty} a_n^2 < \infty$, is more subtle and technical, it primarily ensures that even in situations when the noise-terms have self-correlation they would not move the iterative process out of its track of converging with probability 1 to the exact fixed point. In certain situations, a slower decreasing of learning rate schedule still leads to convergence, and is faster in practice.

As the above discussion has shown, the Robbins-Monro paper spawned a huge literature on the analysis and applications of such stochastic algorithms. This is because the problem of estimating unknown parameters of a model from observed data is quite a fundamental one, with variants of this problem appearing in one form or another in control theory, learning theory and other fields of engineering.

Because of the pervasive reach of stochastic approximation methods, any serious formalization effort of an algorithm involving parameter estimation when the underlying model is unknown will eventually have to contend with formalizing tricky stochastic convergence proofs. We chose to formalize Dvoretzky's theorem as it implies the convergence of both the Robbins-Monro and Kiefer-Wolfowitz algorithms, various stochastic gradient descent agorithms and various reinforcement learning algorithms such as Q-learning based on Bellman's optimality operator.

[5] for example, if we start with $a_n = \frac{1}{n+1}$ take $\rho_n = \frac{1}{\log(n+1)}$, in general take $\rho_n^{-1} = \sum_{k=1}^{n} a_n$, (see [1]).

▶ Remark. Throughout the text which follows, hyperlinks to theorems, definitions and lemmas which have formal equivalents in the Coq development are indicated by a ✿. Our formalization is open-source and is available at `https://github.com/IBM/FormalML`.

2 Dvoretzky's Theorem

After Dvoretzky's original publication [21] of his theorem and several very useful extensions, several shorter proofs have been proposed. A simplified proof was published by Wolfowitz [43] who like Blum relied on the conditional version of Kolmogorov's law exposed by Loève [28]. A third, more simplified proof was published by Derman and Sacks [20], who again relied on the conditional version of Kolomogorov's law, streamlined the chain of inequality manipulations with Dvoretzky's bounding series parameters $(\alpha_n, \beta_n, \gamma_n)$ and used Chebyshev's inequality and the Borel-Cantelli lemma to arrive at a very short proof. Robbins and Siegmund generalized the theorem to the context where the variables take value in generic Hilbert spaces using the methods of supermartingale theory [29], as did Venter [40]. For a survey see Lai [27]. Dvoretzky himself published a revisited version in [22].

We have chosen to formalise the proof following Derman and Sacks [20] as this version appeared to us as being the shortest and most suitable to formalize using constructions from our library of formalized probability theory.

In this paper we present complete formalization of the scalar version of Dvoretzky's theorem, with random variables taking value in $\mathbb{R}$.

Here is a full statement of Dvoretzky's theorem:

▶ **Theorem 4** (Regular Dvoretzky's Theorem ✿)**.** *Assuming the following:*
H₁ : *Let* $(\Omega, \mathcal{F}, P)$ *be a probability space*
H₂ : *For* $n = 1, 2, \ldots$
H₃ : *Let* $\mathcal{F}_n$ *be an increasing sequence of sub* σ*-fields of* $\mathcal{F}$
H₄ : *Let* X_n *be* $\mathcal{F}_n$*-measurable random variables taking values in* $\mathbb{R}$*.*
H₅ : *Let* $T_n : \mathbb{R}^n \to \mathbb{R}$ *be a measurable function*
H₆ : *Let* W_n *be* $\mathcal{F}_{n+1}$*-measurable random variables taking values in* $\mathbb{R}$ *such that*

$$X_{n+1} = T(x_1, \ldots, x_n) + W_n$$

H₇ : $\mathbb{E}(W_n|\mathcal{F}_n) = 0$
H₈ : $\sum_{n=1}^{\infty} \mathbb{E}W_n^2 < \infty$
H₉ : *Let* $\alpha_n, \beta_n, \gamma_n$ *be a series of real numbers such that*
H₁₀ : $\alpha_n \geq 0$
H₁₁ : $\beta_n \geq 0$
H₁₂ : $\gamma_n \geq 0$
H₁₃ : $\lim_{n=\infty} \alpha_n = 0$
H₁₄ : $\lim_{n=\infty} \sum_{k=1}^{n} \beta_k < \infty$
H₁₅ : $\lim_{n=\infty} \sum_{k=1}^{n} \gamma_k = \infty$
H₁₆ : *Let* x_* *be a point in* $\mathbb{R}$ *such that for all* $n = 1, 2, \ldots$ *and for all* $x_1, \ldots, x_n \in \mathbb{R}$*,*

$$|T_n(x_1, \ldots, x_n) - x_*| \leq \max(\alpha_n, (1 + \beta_n)|x_n - x_*| - \gamma_n)$$

Then the sequence of random variables $X_1, X_2, \ldots$ *converges with probability 1 to* x_**:*

$$P\{\lim_{n=\infty} X_n = x_*\} = 1$$

An increasing sequence $\mathcal{F}_n$ of sub-σ-fields of $\mathcal{F}$ (a filtration) formalizes a notion of a discrete stochastic process moving forward in time steps n, where $\mathcal{F}_n$ formalizes the history of the process up to the time step n. Assuming an $\mathcal{F}_n$-measurable random variable X_n means assuming a stochastic variable X_n that is included into the history up to the time step n.

We have also formalized the extended version of Dvoretzky's theorem in which $\alpha_n, \beta_n, \gamma_n$ are promoted to real valued functions and T_n is promoted to be an $\mathcal{F}_n$-measurable random variable. The hypotheses that have been modified in the extended version are marked by the symbol $\star$ below:

▶ **Theorem 5** (Extended Dvoretzky's theorem ✿). *Assuming the following:*
$\mathbf{H_1}$: *Let* $(\Omega, \mathcal{F}, P)$ *be a probability space*
$\mathbf{H_2}$: *For* $n = 1, 2, \ldots$
$\mathbf{H_3}$: *Let* $\mathcal{F}_n$ *be an increasing sequence of sub* σ*-fields of* $\mathcal{F}$
$\mathbf{H_4}$: *Let* X_n *be* $\mathcal{F}_n$*-measurable random variables taking values in* $\mathbb{R}$.
$\star\mathbf{H_5}$: *Let* T_n *be* $\mathcal{F}_n$*-measurable* $\mathbb{R}$*-valued random variable*
$\mathbf{H_6}$: *Let* W_n *be* $\mathcal{F}_{n+1}$*-measurable* $\mathbb{R}$*-valued random variables such that:*

$$X_{n+1} = T(x_1, \ldots, x_n) + W_n$$

$\mathbf{H_7}$: $\mathbb{E}(W_n|\mathcal{F}_n) = 0$
$\mathbf{H_8}$: $\sum_{n=1}^{\infty} \mathbb{E}W_n^2 < \infty$
$\star\mathbf{H_9}$: *Let* $\alpha_n, \beta_n, \gamma_n : \Omega \to \mathbb{R}$ *be functions*[6] *such that:*
$\mathbf{H_{10}}$: $\alpha_n \geq 0$
$\mathbf{H_{11}}$: $\beta_n \geq 0$
$\mathbf{H_{12}}$: $\gamma_n \geq 0$
$\star\mathbf{H_{13}}$: $\lim_{n\to\infty} \alpha_n = 0$ *with probability 1*
$\star\mathbf{H_{14}}$: $\lim_{n\to\infty} \sum_{k=1}^{n} \beta_k < \infty$ *with probability 1*
$\star\mathbf{H_{15}}$: $\lim_{n\to\infty} \sum_{k=1}^{n} \gamma_k = \infty$ *with probability 1*
$\mathbf{H_{16}}$*: Let* x_* *be a point in* $\mathbb{R}$ *such that for all* $n = 1, 2, \ldots$ *and for all* $x_1, \ldots, x_n \in \mathbb{R}$ *we have:*

$$|T_n(x_1, \ldots, x_n) - x_*| \leq \max(\alpha_n, (1 + \beta_n)|x_n - x_*| - \gamma_n) \tag{20}$$

Then the sequence of random variables $X_1, X_2, \ldots$ *converges with probability 1 to* x_**:*

$$P\{\lim_{n\to\infty} X_n = x_*\} = 1$$

We now turn to describing the formalization of the above theorems. First, we give a description of our comprehensive supporting Probability Theory library in Section 3 (which may be of independent interest), then we shall give an overview of the proof of Theorem 4 in Section 4.1, and finally detail the variants of this theorem we have formalized in Section 4.2.

3 Formalized Probability Library

Our formalization of both Dvoretzky theorems is built on top of our general library of formalized Probability Theory. In particular, we are not restricted to discrete probability measures.

[6] Technically, Dvoretzky in his revisited paper [22] requires $\alpha_n, \beta_n, \gamma_n$ to be $\mathcal{F}_n$-measurable, but this assumption wasn't actually used in the proof, so we have omitted it.

3.1 σ-Algebras and Probability Spaces

We first introduce `pre_events` ✿ which are just subsets of a type `T` i.e., maps `T` → `Prop`. Then we define σ-algebras, `SigmaAlgebra(T)` ✿, as collections of `pre_events` which are closed under countable union and complement and include the full subset of all elements in `T`:

```
Class SigmaAlgebra (T : Type) :=
    {
    sa_sigma : pre_event T → Prop;
    sa_countable_union (collection: nat → pre_event T) :
      (forall n, sa_sigma (collection n)) →
      sa_sigma (pre_union_of_collection collection);
    sa_complement (A:pre_event T) :
      sa_sigma A → sa_sigma (pre_event_complement A) ;
    sa_all : sa_sigma pre_Ω
    }.
```

Then, we label `pre_events` which are members of a σ-algebra as `events` ✿. Special σ-algebras, like that generated by a set of `pre_events` ✿ and the Borel σ-algebra ✿, are constructed as usual.

One interesting feature of the formalization of both of these is that they are both provided with alternative characterizations, which is useful for using the definitions. For the borel σ-algebra, we define two variants: `borel_sa` ✿, defined as the σ-algebra generated by the half-open intervals, and `open_borel_sa` ✿, defined as the σ-algebra generated by the open sets. After proving that the definitions yield the same σ-algebra ✿, we can choose which definition is simpler to work with in a given context, simplifying some proofs.

For the definition of $\sigma(X)$, the σ-algebra generated by a set X, we start with the standard definition ✿: the intersection ✿ of the set of σ-algebras that contain X ✿. This is useful, but as it is non-constructive, it lacks a convenient induction principle. As an alternative, we define the explicit closure of a set of `events` ✿, built by starting with the set (augmented by Ω), and repeatedly adding in complements and countable unions. In Coq, this is naturally defined using an inductive data type. This closure is then shown to be (a σ-algebra ✿ and) equivalent to $\sigma(X)$ ✿.

While definitions generally use the standard definition of $\sigma(X)$, some theorems are more easily proven by switching to the equivalent closure-based characterization. This enables induction, providing an easy way to extend a property on the generating set to the generated σ-algebra, by showing that complements and countable unions preserve the property in question.

Next, we introduce probability spaces ✿ over a σ-algebra, equipped with a measure mapping each `event` to a real number r, such that $0 \leq r \leq 1$.

```
Class ProbSpace {T : Type} (σ : SigmaAlgebra T) :=
{
  ps_P : event σ → R;
  ps_proper :> Proper (event_equiv ==> eq) ps_P ;
  ps_countable_disjoint_union (collection: nat → event σ) :
  (* Assume: collection is a subset of Sigma and
     its elements are pairwise disjoint. *)
    collection_is_pairwise_disjoint collection →
    sum_of_probs_equals ps_P collection (ps_P (union_of_collection collection));
  ps_one : ps_P Ω= R1;
  ps_pos (A:event σ): (0 <= ps_P A)
}.
```

The usual properties of probability spaces, such as monotonicity ✿, complements ✿, and non-disjoint unions ✿, are verified.

3.2 Almost Everywhere

Having defined probability spaces, we can introduce a commonly used assertion in probabilistic proofs: that a certain property holds *almost everywhere* on a probability space. By this we mean the set of points where the property holds includes a measurable event of measure 1. We define a predicate `almost` ✿ to indicate propositions which hold almost everywhere. It is parameterized by a probability space and proposition on that space.

```
Definition almost {Ts:Type} {dom: SigmaAlgebra Ts}(prts: ProbSpace dom) (P:Ts → Prop)
  := exists E, ps_P E = 1 ∧ forall x, E x → P x.
```

We have introduced machinery to make it more convenient to reason about `almost` propositions. For example, if we want to show that `almost P → almost Q → almost R`, we reduce the proof to showing that `almost (P → Q → R)` ✿, which itself is implied by `P → Q → R` ✿. Usual theorem proving tools can then be used.

On top of the basic `almost` definition, we defined `almostR2` ✿, which says that a binary relation holds `almost` everywhere.

```
Definition almostR2 (R:Td→ Td→ Prop) (r1 r2:Ts → Td) : Prop
  := almost (fun x ⇒ R (r1 x) (r2 x)).
```

This is useful, since it inherits many properties from the base relation (e.g. it is a preorder if the base relation is ✿), and simplifies definitions.

3.3 Measurability and Expectation

We next introduce the concept of measurable functions with respect to two σ-algebras. Since we are focusing on probability spaces, we call these measurable functions `RandomVariables` ✿.

```
(* A random variable is a mapping from a probability space to a sigma algebra. *)
Class RandomVariable {Ts:Type} {Td:Type}
      (dom: SigmaAlgebra Ts)
      (cod: SigmaAlgebra Td)
      (rv_X: Ts → Td)
  := (* for every element B in the sigma algebra, the preimage
     of rv_X on B is an event in the probability space *)
    rv_preimage_sa: forall (B: event cod), sa_sigma (event_preimage rv_X B).
```

In order to define the `Expectation` of a `RandomVariable`, we follow the usual technique of first treating the case of finite range functions ✿, then extending to nonnegative functions ✿ (resulting in an extended real) and then to general random variables. In the general case, the expectation is the difference of the expectation of the positive and negative parts of a random variable. ✿ Exceptions are handled using the Coq `option` type. For example, the difference of the expectations of the positive and negative parts of a random variable is not defined if they are both the same infinity. This exception is captured by allowing the difference to be `None` in that case. A well defined Expectation yields `Some r`, for some value in Coquelicot's `Rbar` type [16]. This represents a value in the extended reals: either a `Finite` real value, or positive or negative infinity (`p_infty` or `m_infty`).

```
Definition Expectation (rv_X : Ts → R) : option Rbar :=
  Rbar_minus' (NonnegExpectation (pos_fun_part rv_X))
              (NonnegExpectation (neg_fun_part rv_X)).
```

Originally our results about `Expectation` were for random variables taking images in the reals, but as we introduced limiting processes we needed to extend our definition to random variables taking values in the extended reals (`Rbar`).

This requires extending the support for limits in Coquelicot, allowing for sequences of functions over the extended reals ✿. The approach we took was to copy over all the definitions and lemmas in Coquelicot's `Lim_seq` module, extending them as appropriate, and re-proving them. A few changes were made, such as defining the extended version of `is_lim_seq` ✿ to hold when the inf and sup sequence limits coincide. The original definition uses filters, and is problematic to extend to the extended reals, since they do not form a uniform space. Pleasantly, however, almost all of the lemmas continue to hold with minor modification.

The above construction of Expectation and its properties (including linearity ✿✿, the monotone convergence theorem ✿, and other standard results) are then generalized and proven for this generalization to functions whose image is the extended reals ✿.

On top of our general definition of `Expectation`, we define the `IsFiniteExpectation` property, which asserts that a function has a well-defined, finite expectation ✿✿. For functions that satisfy this property, we can define their `FiniteExpectation` ✿✿, which returns their (real) expectation. This simplifies working with such functions, and avoids otherwise necessary side-conditions on properties such as linearity ✿✿.

3.4 L^p Spaces

Using these building blocks, we can define L^p spaces, which are the space of measurable functions where the p-th power of its absolute value has finite expectation ✿.

```
Definition IsLp {Ts} {dom: SigmaAlgebra Ts} (prts: ProbSpace dom) (n:R) (rv_X:Ts→ R)
    := IsFiniteExpectation prts (rvpower (rvabs rv_X) (const n)).
```

This space is then quotiented, identifying functions that are equal `almost` everywhere (see Section 3.2) ✿. We use a quotient construction ✿ that avoids needing axioms beyond those already proposed in Coq's standard libraries[7]. This quotienting operation is required in order to define a norm on the space (defined as the p-th root of the `Expectation` of the absolute value of the p-th power of the function), as having a zero `Expectation` only implies that a non-negative function is zero `almost` everywhere.

For nonnegative p, L^p is shown to be a module space ✿. For $1 \leq p \leq \infty$, it is shown to be a Banach space (complete normed module space) ✿ ✿.

Furthermore, the important special case of L^2 is proven to be a Hilbert space ✿, where the inner product of x and y is defined as the `Expectation` of the product of x and y.

3.5 Conditional Expectation

Building on top of this work, we turn to the definition of conditional expectation, defining it with respect to a general σ-algebra `dom2` (the ambient σ-algebra being `dom`). We first postulate a relational definition ✿, characterized by the universal property of conditional expectations: for any event P that is in the sub σ-algebra `dom2`, if we multiply the original function and its conditional expectation by that event's associated indicator function, we get equal expectations.

[7] Specifically, we use functional and propositional extensionality as well as constructive definite description (also known as the axiom of unique choice).

```
Definition is_conditional_expectation {Ts:Type} {dom: SigmaAlgebra Ts}
    (prts: ProbSpace dom) (dom2 : SigmaAlgebra Ts)
    (f : Ts → R) (ce : Ts → Rbar)
    {rvf : RandomVariable dom borel_sa f}
    {rvce : RandomVariable dom2 Rbar_borel_sa ce}
  := forall P (dec:dec_pre_event P),
    sa_sigma (SigmaAlgebra := dom2) P →
    Expectation (rvmult f (EventIndicator dec)) =
    Rbar_Expectation (Rbar_rvmult ce (EventIndicator dec)).
```

Using this definition, we can show uniqueness (where equality is almost everywhere) ✿, and many standard properties of conditional expectation, such as linearity ✿✿, preservation of `Expectation` ✿, (`almost`) monotonicity ✿, and the tower law ✿. We also show the "factor out" property ✿, which enables factoring out of a conditional expectation a random variable that is measurable with respect to the sub σ-algebra. In addition, we verify its interactions with limits (e.g. the conditional version of the monotone convergence theorem ✿), and prove Jensen's lemma ✿, bounding how convex functions affect the conditional expectation.

After having proven these properties for the `is_conditional_expectation` relation, we still need to show that the conditional expectation generally exists (at least for functions that are non-negative or have finite expectation).

To do this, we build on our work on L^p spaces (Section 3.4), and in particular our proof that that L^2 is a Hilbert space. Given an L^2 function, this implies that the subset of functions which are measurable with respect to a smaller σ-algebra `dom2` forms a linear subspace.

The L^2 conditional expectation ✿ of an L^2 random variable X with respect to `dom2` is then defined as the orthogonal projection ✿ of X onto that subspace. For this construction and definitions of Hilbert spaces we use the library from the formal development of the Lax-Milgram theorem [14]. Note that this definition is for a function in the quotiented space (recall that L^2 is quotiented to identify functions that are equal almost everywhere).

We can then define conditional expectation on the unquotiented space by injecting the inputs into the quotiented space, using the conditional expectation operator just defined on L^2 functions, and then choosing a representative from the equivalence class of functions it returns ✿. This unquotienting gives insight into why most theorems about conditional expectations only `almost` hold, as it is defined on equivalence classes of `almost` equal functions.

Next, we extend our notion of conditional expectation to nonnegative functions whose usual expectation is finite using the property that L^2 functions are dense in L^1. In particular, given a nonnegative L^1 function f, we can define an L^2 sequence $g_n = \min(f, n)$. The conditional expectation of f is defined as the limit of the conditional expectation of the g_n ✿.

Using this definition directly has some disadvantages: it forces essentially all theorems, including simple ones such as the result being non-negative, or that the conditional expectation is the identity operation on functions that are already measurable with respect to the sub σ-algebra `dom2`, be only `almost` valid. To address this, we wrap this definition in a wrapper ✿ that takes the function returned by the original (limit based) definition and tweaks it slightly, producing a "fixed" function `almost` equivalent to the original, but where such simple properties hold unconditionally.

Finally, we extend this to all measurable functions by taking the difference of the nonnegative conditional expectation of its positive and negative parts ✿. While this function is defined for all measurable functions, it can only be shown to be a conditional expectation (the `is_conditional_expectation` relation defined above) for functions that are either non-

negative ✿ or have finite expectation ✿. Using this property, we now lift all of the properties proven above for the relational version to our explicitly defined version, verifying that it satisfies all the expected properties. For convenience, we also provide a wrapper definition `FiniteConditionalExpectation` ✿, which assumes that the function has finite expectation, and returns a function whose image is in $\mathbb{R}$ (insted of the extended reals), and lift all the expected properties to it.

Connecting back to L^p spaces, we can use Jensen's lemma about convex functions to show that if a function is in L^p then its conditional expectation is as well ✿, allowing us to view conditional expectation as a (contractive ✿) operation on L^p spaces. Furthermore, we show that it minimizes the L^2-loss for an L^2 function ✿.

We chose this approach to defining conditional expectation (via an orthonormal projection on L^2) since we could rely on an existing library of Hilbert space theory [14], thus avoiding other tedious constructions involving Radon-Nikodym derivatives etc.

3.6 Filtrations and Martingales

We next introduce a notion of σ-algebra filtrations ✿, which are an increasing sequence of σ-algebras. We say that a sequence of random variables X_n `IsAdapted` ✿ to a filtration F_n, if each random variable of the sequence is measurable with respect the corresponding σ-algebra.

Building on these definitions and our development of conditional expectation, we started developing the basics of martingale theory ✿.

Additionally, the language of filtrations and adapted processes enables us to represent the history of a stochastic process, which is critical for stating and verifying properties of stochastic approximation methods.

3.7 Additional results

There are many other results proven in the library; here we highlight two that are used in the Derman-Sacks proof: Chebyshev's inequality and the Borel-Cantelli lemma.

Chebyshev's inequality ✿ which states that given a random variable X and a positive constant a, the probability of $\|X\| \geq a$ is less that or equal to the expectation of X^2/a^2.

```
Lemma Chebyshev_ineq_div_mean0
(X : Ts → R) (rv : RandomVariable dom borel_sa X) (a : posreal) :
Rbar_le (ps_P (event_ge dom (rvabs X) a))
        (Rbar_div_pos
        (NonnegExpectation (rvsqr X))
        (mkposreal _ (rsqr_pos a))).
```

Another is the Borel-Cantelli lemma ✿ which states that if the sum of probabilities of a sequence of events is finite, then the probability of all but finitely many of them occuring is 0.

```
Theorem Borel_Cantelli (E : nat → event dom) :
(forall (n:nat), sa_sigma (E n)) →
ex_series (fun n ⇒ ps_P (E n)) →
ps_P (inter_of_collection
        (fun k ⇒ union_of_collection
                    (fun n ⇒ E (n + k)))) = 0.
```

In this theorem statement, `ex_series f`, defined in Coquelicot, assert that the infinite series of partial sums $\lim_{n\to\infty} \sum_{0\leq i\leq n} f(i)$ converges to a finite limit.

3.8 Retrospective design decisions

In this section we discuss some of the design choices we made (and revisited), and our retrospective opinion on their impact. This may be of benefit to those seeking to pursue similar projects.

Initially, we modeled events as sets (now called `pre_events`), accompanying them with proofs that the set was in a given σ-algebra. This resulted in a lot of code threading and transforming these proofs, which was particularly painful when reasoning about lists. Revisiting that decision, we built up `events` as a subset type: a dependent pair of a `pre_event` and a proof that it belongs in a relevant σ-algebra ✿. For many simple uses, this obviates the need for reasoning explicitly about being in a σ-algebra. There are still cases where explicit reasoning is required, but this change definitely simplified the code.

We support general random variables using a typeclass which specifies the sigma algebras for the domain and range along with the function ✿. An initial version of our probability library developed expectation and properties of random variables whose codomain was the reals ✿. However as we proved more properties, especially limiting and convergence properties, it became clear that we needed to allow infinite values, thus to allow random variables with $\bar{\mathbb{R}}$ (the extended real numbers) as codomain ✿. However the native support for limits in the Coquelicot package allows the limiting value to be infinite, but restricts to sequences taking values in $\mathbb{R}$. In order to be able to take limits of random variables to $\bar{\mathbb{R}}$, we developed our own limit package extending the Coquelicot definitions and lemmas to sequences taking values in $\bar{\mathbb{R}}$ ✿. In the end, some results about $\bar{\mathbb{R}}$ valued random variables become simpler since one doesn't need to make unnatural finiteness restrictions, but on the other hand, one needs to be extra careful, since $\bar{\mathbb{R}}$ is not a field as sums and products are not always defined and operations are not associative.

As in the standard development of expectation, we first defined it for functions whose range is a finite subset of $\mathbb{R}$ (or $\bar{\mathbb{R}}$). We decided to represent them as a typeclass which includes a field containing a finite list of values which includes all the values in the range of the function ✿.

```
Class FiniteRangeFunction
      (rv_X:Ts→ Td)
  := {
  frf_vals : list Td ;
  frf_vals_complete : forall x, In (rv_X x) frf_vals;
    }.
```

We decided to allow this list to have duplicates and to contain additional values not in the range. This made several definitions more convenient, for example when defining the sum of two finite range functions, the new list of values is just the sum of all pairs of values, which is guaranteed to contain all the actual values, but can contain values which are not in the image of the sum and can contain duplicated values ✿.

One simplification our code makes is that we deal only with probability spaces, rather then general measures. This was a pragmatic decision, as it simplifies some of the proofs (since, for example, measures must be finite). As our intended use is probability theory, this mostly sufficed. In order to define (dependent) product spaces, we did define the rudiments of measure theory (measures, outer measures, and inner measures) ✿, but the final construction of the product is defined only for probability spaces, since the proof crucially relies on the monotone convergence theorem, which we have not proven for general measure spaces. It would clearly have been nicer to define things more generally, and we may go back and change things in the future, however this simplification allowed us to use our limited resources to greater effect.

4 Formalization Challenges/Overview

We will now sketch the key pieces which go into the formalization of the Derman-Sacks proof.

4.1 Overview of the proof

The Derman-Sacks proof relies on a number of prerequisites in Probability Theory and Real Analysis. For example, the proof begins by stating that we may replace the series $\sum_n \mathbb{E}W_n^2 < \infty$ by the series $\sum_n \frac{\mathbb{E}W_n^2}{\alpha_n^2} < \infty$ where $\alpha_n \to 0$. This statement invokes a classical theorem of du Bois-Reymond [13] which states:

▶ **Theorem 6 (✿).** *Let (a_n) be a sequence of nonnegative real numbers. The series $\sum_n a_n$ converges if and only if there is another sequence of positive real numbers (b_n) such that $b_n \to \infty$ and $\sum_n a_n b_n < \infty$.*

In other words, this theorem states that *no worst convergent series exists* (see [5]). This elementary theorem did require some effort to formalize, in part because existing proofs such as the one in [5] require the sequence (a_n) to consist only of positive terms, while our application (Dvoretzky's theorem) needed them to be non-negative. Additionally, we had to prove convergence of the product series without using the integral test (as used in [5]), because it was unavailable in our library. Our final proof of Theorem 6 involved a case analysis in which we case on whether the sequence (a_n) was eventually positive or not ✿, and we bypassed the need to use the integral test by using an exercise from Rudin's *Principles of Mathematical Analysis* [31].

The main workhorse of the Derman-Sacks proof is the sequence $Z_n := W_n \operatorname{sgn} T_n$. First, they apply the following theorem[8] to the sequence of random variables (Z_n):

▶ **Theorem 7** (Loève [28] ✿). *Let $X_1, X_2, \ldots$ be a sequence of random variables adapted to a filtration $(\mathcal{F}_n)_{n\in\mathbb{N}}$. Assume that $\mathbb{E}[X_{n+1} \mid \mathcal{F}_n] = 0$ almost surely for all n and also that $\sum_{n=1}^{\infty} \mathbb{E}X_n^2$ converges. Then we have that $\sum_{n=1}^{\infty} X_n$ converges almost surely.*

to conclude that the series $\sum_n Z_n$ converges almost surely. To apply this theorem we need to prove that (Z_n) is adapted to the filtration $\mathcal{F}$, which critically uses the fact that $T_n : \mathcal{H}^n \to \mathcal{H}$ is a measurable function. (Here we take $\mathcal{H} = \mathbb{R}$.) The proof of the theorem uses $\mathbb{E}[X_{n+1} \mid \mathcal{F}_n] = 0$ to show that since the sequence is adapted, we have $\mathbb{E}[X_i X_j] = 0$ for all $i \neq j$. This depends on the "factor out" property of conditional expectation ✿ (see Section 3.5).

Next, it is shown that $|Z_n| \leq \alpha_n$ almost surely for sufficiently large n. This argument uses the Borel-Cantelli lemma ✿ and the Chebyshev inequality ✿, both of which needed a significant amount of probability theory to be set up (see Section 3.7). Using this bound for Z_n and the bound for $|T_n|$ in the hypothesis, an elementary argument shows that

$$|X_{n+1}| \leq \max(2\alpha_n, |T_n| + Z_n) \leq \max(2\alpha_n, (1+\beta_n)|X_n| + Z_n - \gamma_n)$$

almost surely for sufficiently large n.

Now, the conclusion $X_{n+1} \to 0$ almost surely follows by applying the following lemma:

[8] the proof of this theorem is a modification of Theorem 6.2.1 in Ash's *Probability and Measure Theory* [6]

▶ **Lemma 8 (✿).** *Let* $\{a_n\}, \{b_n\}, \{c_n\}, \{\delta_n\}$ *and* $\{\xi_n\}$ *be sequences of real numbers such that*

1. $\{a_n\}, \{b_n\}, \{c_n\}, \{\xi_n\}$ *are non-negative*
2. $\lim_{n\to\infty} a_n = 0,\ \sum_n b_n < \infty,\ \sum_n c_n = \infty,\ \sum_n \delta_n$ *converges.*
3. *For all* n *larger than some* N_0, $\xi_{n+1} \leq \max(a_n, (1+b_n)\xi_n + \delta_n - c_n)$

then, $\lim_{n\to\infty} \xi_n = 0$.

The proof of the lemma is somewhat unusual since it involves running an iteration backwards: the property (3) is applied repeatedly to derive an inequality between ξ_{n+1} and ξ_N for $n > N > N_0$ ✿. Besides using several properties of infinite products and list maximums, the final convergence result is an application of Abel's descending convergence criterion ✿ which says if the series $\sum_n b_n$ converges, and a_n is a bounded descending sequence, then the series $\sum_n a_n b_n$ also converges.

We note that our formalization is firmly within the Classical territory for a number of reasons: first of all, the theory of Real numbers within the Coq standard library (which we use) uses non-computable axioms [23]. Secondly, while constructive measure theory and constructive analysis are both actively researched topics (see [19, 18, 11]) we are unaware if our main result (Dvoretzky's theorem) is constructively valid. Thirdly, as we remarked above, our proof of Theorem 6 requires a case split on whether a particular sequence of real numbers is eventually zero or not, for which we use the axiom of constructive indefinite description.

4.2 Variants of Dvoretzky's Theorem

While Dvoretzky's theorem admits generalizations in many different ways, we chose to focus on formalizing the ones most suited for applications.

1. As already mentioned, we prove Theorem 5 which is a generalization of Theorem 4 in which the sequences of numbers $\alpha_n,\ \beta_n,\ \gamma_n$ are replaced by sequences of functions on the probability space. This generalization is called the *extended* Dvoretzky theorem ✿. All conditions on the sequences $\alpha_n,\ \beta_n,\ \gamma_n$ now hold pointwise, almost everywhere.
2. To apply Theorem 7 in the proof of Theorem 4 we needed to prove that (Z_n) is adapted to the filtration $\mathcal{F}$, which needed us to make assumptions on the functions T_n. These assumptions on T_n can be modified and generalized as:
   a. in the regular (non-extended) case, $T_n : \mathbb{R}^n \to \mathbb{R}$ are deterministic and measurable. ✿
   b. in the extended case, $T_n : \mathbb{R}^n \times \Omega \to \mathbb{R}$ are stochastic and $\mathcal{F}_n$-adapted. ✿

   Since Derman-Sacks do not explicitly state either assumption, we formalized Dvoretzky's theorem under both assumptions. It should be noted that Dvoretzky's original paper [21] and his revisited paper [22] treat both the above cases.
3. We have also formalized a corollary of the extended Dvoretzky's theorem ✿ which proves that the theorem holds in the context where the bound on T in (14) is assumed as follows with all other assumptions intact:

   $$|T_n(x_1, \ldots, x_n) - x_*| \leq \max(\alpha_n, (1 + \beta_n - \gamma_n)|x_n - x_*|)$$

   While this formulation is weaker compared to the original, it is convenient to have it for several applications of stochastic approximation theorems. A proof of this corollary used a classical analysis result of Abel [1] on the fact that the terms in a divergent sum-series could be multiplied by infinitesimally small series and the sum-series would still diverge ✿. This was addressed in Dvoretzky's paper [21, (5.1)].

5 Related work

While our results are general, our intended application was formalizing machine learning theory, on which there is a growing body of work [34, 35, 37, 24, 32, 9, 10]. Our work is a step in this direction, providing future developers of secure machine learning systems a library of formalized stochastic approximation results. Keeping this in mind, we have formalized different versions of our main result (Dvoretzky's theorem) to facilitate ease of use (see Section 4.2).

For the formalization itself, we make extensive use of the Coquelicot library of Boldo et al. [16] and the library which proved the Lax-Milgram theorem [14] which includes definitions and basic properties of hilbert spaces. There have also been quite a few formalizations of probability theory in Coq: see Polaris [33], Infotheo [4], and Alea [7]. Alea is an early work and to the best of our knowledge incompatible with latest versions of Coq while Infotheo and Polaris either fundamentally focus on discrete probability theory (see [3]) or do not have the results we needed to prove Dvoretzky's theorem.

More recently there have been two projects in Coq which formalize measure theory and Lebesgue integration. The MathComp-Analysis project has general measure theory and integration developed on top of their library which is an alternative to Coquelicot [2]. The Numerical Analysis in Coq (coq-num-analysis) project is built on top of Coquelicot and includes support for Lebesgue integration of nonnegative functions [15]. Neither of these were available at the time we began to develop our probability library. Since we depend on Coquelicot, we could have developed on top of the coq-num-analysis library if it were available earlier. This would have given us the added benefit of supporting general measures instead of our restriction to probability measures. Refactoring our library to build on top of one or more of these formalizations might be a possible direction for future work.

Formal proofs about convergence of random variables (the Central Limit Theorem) have been given in Avigad et al [8] using the Isabelle/HOL system. Parts of Martingale theory and stochastic processes have also recently made their way into the Lean math library [36].

To the best of our knowledge, our work presents the first formal proof of correctness of any theorem in Stochastic Approximation.

6 Applications & Future Work

Our own interest in stochastic approximation began with an attempt to extend our work on convergence proofs of (model-based) Reinforcement Learning (RL) algorithms [39] to include the model-free case. Model-based RL algorithms converge to an *optimal policy* (a sequence of actions which an agent should probabilistically perform so as to maximize its expected long-term reward) by making full use of the given transition probability structure of the agent. The term *model-free* refers to the fact that we have no information on how the agent performs it's transitions but can only *observe* its transitions. As we have emphasized above, this situation is perfectly suited for stochastic approximation techniques. Indeed, convergence proofs of Q-Learning (a prominent model-free RL algorithm) appeal to standard results of stochastic approximation (see Watkins & Dayan [41], Jaakkola et al. [25]), Tsitsiklis [38]. We plan to use our formalization of Dvoretzky's theorem to complete a convergence proof of the Q-learning algorithm.

Additionally, as part of this process, we have built up a large library for basic results on (general) probability spaces in Coq, including a general definition of conditional expectation. This library is publically available at `https://github.com/IBM/FormalML` and open source. We invite others to use our library and collaborate with us on extending and enhancing it.

References

1 Niels Henrik Abel. Note sur le mémoire de Mr. L. Olivier No. 4. du second tome de ce Journal, ajant pour titre "*remarques sur les series infinies et leur convergence*". *Crelles Journal*, 3, 1828.

2 Reynald Affeldt and Cyril Cohen. Formalization of the Lebesgue measure in MathComp-Analysis. The Coq Workshop 2021, online, July 2, 2021, July 2021.

3 Reynald Affeldt, Jacques Garrigue, and Takafumi Saikawa. Reasoning with conditional probabilities and joint distributions in coq. *Computer Software*, 37(3):79–95, 2020. doi:10.11309/jssst.37.3_79.

4 Reynald Affeldt and Manabu Hagiwara. Formalization of Shannon's theorems in SSReflect-Coq. In *3rd Conference on Interactive Theorem Proving (ITP 2012), Princeton, New Jersey, USA, August 13–15, 2012*, volume 7406 of *Lecture Notes in Computer Science*, pages 233–249. Springer, August 2012.

5 J Marshall Ash. Neither a worst convergent series nor a best divergent series exists. *The College Mathematics Journal*, 28(4):296–297, 1997.

6 Robert B Ash, B Robert, Catherine A Doleans-Dade, and A Catherine. *Probability and measure theory*. Academic press, 2000.

7 Philippe Audebaud and Christine Paulin-Mohring. Proofs of randomized algorithms in Coq. *Science of Computer Programming*, 74(8):568–589, 2009.

8 Jeremy Avigad, Johannes Hölzl, and Luke Serafin. A formally verified proof of the central limit theorem. *Journal of Automated Reasoning*, 59(4):389–423, 2017.

9 Alexander Bagnall and Gordon Stewart. Certifying the True Error: Machine Learning in Coq with Verified Generalization Guarantees. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019*, pages 2662–2669. AAAI Press, 2019. doi:10.1609/aaai.v33i01.33012662.

10 Alexander Bentkamp, Jasmin Christian Blanchette, and Dietrich Klakow. A Formal Proof of the Expressiveness of Deep Learning. *J. Autom. Reason.*, 63(2):347–368, 2019. doi:10.1007/s10817-018-9481-5.

11 Errett Albert Bishop. Foundations of constructive analysis, 1967.

12 Julius R. Blum. Approximation Methods which Converge with Probability one. *The Annals of Mathematical Statistics*, 25(2):382–386, 1954. doi:10.1214/aoms/1177728794.

13 Paul du Bois-Reymond. Eine neue Theorie der Convergenz und Divergenz von Reihen mit positiven Gliedern., 1873.

14 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.

15 Sylvie Boldo, François Clément, Florian Faissole, Vincent Martin, and Micaela Mayero. A coq formalization of lebesgue integration of nonnegative functions. *Journal of Automated Reasoning*, pages 1–39, 2021.

16 Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for Coq. *Mathematics in Computer Science*, 9, March 2014. doi:10.1007/s11786-014-0181-1.

17 Han-Fu Chen. *Stochastic approximation and its applications*, volume 64. Springer Science & Business Media, 2006.

18 Thierry Coquand and Erik Palmgren. Metric boolean algebras and constructive measure theory. *Archive for Mathematical Logic*, 41(7):687–704, 2002.

19 Thierry Coquand and Bas Spitters. Integrals and valuations. *arXiv preprint arXiv:0808.1522*, 2008.

20 C Derman and J Sacks. On Dvoretzky's stochastic approximation theorem. *The Annals of Mathematical Statistics*, 30(2):601–606, 1959.

21 A. Dvoretzky. On stochastic approximation. In *Proceedings of the Third Berkeley Symposium on Mathematical Statistics and Probability*. University of California Press, 1956.

22 Aryeh Dvoretzky. Stochastic approximation revisited. *Advances in Applied Mathematics*, 7(2):220–227, 1986. doi:10.1016/0196-8858(86)90033-3.

23 Herman Geuvers and Milad Niqui. Constructive reals in coq: Axioms and categoricity. In *International Workshop on Types for Proofs and Programs*, pages 79–95. Springer, 2000.

24 Johannes Hoelzl. Markov processes in Isabelle/HOL. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 100–111, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3018610.3018628.

25 Tommi Jaakkola, Michael I Jordan, and Satinder P Singh. On the convergence of stochastic iterative dynamic programming algorithms. *Neural computation*, 6(6):1185–1201, 1994.

26 J. Kiefer and J. Wolfowitz. Stochastic Estimation of the Maximum of a Regression Function. *The Annals of Mathematical Statistics*, 23(3):462–466, 1952. doi:10.1214/aoms/1177729392.

27 Tze Leung Lai. Stochastic Approximation. *The Annals of Statistics*, 31(2):391–406, 2003. URL: http://www.jstor.org/stable/3448398.

28 Michel Loève. On almost sure convergence. Proc. Berkeley Sympos. math. Statist. Probability, California July 31 - August 12, 1950, 279-303 (1951)., 1951.

29 H. Robbins and D. Siegmund. A convergence theorem for non negative almost supermartingales and some applications. In Jagdish S. Rustagi, editor, *Optimizing Methods in Statistics*, pages 233–257. Academic Press, 1971. doi:10.1016/B978-0-12-604550-5.50015-8.

30 Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

31 Walter Rudin et al. *Principles of mathematical analysis*, volume 3. McGraw-hill New York, 1976.

32 Daniel Selsam, Percy Liang, and David L. Dill. Developing Bug-Free Machine Learning Systems With Formal Mathematics. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 3047–3056. PMLR, 2017. URL: http://proceedings.mlr.press/v70/selsam17a.html.

33 Joseph Tassarotti and Robert Harper. A separation logic for concurrent randomized programs. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019.

34 Joseph Tassarotti, Jean-Baptiste Tristan, and Koundinya Vajjha. A Formal Proof of PAC Learnability for Decision Stumps. *CoRR*, abs/1911.00385, 2019. arXiv:1911.00385.

35 Joseph Tassarotti, Koundinya Vajjha, Anindya Banerjee, and Jean-Baptiste Tristan. A formal proof of PAC learnability for decision stumps. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 5–17, 2021.

36 The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*, pages 367–381, 2020.

37 Jean-Baptiste Tristan, Joseph Tassarotti, Koundinya Vajjha, Michael L. Wick, and Anindya Banerjee. Verification of ML Systems via Reparameterization. *CoRR*, abs/2007.06776, 2020. arXiv:2007.06776.

38 John N Tsitsiklis. Asynchronous stochastic approximation and q-learning. *Machine learning*, 16(3):185–202, 1994.

39 Koundinya Vajjha, Avraham Shinnar, Barry Trager, Vasily Pestun, and Nathan Fulton. Certrl: formalizing convergence proofs for value and policy iteration in coq. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 18–31, 2021.

40 J. H. Venter. On Dvoretzky Stochastic Approximation Theorems. *The Annals of Mathematical Statistics*, 37(6):1534–1544, 1966. doi:10.1214/aoms/1177699145.

41 Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

42 J. Wolfowitz. On the Stochastic Approximation Method of Robbins and Monro. *The Annals of Mathematical Statistics*, 23(3):457–461, 1952. URL: http://www.jstor.org/stable/2236689.

43 J. Wolfowitz. On Stochastic Approximation Methods. *The Annals of Mathematical Statistics*, 27(4):1151–1156, 1956. doi:10.1214/aoms/1177728082.

Mechanizing Soundness of Off-Policy Evaluation

Jared Yeager
Manning College of Information and Computer Sciences,
University of Massachusetts Amherst, MA, USA

J. Eliot B. Moss
Manning College of Information and Computer Sciences,
University of Massachusetts Amherst, MA, USA

Michael Norrish
School of Computing, Australian National University, Canberra, Australia

Philip S. Thomas
Manning College of Information and Computer Sciences,
University of Massachusetts Amherst, MA, USA

Abstract

There are reinforcement learning scenarios – e.g., in medicine – where we are compelled to be as confident as possible that a policy change will result in an improvement before implementing it. In such scenarios, we can employ *off-policy evaluation* (OPE). The basic idea of OPE is to record histories of behaviors under the current policy, and then develop an estimate of the quality of a proposed new policy, seeing what the behavior would have been under the new policy. As we are evaluating the policy without actually using it, we have the "off-policy" of OPE. Applying a concentration inequality to the estimate, we derive a confidence interval for the expected quality of the new policy. If the confidence interval lies above that of the current policy, we can change policies with high confidence that we will do no harm.

We focus here on the mathematics of this method, by mechanizing the soundness of off-policy evaluation. A natural side effect of the mechanization is both to clarify all the result's mathematical assumptions and preconditions, and to further develop HOL4's library of verified statistical mathematics, including concentration inequalities. Of more significance, the OPE method relies on importance sampling, whose soundness we prove using a measure-theoretic approach. In fact, we generalize the standard result, showing it for contexts comprising both discrete and continuous probability distributions.

2012 ACM Subject Classification Theory of computation → Interactive proof systems; Theory of computation → Logic and verification; Theory of computation → Sequential decision making; Mathematics of computing → Hypothesis testing and confidence interval computation

Keywords and phrases Formal Methods, HOL4, Reinforcement Learning, Off-Policy Evaluation, Concentration Inequality, Hoeffding

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.32

Supplementary Material *Software (Source Code)*: `https://github.com/jdyeager/itp_ope`
archived at `swh:1:dir:a554d5232ce611cfaa8df6b8c4fa0c164e51b2bb`

Funding This material is based upon work supported by the National Science Foundation under Grant No. CCF-2018372. 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

Reinforcement learning (RL) algorithms are machine learning algorithms that learn to make sequences of optimal or nearly optimal decisions through interactions with their environment. Their use has been proposed for a variety of high-risk high-reward applications including improving sepsis treatment [21], insulin dosing for type 1 diabetes treatment [40], epilepsy

treatment [12], and for various applications related to autonomous vehicles [10]. However, none of these *proposed* applications have come to fruition, partially due to concerns about safety. If the RL algorithm proposed a new policy (mechanism for making decisions) that is worse than a currently used policy it could be dangerous or costly.

To ensure that the policies produced by RL algorithms are safe, RL researchers have recently increased their focus on *off-policy evaluation* (OPE) methods – methods that use historical data collected from the use of a current policy to estimate and bound the performance of a newly proposed policy without requiring the (possibly dangerous) newly proposed policy to actually be used [32, 18, 39]. These methods are based on *importance sampling*. At a high level, the idea of importance sampling for RL is to consider previous runs of the current policy (or policies) and to take a weighted average of the observed historical performance, where the weight corresponds to the likelihood of the historical observations under the newly proposed policy divided by their likelihood under the current policy. Because importance sampling provides unbiased estimates of the performance of the newly proposed policy [38], confidence intervals for the mean of a random variable (e.g., based on Hoeffding's inequality [14] or Student's t-test [34]) can be applied to obtain confidence intervals for the performance of the new policy if it were to be used.

Particularly in cases like medicine, we desire more than just confidence in a statistical result in the sense of confidence intervals, but also confidence that the overall *process* of OPE is sound. A step in that direction is mechanizing a proof of OPE's soundness, much like how other researchers provided mechanized proofs for the soundness of supervised learning generalization guarantees (as opposed to the reinforcement learning guarantees we provide) [3]. A complete program of establishing confidence would further prove correctness of a software implementation of OPE down to machine code – a step we leave to future work.

Concerning the mathematics of OPE, previous work has offered hand proofs for the cases of discrete probability distributions and continuous ones, but not for hybrid distributions – ones with both discrete and continuous components. Yet hybrid distributions arise quite naturally in practice because many systems have both discrete controls, such as on-off switches, and continuous ones, such as throttles and torque actuators. Another example is when a robot component experiences contact with an external object, which constrains actions and movement, versus no-contact when actions and movement are less constrained. Thus there can be a discrete probability that a throttle cannot be used in the current state, and in other states a continuous distribution of how much throttle to apply.

In addition to rounding out the mathematics of OPE by handling hybrid distributions, mechanizing a proof of OPE clarifies the preconditions of its soundness, ensuring that hand proofs did not overlook anything of significance.

1.1 Contributions

Our starting point is the existing HOL4 libraries, which include a number of theorems about probability and measure theory, but not much in the way of *statistics*.
The main results we present here are:

- OPE gives an unbiased estimate for any property (statistic) of the new policy representable as an *integrable* function on histories.
- Thus, more specifically, OPE gives an unbiased estimate of the standard RL performance metric: the expected discounted sum of rewards [35], also known as the expected *return*.
- Applying Hoeffding's inequality, we obtain a confidence interval for the expected return for any desired confidence level.

Other similar results we do not explicitly present here:

- OPE gives an unbiased estimate for any property (statistic) of the new policy that can be represented as a positive measurable function on histories.
- From that result we further show that OPE gives an unbiased estimate for any point of the cumulative distribution function of any property (statistic) of the new policy representable as a measurable function on histories.

Other contributions include:

- Proof of Hoeffding's inequality (and Hoeffding's Lemma) in HOL4.
- Machinery in HOL4 to represent trajectories and histories, and measure spaces over them.
- Machinery in HOL4 to represent arbitrary n-way product measure spaces.

Our HOL4 sources are available from `https://github.com/jdyeager/itp_ope`.

2 Background

First, let us further clarify the starting point for our work, namely theories already present in the HOL4 libraries. Hurd [16, 17] developed the original formulation of measure theory and probability. Coble [8] added Lebesgue integration, Radon-Nikodym derivative, and random variable theories. Mhamdi [26] added formalization of "almost everywhere" and proved Markov's inequality, and later refined integration to allow extended real results [27]. Tian [41] applied that extension to measure theory and probability. There are similar developments of much of this background theory in Isabelle/HOL (by Hölzl and Heller [15]), and in Coq (by Boldo et al. [5]).

While much of what OPE builds on is standard mathematics, such as measure theory, there are a few mathematical topics worthy of mention here. One is how Radon-Nikodym derivatives (already in HOL4) allow us to fold the discrete, continuous, and hybrid probability distribution cases into one general case. Another is the expression of OPE itself (not in HOL4). Among more standard results, we review briefly below Markov's inequality (in HOL4), Hoeffding's Lemma, and Hoeffding's inequality (both new to HOL4 – Hoeffding's inequality has been proven in Coq [3]). Before we review these mathematical topics, we first mention some standard definitions we use in our proofs:

Extended reals: The extended reals ($\bar{\mathbb{R}}$) consist of the real numbers augmented with $+\infty$ and $-\infty$, and are useful in capturing unbounded measures and integrals.

Sigma algebras: Let X be a set of points and Σ be a set of subsets of X, $\Sigma \subseteq 2^X$.[1] We say (X, Σ) is a *sigma algebra* if $X \in \Sigma$ and Σ is closed under complementation and countable unions. This appears in HOL4 as (note the indexing/extracting functions space and subsets):

$$\begin{aligned}
&\mathsf{subset\text{-}class}\ X\ \Sigma \overset{\text{def}}{=} \forall s.\ s \in \Sigma \Rightarrow s \subseteq X,\\
&\mathsf{algebra}\ a \overset{\text{def}}{=}\\
&\quad \mathsf{subset\text{-}class}\ (\mathsf{space}\ a)\ (\mathsf{subsets}\ a) \wedge \emptyset \in \mathsf{subsets}\ a \wedge\\
&\quad (\forall s.\ s \in \mathsf{subsets}\ a \Rightarrow \mathsf{space}\ a - s \in \mathsf{subsets}\ a) \wedge\\
&\quad \forall s\ t.\ s \in \mathsf{subsets}\ a \wedge t \in \mathsf{subsets}\ a \Rightarrow s \cup t \in \mathsf{subsets}\ a, \text{ and}\\
&\sigma\text{-}\mathsf{algebra}\ a \overset{\text{def}}{=} \mathsf{algebra}\ a \wedge \forall c.\ \mathsf{countable}\ c \wedge c \subseteq \mathsf{subsets}\ a \Rightarrow \bigcup c \in \mathsf{subsets}\ a.
\end{aligned}$$

There is a standard sigma algebra over the extended reals, called Borel in HOL4. This is useful when integrating measurable functions.

[1] Σ is a conventional name for the measurable sets, not to be confused with summation.

Measurable functions: For (X, Σ) and (Y, T) sigma algebras, $f : X \to Y$ is a *measurable function* (from (X, Σ) to (Y, T)) if $\forall E \in T : f^{-1}(E) \in \Sigma$. In HOL4:

$$\begin{aligned}&\vdash f \in \mathsf{measurable}\ a\ b \iff\\ &\quad \sigma\text{-}\mathsf{algebra}\ a \wedge \sigma\text{-}\mathsf{algebra}\ b \wedge f \in (\mathsf{space}\ a \to \mathsf{space}\ b) \wedge\\ &\quad \forall s.\ s \in \mathsf{subsets}\ b \Rightarrow f^{-1}\ s \cap \mathsf{space}\ a \in \mathsf{subsets}\ a.\end{aligned}$$

In HOL4 these are called *Borel-measureable* functions if (Y, T) is the Borel sigma algebra.

Measure spaces: For $\mu : \Sigma \to \bar{\mathbb{R}}$, we say μ is a *measure* and (X, Σ, μ) a *measure space* if: (X, Σ) is a sigma algebra; $\mu(\emptyset) = 0$; and μ is positive and countably additive. Expressed in detail in HOL4 this is:

$$\begin{aligned}&\mathsf{measure\text{-}space}\ m \stackrel{\mathsf{def}}{=}\\ &\quad \sigma\text{-}\mathsf{algebra}\ (\mathsf{m\text{-}space}\ m, \mathsf{measurable\text{-}sets}\ m) \wedge \mathsf{positive}\ m \wedge \mathsf{countably\text{-}additive}\ m, \text{ where}\\ &\mathsf{positive}\ m \stackrel{\mathsf{def}}{=} \mathsf{measure}\ m\ \emptyset = 0 \wedge \forall s.\ s \in \mathsf{measurable\text{-}sets}\ m \Rightarrow 0 \leq \mathsf{measure}\ m\ s, \text{ and}\\ &\mathsf{countably\text{-}additive}\ m \stackrel{\mathsf{def}}{=}\\ &\quad \forall f.\ f \in (\mathbb{N} \to \mathsf{measurable\text{-}sets}\ m) \wedge (\forall i\ j.\ i \neq j \Rightarrow \mathsf{DISJOINT}\ (f\ i)\ (f\ j)) \wedge\\ &\qquad \bigcup (\mathsf{IMAGE}\ f\ \mathbb{N}) \in \mathsf{measurable\text{-}sets}\ m \Rightarrow\\ &\qquad \mathsf{measure}\ m\ (\bigcup (\mathsf{IMAGE}\ f\ \mathbb{N})) = \mathsf{suminf}\ (\mathsf{measure}\ m \circ f).\end{aligned}$$

Sigma finite measure spaces: A measure space is *sigma finite* if it can be partitioned into countably many measurable sets of finite measure, or equivalently (as represented in HOL4), the space is the limit of measurable sets of finite measure:

$$\begin{aligned}&\mathsf{sigma\text{-}finite\text{-}measure\text{-}space}\ m \stackrel{\mathsf{def}}{=} \mathsf{measure\text{-}space}\ m \wedge \mathsf{sigma\text{-}finite}\ m, \text{ where}\\ &\mathsf{sigma\text{-}finite}\ m \stackrel{\mathsf{def}}{=}\\ &\quad \exists f.\ f \in (\mathbb{N} \to \mathsf{measurable\text{-}sets}\ m) \wedge (\forall n.\ f\ n \subseteq f\ (\mathsf{SUC}\ n)) \wedge\\ &\qquad \bigcup (\mathsf{IMAGE}\ f\ \mathbb{N}) = \mathsf{m\text{-}space}\ m \wedge \forall n.\ \mathsf{measure}\ m\ (f\ n) < +\infty.\end{aligned}$$

Almost Everywhere: A property is said to be true *almost everywhere* (a.e.) if the set of points where the property is false has measure zero. In HOL4 this appears as a quantifier:

$$\begin{aligned}&(\mathsf{AE}\ x :: m.\ P\ x) \stackrel{\mathsf{def}}{=} \exists N.\ \mathsf{null\text{-}set}\ m\ N \wedge \{\ x \mid x \in \mathsf{m\text{-}space}\ m \wedge \neg P\ x\ \} \subseteq N, \text{ where}\\ &\mathsf{null\text{-}set}\ m\ s \stackrel{\mathsf{def}}{=} s \in \mathsf{measurable\text{-}sets}\ m \wedge \mathsf{measure}\ m\ s = 0.\end{aligned}$$

Density: We will find it useful to take a measure space (X, Σ, μ) and positive function f, and re-weight μ by f, thus incorporating f into μ. This is done by integrating over f:

$$\begin{aligned}&\mathsf{density}\ m\ f \stackrel{\mathsf{def}}{=} (\mathsf{m\text{-}space}\ m, \mathsf{measurable\text{-}sets}\ m, f * m), \text{ where}\\ &f * m \stackrel{\mathsf{def}}{=} (\lambda s.\ \textstyle\int^+ m\ (\lambda x.\ f\ x \cdot \mathbb{1}\ s\ x)).\end{aligned}$$

Probability: Probability, both mathematically and in HOL4, is a renaming of measure theory concepts:

$$\begin{aligned}&\mathsf{prob\text{-}space}\ p \stackrel{\mathsf{def}}{=} \mathsf{measure\text{-}space}\ p \wedge \mathsf{measure}\ p\ (\mathsf{m\text{-}space}\ p) = 1,\\ &\mathsf{p\text{-}space} \stackrel{\mathsf{def}}{=} \mathsf{m\text{-}space},\\ &\mathsf{events} \stackrel{\mathsf{def}}{=} \mathsf{measurable\text{-}sets},\\ &\mathsf{prob} \stackrel{\mathsf{def}}{=} \mathsf{measure},\\ &\mathsf{real\text{-}random\text{-}variable}\ X\ p \stackrel{\mathsf{def}}{=}\\ &\quad \mathsf{random\text{-}variable}\ X\ p\ \mathsf{Borel} \wedge \forall x.\ x \in \mathsf{p\text{-}space}\ p \Rightarrow X\ x \neq -\infty \wedge X\ x \neq +\infty,\\ &\mathsf{random\text{-}variable}\ X\ p\ s \stackrel{\mathsf{def}}{=} X \in \mathsf{measurable}\ (\mathsf{p\text{-}space}\ p, \mathsf{events}\ p)\ s, \text{ and}\\ &\mathsf{expectation} \stackrel{\mathsf{def}}{=} \textstyle\int.\end{aligned}$$

For convenience of notation, we use $\mathrm{P}_p[s]$ and $\mathbb{E}_p[f]$ in lieu of prob and expectation respectively, where the subscript p is the probability space and the set s is intersected with p-space p.

Markov's inequality: This, perhaps less basic, result gives an upper bound to how much of a positive function is above a given threshold:

$$\vdash \text{prob-space } p \ \wedge \ \text{integrable } p\ X \ \wedge \ 0 \ < \ c \ \Rightarrow \\ \mathrm{P}_p[\{\ x \mid c \ \leq \ |X\ x|\ \}] \ \leq \ c^{-1} \cdot \mathbb{E}_p[(\lambda\, x.\ |X\ x|)].$$

We use this in proving Hoeffding's inequality.

Hoeffding's lemma: This result gives an upper bound to a moment generating function of a real-valued random variable with expectation 0, bounded almost everywhere between a and b:

$$\mathrm{E}\left[e^{cX}\right] \leq \exp\left(\frac{c^2(b-a)^2}{8}\right).$$

This is used in proving Hoeffding's inequality.

Hoeffding's inequality: This result bounds how much a sum of random variables deviates from its expectation. For n variables X_i, respectively bounded almost everywhere between a_i and b_i, for $t > 0$, where $S_n = \sum_{i=1}^{n} X_i$:

$$\mathrm{P}\left(S_n - \mathrm{E}\left[S_n\right] \geq t\right) \leq \exp\left(-\frac{2t^2}{\sum_{i=1}^{n}(b_i - a_i)^2}\right).$$

2.1 Off-Policy Evaluation

Here we present a standard mathematical formulation of an RL problem, known as a *partially observable Markov decision process* (POMDP) with a *state-free* policy [19, Section 7.1], using the motivating example of optimizing dosing for type 1 diabetes treatment [4, 40]. In this motivating application an RL algorithm is used to determine how much insulin should be injected into a person's blood prior to their eating each meal, in order to keep their blood glucose (blood sugar) near ideal levels.

Let $\mathcal{S}$, $\mathcal{O}$, and $\mathcal{A}$ be sets of states, observations, and actions respectively. For simplicity when first presenting these methods, we assume that these sets are finite, resulting in subsequent distributions being discrete. However, our formalization is for the general setting where these can be arbitrary measurable spaces and distributions over these sets can be discrete, continuous, or hybrid. In the diabetes treatment example, each state $s \in \mathcal{S}$ is a complete characterization of the patient and the meal they will eat. The RL agent does not know or observe the state, but rather makes an observation $o \in \mathcal{O}$. For example, in prior work [4, 40], the observation corresponded to a vector of three real numbers indicating the patients' current blood glucose level (from a sample of the person's blood), target blood glucose level (as specified by a doctor), and the size of the meal they are about to eat (in grams of carbohydrates, roughly estimated by the patient). Each action $a \in \mathcal{A}$ is a positive real number corresponding to an amount of insulin that could be injected.

The environment that an RL agent interacts with has state $S_t \in \mathcal{S}$ at time $t \in \{-1, 0, 1, \ldots, T-1\}$, where S_{-1} denotes an initial state. Here T, the number of time steps before the process terminates, is called the *horizon*. The initial state is sampled from an *initial state distribution* d_0. The RL agent does not necessarily observe the state itself, and instead makes some observation $O_t \in \mathcal{O}$ where $O_t \sim \Omega(\cdot|S_{t-1})$ – where the notation $X \sim Z(\cdot|Y)$ indicates drawing X from the joint distribution $Z(x, y)$ with y fixed with value

Y, or equivalently, drawing X from the distribution $\lambda x.Z(x, Y)$. Next the RL agent selects an action $A_t \in \mathcal{A}$ according to a policy π, where $A_t \sim \pi(\cdot|O_t)$. This action causes the state of the environment to change to $S_t \sim P(\cdot|S_{t-1}, A_t)$, where P is called the *transition function*. This transition of the environment results in the agent's receiving a real-valued reward, $R_t \sim d_R(\cdot|S_{t-1}, A_t, S_t)$.

A *trajectory* $\mathcal{T} = (S_{-1}, O_0, A_0, S_0, R_0, O_1, A_1, S_1, R_1, \ldots, O_{T-1}, A_{T-1}, S_{T-1}, R_{T-1})$ is the sequence of states, observations, actions, and rewards observed during one run from time $t = 0$ to time $t = T - 1$ (one such run is called an *episode*). A *history* H is similar, but contains only the terms that are known to the RL algorithm, namely $H = (O_0, A_0, R_0, O_1, A_1, R_1, \ldots, O_{T-1}, A_{T-1}, R_{T-1})$ (keeping the observations but dropping the states).

The *return* of an episode is the discounted sum of rewards, and can be viewed as a function g of the trajectory or history: $g(\mathcal{T}) = g(H) = \sum_{t=0}^{T-1} \gamma^t R_t$, where $\gamma \in [0, 1]$ is a parameter that discounts rewards that occur later in the trajectory. The performance of a policy π is the expected discounted sum of rewards that result from using the policy to make decisions: $J(\pi) = \mathrm{E}[g(H)|\pi]$.[2] We assume that we have access to n histories $(H_i)_{i=1}^n$ generated by past policies $(\beta_i)_{i=1}^n$, and also to a newly proposed policy π. OPE methods [32] use the historical data $D = (H_i)_{i=1}^n$ to estimate $J(\pi)$.

To further ground this notation and terminology, consider again our diabetes treatment example. Here, insulin injections are given prior to each meal, and each day is considered a separate episode. Assuming three meals per day, this corresponds to a horizon of $T = 3$. S_{-1} corresponds to a complete description of the patient and meal just prior to eating breakfast, O_0 is the observation about the state of the person and meal (blood glucose, target blood glucose, and meal size) just prior to breakfast, A_0 is the amount of insulin (a real number) to be injected prior to eating breakfast, S_0 is the state of the patient just prior to eating lunch, etc. The reward R_t is designed to penalize deviation from optimal blood glucose levels, with larger penalties for dangerously low blood levels, called *hypoglycemia*. The precise specification of these rewards must be carefully designed by experts to ensure that an agent that maximizes the expected discounted sum of rewards will produce the desired behavior [4, Page 11]. Current basic insulin dosage calculators determine the injection size using equations similar to:

$$\text{injection size} = \frac{\text{blood glucose} - \text{target blood glucose}}{CF} + \frac{\text{meal size}}{CR}, \tag{1}$$

where CF and CR are parameters chosen by a doctor [40], and which should be fine-tuned over time. When viewed as a policy, the injection size is the action A_t; and the blood glucose, target blood glucose, and meal size correspond to the observation O_t. If (1) were used to select actions, this would correspond to a *deterministic policy*. To make this policy stochastic (as required for off-policy evaluation and RL in general), one might add a Gaussian random variable with a mean of 0 (we will call this "noise") to the injection size, and $\pi(A_t|O_t)$ would then correspond to the probability of A_t being the action given observation O_t when using (1) with noise added.

However, the addition of noise to the action might result in unsafe injection sizes that deviate significantly (but with low probability) from the injection size intended by the doctor when they specified values for CF and CR. Though there are more sophisticated techniques

[2] The definition of $J(\pi)$ is *not* a conditional expected value. The conditioning notation indicates that the history H was generated by selecting actions using the policy π.

for creating a stochastic policy for this application that could be trusted [40], here we present one intuitive motivating alternative: the magnitude of the noise might be limited to some maximum value, $\eta_{\max}$. Let $\mathrm{E}[A_t|O_t]$ denote the expected injection size given observation O_t – that is, the injection prescribed by (1). Notice that the inclusion of clipped noise to the action results in the policy being a hybrid distribution, with probability density on the open interval $(\mathrm{E}[A_t|O_t] - \eta_{\max}, \mathrm{E}[A_t|O_t] + \eta_{\max})$ and probability masses at $\mathrm{E}[A_t|O_t] - \eta_{\max}$ and $\mathrm{E}[A_t|O_t] + \eta_{\max}$.

Lastly, for the diabetes treatment example the data D corresponds to data collected using initial values for CR and CF chosen by a doctor, with one history per day. The goal of an RL agent would be to use this data to find a new policy (new values for CR and CF) that results in an increased expected return. If the rewards, R_t, are defined appropriately, this would correspond to values for CR and CF that better regulate the patient's blood glucose levels. However, if the new policy is worse, it could have devastating consequences. For example, a single instance of severe hypoglycemia triples the five year mortality rate for a person with type 1 diabetes [25] and can have other severe consequences [40].

The importance sampling [20] estimator for $J(\pi)$ is then $\mathrm{IS} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{\Pr(H_i|\pi)}{\Pr(H_i|\beta_i)}g(H_i)\right)$. Denoting actions and observations at time t in the i^{th} trajectory or history as A_t^i and O_t^i respectively, some simplification shows IS is equivalent to [32, 38]:

$$\mathrm{IS} = \frac{1}{n}\sum_{i=1}^{n}\left(\rho_i(H_i)g(H_i)\right), \qquad \text{where } \rho_i(H_i) = \prod_{t=0}^{T-1}\frac{\pi(A_t^i|O_t^i)}{\beta_i(A_t^i|O_t^i)}.$$

Peer reviewed (but not machine verified) proofs have shown that, when for all i and H $\Pr(H|\beta_i) = 0 \to \Pr(H|\pi) = 0$ (a condition assumed from here on, for histories and trajectories), the importance sampling estimator is unbiased [32, 37]. That is, $\mathrm{E}[\mathrm{IS}] = J(\pi)$.

We now provide an overview of the proof that the importance sampling estimator is unbiased. Let $\mathcal{T}_\pi$ and $\mathcal{H}_\pi$ denote the sets of all possible trajectories and histories when using policy π. We write $\mathcal{T} \sim \pi$ or $H \sim \pi$ to denote that a trajectory or history is generated using the policy π. Similarly, when a policy is used as a subscript on a probability it indicates that the relevant random variables come from using the specified policy. For example, $\Pr_\pi(H)$ is the probability of history H when policy π is used. As with g, the ρ_i can be viewed as functions of histories or trajectories: $\rho_i(\mathcal{T}) = \rho_i(H) = \prod_{t=0}^{T-1}\frac{\pi(A_t^i|O_t^i)}{\beta_i(A_t^i|O_t^i)}$. We now begin with $J(\pi) = \mathrm{E}_{\mathcal{T}\sim\pi}\left[g(\mathcal{T})\right]$ and derive an equality to E[IS], following the four steps that we later use in our HOL4 proof. For the first three steps, it suffices to consider unindexed β and ρ.

1. We begin with a change of measure – changing from trajectories generated by π to trajectories generated by β.

$$\mathrm{E}_{\mathcal{T}\sim\pi}\left[g(\mathcal{T})\right] = \sum_{\mathcal{T}\in\mathcal{T}_\pi}\Pr_\pi(\mathcal{T})g(\mathcal{T}) = \sum_{\mathcal{T}\in\mathcal{T}_\pi}\Pr_\beta(\mathcal{T})\frac{\Pr_\pi(\mathcal{T})}{\Pr_\beta(\mathcal{T})}g(\mathcal{T}) = \mathrm{E}_{\mathcal{T}\sim\beta}\left[\frac{\Pr_\pi(\mathcal{T})}{\Pr_\beta(\mathcal{T})}g(\mathcal{T})\right],$$

2. Next we show that the ratio of the probability of $\mathcal{T}$ under π divided by the probability under β does not depend on functions that are not known in practice (e.g., the transition and observation functions P and Ω). That is:

$$\begin{aligned}\frac{\Pr_\pi(\mathcal{T})}{\Pr_\beta(\mathcal{T})} &= \frac{d_0(S_{-1})\prod_{t=0}^{T-1}\Omega(O_t|S_{t-1})\pi(A_t|O_t)P(S_t|S_{t-1},A_t)d_R(R_t|S_{t-1},A_t,S_t)}{d_0(S_{-1})\prod_{t=0}^{T-1}\Omega(O_t|S_{t-1})\beta(A_t|O_t)P(S_t|S_{t-1},A_t)d_R(R_t|S_{t-1},A_t,S_t)}\\ &= \prod_{t=0}^{T-1}\frac{\pi(A_t|O_t)}{\beta(A_t|O_t)} = \rho(\mathcal{T}).\end{aligned}$$

Combining with the previous result, we therefore have that $J(\pi) = \mathrm{E}_{\mathcal{T}\sim\beta}[\rho(\mathcal{T})g(\mathcal{T})]$.

3. Next, as neither $\rho(\mathcal{T})$ nor $g(\mathcal{T})$ depend on the states, we apply the law of total probability to sum out the states and get: $\mathrm{E}_{\mathcal{T}\sim\beta}[\rho(\mathcal{T})g(\mathcal{T})] = \mathrm{E}_{H\sim\beta}[\rho(H)g(H)]$. Combining with the previous result, we therefore have that $J(\pi) = \mathrm{E}_{H\sim\beta}[\rho(H)g(H)]$.
4. Finally, we bring these results – and another application of the law of total probability – together to show (writing $H^n \sim \beta^n$ for the more precise $(H_i)_{i=1}^n \sim (\beta_i)_{i=1}^n$):

$$\mathrm{E}_{H^n\sim\beta^n}[\mathrm{IS}] = \mathrm{E}_{H^n\sim\beta^n}\left[\frac{1}{n}\sum_{i=1}^{n}(\rho_i(H_i)g(H_i))\right] = \frac{1}{n}\sum_{i=1}^{n}\mathrm{E}_{H^n\sim\beta^n}[\rho_i(H_i)g(H_i)]$$
$$= \frac{1}{n}\sum_{i=1}^{n}\mathrm{E}_{H_i\sim\beta_i}[\rho_i(H_i)g(H_i)] = \frac{1}{n}\sum_{i=1}^{n}J_{\mathcal{T}}(\pi) = J_{\mathcal{T}}(\pi).$$

Our goal is to mechanize this proof, establishing the result for hybrid (mixed discrete and continuous) distributions.

2.2 Radon-Nikodym Derivatives

Radon-Nikodym derivatives generalize the concept of *probability density functions* (PDFs) and *probability mass functions* (PMFs) and allow us to transform one measure space into another via the density operator above. They capture the idea of a point-wise density ratio between two measure spaces, allowing a change of measure by integration.

Consider measure spaces (X, Σ, μ) and (X, Σ, ν), and function f such that $\nu = f * \mu$, where f is called a *Radon-Nikodym derivative*. If we take (X, Σ, μ) to be the canonical uniform measure space on the real line and f to be the PDF of a normal distribution, then $\nu(s) = (f * \mu)(s)$ would be the probability that a random number drawn from a normal distribution is in s, and $\nu((-\infty, x))$ would be the normal *cumulative distribution function* (CDF) at x.

In this context discrete, continuous, and hybrid measures (e.g., probability measures, though this discussion applies to all measures) refer to characteristics of ν, such as whether it characterizes a distribution that has point masses (like a Bernoulli distribution), density (like a normal distribution), or both. In cases where we do not have access to ν (such as a π-weighted measure space for trajectories), we might use a different measure μ together with a correction term that "re-weights" the samples from μ. This re-weighting term is the Radon-Nikodym derivative.

We want our proofs to capture discrete, continuous, and hybrid ν. PDFs cannot capture discrete distributions, PMFs cannot capture continuous distributions, and neither can capture hybrid distributions. Taking a measure-theoretic approach using Radon-Nikodym derivatives allows us to capture all of these cases. Though this could also be achieved using PDFs and Dirac delta functions, Dirac delta functions are *generalized functions* [29] not functions, are not currently in the HOL4 libraries, and present challenges for completing a formal proof [33].

3 Proofs

We now present salient aspects of the mechanized proofs, covering Hoeffding's inequality, product spaces and isomorphisms, trajectories and histories, importance sampling, OPE, and lastly confidence intervals on OPE.

3.1 Hoeffding's Inequality

Much of what is needed to prove Hoeffding's inequality is already in HOL4's library. We do need a theorem about the expectation of the product of independent random variables:

▶ **Lemma 1** (Product of Independent Variables).

$$\vdash \mathsf{prob\text{-}space}\ p \wedge (\forall\, i.\ i < n \Rightarrow \mathsf{real\text{-}random\text{-}variable}\ X_i\ p) \wedge \mathsf{independent}\ p\ X\ (\mathsf{count}\ n) \wedge (\forall\, i.\ i < n \Rightarrow \mathsf{integrable}\ p\ X_i\ ) \Rightarrow \mathbb{E}_p[(\lambda\, x.\ \prod_{i<n}(X_i\ x))] = \prod_{i<n} \mathbb{E}_p[X_i\ ].$$

In addition to the standard presentation of Hoeffding's lemma, we prove a generalized version (not centered at 0):

▶ **Lemma 2** (Hoeffding's lemma).

$$\vdash \mathsf{prob\text{-}space}\ p \wedge \mathsf{real\text{-}random\text{-}variable}\ X\ p \wedge \mathbb{E}_p[X] = 0 \wedge a \le 0 \wedge 0 \le b \wedge (\mathsf{AE}\ x :: p.\ a \le X\ x \wedge X\ x \le b) \Rightarrow \mathbb{E}_p[(\lambda\, x.\ \mathsf{exp}\ (c \cdot X\ x))] \le \mathsf{exp}\ (c^2 \cdot (b - a)^2\ /\ 8);\ \text{and}$$

$$\vdash \mathsf{prob\text{-}space}\ p \wedge \mathsf{real\text{-}random\text{-}variable}\ X\ p \wedge (\mathsf{AE}\ x :: p.\ a \le X\ x \wedge X\ x \le b) \Rightarrow \mathbb{E}_p[(\lambda\, x.\ \mathsf{exp}\ (c \cdot (X\ x - \mathbb{E}_p[X])))] \le \mathsf{exp}\ (c^2 \cdot (b - a)^2\ /\ 8).$$

Hoeffding's inequality then follows with some algebra:

▶ **Lemma 3** (Hoeffding's inequality [14]). *Given p a probability space, n a positive natural number, and $X_{i<n}$, n independent random variables drawn from p that almost surely lie in their respective intervals $[a_i, b_i]$, let $S_n\ x = \sum_{i<n}(X_i\ x)$. We then have that the probability that the sum of the X_i minus their expectation exceeds some fixed t is bounded above by an expression in t and the a_i and b_i:*

$$\vdash \mathrm{P}_p[\{\ x \mid t \le S_n\ x - \mathbb{E}_p[(\lambda\, x.\ S_n\ x)]\ \}] \le \mathsf{exp}\ (-2 \cdot t^2\ /\ \textstyle\sum_{i<n}(b_i - a_i)^2)$$

We also prove a corollary, for positive δ, that is more helpful in constructing confidence intervals:

$$\vdash 1 - \delta \le \mathrm{P}_p[\{\ x \mid n^{-1} \cdot S_n\ x - \mathsf{sqrt}\ (\ln \delta^{-1} \cdot \textstyle\sum_{i<n}(b_i - a_i)^2\ /\ (2 \cdot n^2)) \le \mathbb{E}_p[(\lambda\, y.\ n^{-1} \cdot S_n\ y)]\ \}].$$

The proof is straightforward, directly paralleling one found in Wikipedia:[3]

$$\begin{aligned}
&\mathrm{P}\left(S_n - \mathrm{E}\left[S_n\right] \ge t\right) && \text{where } S_n = \sum_{i=1}^{n} X_i \\
&= \mathrm{P}\left(\exp(s(S_n - \mathrm{E}\left[S_n\right])) \ge \exp(st)\right) && \text{for } s > 0, \text{ by monotonicity of exp} \\
&\le \exp(-st)\,\mathrm{E}\left[\exp(s(S_n - \mathrm{E}\left[S_n\right]))\right] && \text{by Markov's inequality} \\
&= \exp(-st)\prod_{i=1}^{n}\mathrm{E}\left[\exp(s(X_i - \mathrm{E}\left[X_i\right]))\right] && \text{by algebraic manipulation} \\
&\le \exp(-st)\prod_{i=1}^{n}\exp\left(\frac{s^2(b_i-a_i)^2}{8}\right) && \text{by Hoeffding's lemma}
\end{aligned}$$

[3] See `https://en.wikipedia.org/wiki/Hoeffding%27s_inequality`, as of February, 2022. The Wikipedia formulation assumes the probability space for the X_i without naming it. In HOL4 it is an explicit argument p, and x is a point in that space.

$$= \exp\left(-st + \tfrac{1}{8}s^2 \sum_{i=1}^{n}(b_i - a_i)^2\right) \qquad \text{by algebraic manipulation}$$

$$= \exp\left(-\frac{2t^2}{\sum_{i=1}^{n}(b_i - a_i)^2}\right) \qquad \text{by setting } s = \frac{4t}{\sum_{i=1}^{n}(b_i - a_i)^2}$$

The corollary can be obtained by setting $t = \sqrt{\ln \delta^{-1} \sum_{i=1}^{n}(b_i - a_i)^2/2}$ and dividing everything by n.

3.2 Product Spaces

Our proofs use measure spaces that are products of arbitrary (finite) numbers of measure spaces. A technical limitation of HOL4's type system is that it does not support arbitrary n-tuples of types. To model such things requires using an indexing function of type num $\to \alpha$ where α subsumes the types of all the dimensions. The existing Martingale theory package provides theorems about pairwise products of measure spaces. We recapitulate that approach in the inductive step of building n-fold products.

▶ **Definition 4** (pi-m-space). *The space of a product measure space is the product of the spaces of the component measure spaces. We use a subsidiary definition* updated-at*, which defines n-fold product sets. The term $f(\!|n \mapsto e|\!)$ denotes a function that is everywhere the same as f except at n, where it maps to e.*

pi-m-space 0 mn $\stackrel{\text{def}}{=}$ $\{\ (\lambda\, i.\ \text{ARB})\ \}$
pi-m-space (SUC n) mn $\stackrel{\text{def}}{=}$ updated-at n (pi-m-space n mn) (m-space $(mn\ n)$), *where*

updated-at n fs s $\stackrel{\text{def}}{=}$ $\{\ f(\!|n \mapsto e|\!) \mid f \in fs \wedge e \in s\ \}$.

▶ **Definition 5** (pi-measurable-sets). *The measurable sets of a product measure space are those of the smallest sigma algebra formed by the rectangular sets of the component measure spaces. We use a subsidiary definition* pi-prod-sets*, which defines the n-fold rectangular sets.* pi-sig-alg *is a packaging of the space and measurable sets.*

pi-measurable-sets 0 mn $\stackrel{\text{def}}{=}$ POW $\{\ (\lambda\, i.\ \text{ARB})\ \}$
pi-measurable-sets (SUC n) mn $\stackrel{\text{def}}{=}$
 subsets
  (sigma (pi-m-space (SUC n) mn)
   $\{\ \{\ f(\!|n \mapsto e|\!) \mid f \in fs \wedge e \in s\ \} \mid$
   $fs \in$ pi-measurable-sets n $mn \wedge s \in$ measurable-sets $(mn\ n)\ \}$);

pi-prod-sets n $fsts$ sts $\stackrel{\text{def}}{=}$ $\{$ updated-at n fs $s \mid fs \in fsts \wedge s \in sts\ \}$; and

pi-sig-alg n mn $\stackrel{\text{def}}{=}$ (pi-m-space n mn, pi-measurable-sets n mn).

▶ **Definition 6** (pi-measure). *The measure of a set in a product measure space is obtained by integrating over an indicator function of that set.* pi-measure-space *is a packaging of all components of the measure space.*

$$\begin{aligned}&\text{pi-measure } 0\ mn \stackrel{\text{def}}{=} (\lambda\, fs.\ \mathbb{1}\ fs\ (\lambda\, i.\ \text{ARB}))\\&\text{pi-measure } (\text{SUC } n)\ mn \stackrel{\text{def}}{=}\\&\quad (\lambda\, fs.\ \int^{+}\ (mn\ n)\ (\lambda\, e.\ \int^{+}\ (\text{pi-measure-space } n\ mn)\ (\lambda\, f.\ \mathbb{1}\ fs\ f(\!|n\ \mapsto\ e|\!)))); \text{ and}\\&\text{pi-measure-space } n\ mn \stackrel{\text{def}}{=}\\&\quad (\text{pi-m-space } n\ mn, \text{pi-measurable-sets } n\ mn, \text{pi-measure } n\ mn).\end{aligned}$$

3.3 Isomorphisms

We show that these product spaces are indeed measure spaces. The proof uses recursion to show an $(n+1)$-way product space of sigma finite measure spaces is a sigma finite measure space because it is "equivalent to" a 2-way product of an n-way product and the remaining space. Firstly, we address the 2-way product being a sigma finite measure space:

▶ **Theorem 7** (2-Way Product is a Sigma Finite Measure Space). *A product of 2 sigma finite measure spaces is a sigma finite measure space.*

$$\begin{aligned}&\vdash \text{sigma-finite-measure-space } m_1\ \wedge\ \text{sigma-finite-measure-space } m_2\ \Rightarrow\\&\quad \text{sigma-finite-measure-space } (m_1\ \times\ m_2).\end{aligned}$$

We also need results formalizing what it means for a measure space to be "equivalent to" another one. Specifically, two measure spaces are isomorphic if there is a "measure preserving" function between them, along the lines of Halmos [13, p. 164]:

▶ **Definition 8** (Isomorphic Measure Spaces).

$$m_0 \cong m_1 \stackrel{\text{def}}{=} \exists f.\ f\ \in\ \text{measure-preserving } m_0\ m_1$$

where

$$\begin{aligned}&\text{measure-preserving } m_0\ m_1 \stackrel{\text{def}}{=}\\&\quad \{\, f\ |\\&\quad f\ \in\ \text{measurability-preserving } (\text{sig-alg } m_0)\ (\text{sig-alg } m_1)\ \wedge\\&\quad \forall\, s.\ s\ \in\ \text{measurable-sets } m_0\ \Rightarrow\ \text{measure } m_0\ s\ =\ \text{measure } m_1\ (\text{IMAGE } f\ s)\ \}\ ;\ \textit{and}\\&\text{measurability-preserving } a\ b \stackrel{\text{def}}{=}\\&\quad \{\, f\ |\\&\quad \sigma\text{-algebra } a\ \wedge\ \sigma\text{-algebra } b\ \wedge\ \text{BIJ } f\ (\text{space } a)\ (\text{space } b)\ \wedge\\&\quad (\forall\, s.\ s\ \in\ \text{subsets } a\ \Rightarrow\ \text{IMAGE } f\ s\ \in\ \text{subsets } b)\ \wedge\\&\quad \forall\, s.\ s\ \in\ \text{subsets } b\ \Rightarrow\ f^{-1}\ s\ \cap\ \text{space } a\ \in\ \text{subsets } a\ \}\,.\end{aligned}$$

These definitions allow us to formalize the utility of isomorphisms:

▶ **Theorem 9** (Isomorphisms Preserve Sigma Finiteness). *Isomorphism to a sigma finite measure space implies being a sigma finite measure space*

$$\vdash \text{sigma-finite-measure-space } m_0\ \wedge\ m_0 \cong m_1\ \Rightarrow\ \text{sigma-finite-measure-space } m_1.$$

Finally, we combine Theorems 7 and 9 with the existence of a measure-preserving function:

▶ **Theorem 10** (n-Way Product is a Sigma Finite Measure Space). *A product of n sigma finite measure spaces is a sigma finite measure space.*

$$\vdash (\forall\, i.\ i < n \Rightarrow \mathsf{sigma\text{-}finite\text{-}measure\text{-}space}\ (mn\ i)) \Rightarrow \mathsf{sigma\text{-}finite\text{-}measure\text{-}space}\ (\mathsf{pi\text{-}measure\text{-}space}\ n\ mn);\ \textit{because}$$

$$\begin{aligned}\vdash\ & \mathsf{sigma\text{-}finite\text{-}measure\text{-}space}\ (\mathsf{pi\text{-}measure\text{-}space}\ n\ mn)\ \wedge \\ & \mathsf{sigma\text{-}finite\text{-}measure\text{-}space}\ (mn\ n)\ \Rightarrow \\ & (\lambda\,(f, e).\ f(\!|n \mapsto e|\!))\ \in \\ & \mathsf{measure\text{-}preserving}\ (\mathsf{pi\text{-}measure\text{-}space}\ n\ mn\ \times\ mn\ n) \\ & \quad (\mathsf{pi\text{-}measure\text{-}space}\ (\mathsf{SUC}\ n)\ mn).\end{aligned}$$

This approach appears again when working with trajectory and history measure spaces.

3.4 Variable Name Conventions and Standard Assumptions

In our model of OPE, we model sets of states, observations, actions, and rewards. At the very least, we need sigma algebras of these, and as the later distributions will need base measure spaces, we need full measure spaces for each of these. Below, these are respectively denoted *ms*, *mo*, *ma*, *mr*. The underlying type variables we use for these are σ, ω, α, ρ respectively (for less general results, ρ, the reward type, is usually of type extreal). For convenience, these standard measure parameters are omitted from argument/parameter lists henceforth, as are assumptions that these spaces are either measure spaces or sigma finite measure spaces.

Individual states are typically s and s' (s' being the later state); observations are o, actions are a, r is reward, and h is history or trajectory (they seldom appear simultaneously). The length of trajectories or histories is n or T, while the number of rows in the database is N.

3.5 Trajectories and Histories as Types

We now show our development of trajectories and histories (described in Section 2.1) as types in HOL4. The respective types are defined:

▶ **Definition 11** (Histories and Trajectories as HOL Types).

$$(\alpha,\ \rho,\ \omega)\ \mathrm{hist}\ =\ \mathsf{hnil}\ |\ \mathsf{hcons}\ ((\alpha,\ \rho,\ \omega)\ \mathrm{hist})\ \omega\ \alpha\ \rho$$

$$(\alpha,\ \rho,\ \sigma,\ \omega)\ \mathrm{traj}\ =\ \mathsf{init}\ \sigma\ |\ \mathsf{tcons}\ ((\alpha,\ \rho,\ \sigma,\ \omega)\ \mathrm{traj})\ \omega\ \alpha\ \sigma\ \rho$$

A history is isomorphic to a list of observation, action, and reward triples; these are drawn from the type variables ω, α, and ρ respectively. The order of arguments in our definition is deliberate: we consider the non-recursive arguments to the "cons" functions to be at the end (rather than head) of the history. A trajectory adds a required initial state of type σ, and then adds a state between each action and reward of a history. The function t-hist *extracts the state-less triples from a trajectory, giving a history. We write $|t|$ to record the number of steps in a trajectory (i.e., $|\mathsf{init}\ s| = 0$), and* t-st *t is the function that returns the final state of trajectory t.*

The definition and machinery of history measure spaces directly parallels, and was developed after, that of trajectories. It suffices to speak only of trajectories here.

Originally, we considered measure spaces of all trajectories of all lengths. This was abandoned for two reasons. First, it was unnecessary. For any database of histories (which contains a finite number of histories), there is a longest trajectory among those from which

the histories are derived. All other histories/trajectories can be extended to the same length via "do nothing" actions. This is why a maximum number of steps is taken in the literature as described earlier. Moreover, only fixed-length PDF-weighted trajectory measure spaces can form probability spaces, which is necessary later on.

Even though we abandoned the use of different trajectory lengths, we pursued it long enough that it inspired trajectory spaces and measurable sets to be defined in a not-directly-recursive manner (unlike the n-way product space). The development parallels that of product spaces directly, as opposed to only within an inductive step.

▶ **Definition 12** (traj-m-space-n). *A product measure space's space is a cross product of the component measure spaces' spaces. We mirror that with* traj-cross*, which takes a trajectory of sets and returns a set of trajectories, each element of which falls component-wise within the input. In order to get a trajectory of spaces of the desired length, we make a helper function named* traj-n-gen*.*

$$\begin{aligned}
&\text{traj-m-space-n } n\ ms\ mo\ ma\ mr \stackrel{\text{def}}{=} \\
&\quad \text{traj-cross } (\text{traj-n-gen } n\ (\text{m-space } ms)\ (\text{m-space } mo)\ (\text{m-space } ma)\ (\text{m-space } mr));
\end{aligned}$$

$$\begin{aligned}
&\text{traj-cross } (\text{init } ss)\ (\text{init } s) \stackrel{\text{def}}{=} s \in ss \\
&\text{traj-cross } (\text{init } ss)\ (\text{tcons } h\ w\ a\ s\ r) \stackrel{\text{def}}{=} \mathsf{F} \\
&\text{traj-cross } (\text{tcons } hs\ ws\ as\ ss\ rs)\ (\text{init } s) \stackrel{\text{def}}{=} \mathsf{F} \\
&\text{traj-cross } (\text{tcons } hs\ ws\ as\ ss\ rs)\ (\text{tcons } h\ w\ a\ s\ r) \stackrel{\text{def}}{=} \\
&\quad w \in ws \wedge a \in as \wedge s \in ss \wedge r \in rs \wedge \text{traj-cross } hs\ h; \text{ and}
\end{aligned}$$

$$\begin{aligned}
&\text{traj-n-gen } 0\ sg\ og\ ag\ rg \stackrel{\text{def}}{=} \text{init } sg \\
&\text{traj-n-gen } (\text{SUC } n)\ sg\ og\ ag\ rg \stackrel{\text{def}}{=} \text{tcons } (\text{traj-n-gen } n\ sg\ og\ ag\ rg)\ og\ ag\ sg\ rg.
\end{aligned}$$

▶ **Definition 13** (traj-measurable-sets-n). *A product measure space's measurable sets form a sigma algebra containing all cross products (the "rectangular" sets) of the component measure spaces' measurable sets. We mirror that with* traj-rect-sets-n*, which takes a trajectory of sets of measurable sets, to a set of trajectories of measurable sets, to a set of rectangular sets of trajectories.* traj-sig-alg-n *is a pairing of the space and measurable sets.*

$$\begin{aligned}
&\text{traj-measurable-sets-n } n \stackrel{\text{def}}{=} \\
&\quad \text{subsets } (\text{sigma } (\text{traj-m-space-n } n)\ (\text{traj-rect-sets-n } n));
\end{aligned}$$

$$\begin{aligned}
&\text{traj-rect-sets-n } n \stackrel{\text{def}}{=} \\
&\quad \text{IMAGE traj-cross} \\
&\qquad (\text{traj-cross} \\
&\qquad\quad (\text{traj-n-gen } n\ (\text{measurable-sets } ms)\ (\text{measurable-sets } mo) \\
&\qquad\qquad (\text{measurable-sets } ma)\ (\text{measurable-sets } mr))); \text{ and}
\end{aligned}$$

$$\text{traj-sig-alg-n } n \stackrel{\text{def}}{=} (\text{traj-m-space-n } n, \text{traj-measurable-sets-n } n).$$

▶ **Definition 14** (traj-measure-n). *The measure of a set is obtained by integrating over an indicator function of that set.* traj-measure-space-n *is a pairing of all components of the measure space:*

$$\textsf{traj-measure-n}\ 0 \stackrel{\text{def}}{=} (\lambda\, hs.\ \int^+ ms\ (\lambda\, s.\ \mathbb{1}\ hs\ (\textsf{init}\ s)))$$
$$\textsf{traj-measure-n}\ (\textsf{SUC}\ n) \stackrel{\text{def}}{=}$$
$$(\lambda\, hs.\ \int^+ (\textsf{traj-measure-space-n}\ n)\ (\lambda\, h.\ \int^+ mo\ (\lambda\, o.\ \int^+ ma\ (\lambda\, a.\ \int^+ ms\ (\lambda\, s.\ \int^+ mr\ (\lambda\, r.\ \mathbb{1}\ hs\ (\textsf{tcons}\ h\ o\ a\ s\ r)))))))); \text{ and}$$
$$\textsf{traj-measure-space-n}\ n \stackrel{\text{def}}{=} (\textsf{traj-m-space-n}\ n, \textsf{traj-measurable-sets-n}\ n, \textsf{traj-measure-n}\ n).$$

Many steps in later proofs require these to be measure spaces, which we prove inductively by showing they form sigma finite measure spaces:

▶ **Theorem 15** (History and Trajectory Measure Spaces)**.**

$\vdash \textsf{sigma-finite-measure-space}\ (\textsf{traj-measure-space-n}\ n)$*; because*
$\vdash \textsf{traj-measure-space-n}\ 0 \cong ms$*; and*
$\vdash \textsf{traj-measure-space-n}\ (\textsf{SUC}\ n) \cong \textsf{traj-measure-space-n}\ n \times mo \times ma \times ms \times mr.$
$\vdash \textsf{sigma-finite-measure-space}\ (\textsf{hist-space}\ n)$*; because*
$\vdash \textsf{hist-space}\ (\textsf{SUC}\ n) \cong \textsf{hist-space}\ n \times mo \times ma \times mr.$

3.6 Preconditions and Useful Functions

Before developing the main OPE results, the mechanization identifies our standard preconditions on d_0 (initial state distribution), Ω (the observation distribution), β (current policy), π (new policy), P (transition distribution), and d_R (reward distribution), grouping them together in a predicate valid-dist-gen-funs. The conditions are about as minimal as one could hope: the distribution functions need to be positive, non-infinite, measurable, and integrate to 1 (form a probability space when used for making a density space). We omit the full definition, but assume it in all contexts mentioning our various distribution functions hereafter.

▶ **Definition 16** (Trajectory PDF, Importance Ratio, and Return)**.** *Using those preconditions, we define a number of functions, specifically the return, importance ratio, and PDF:*

$$\textsf{traj-pdf}\ d_0\ P\ \Omega\ d_R\ \beta\ (\textsf{init}\ s) \stackrel{\text{def}}{=} d_0\ s$$
$$\textsf{traj-pdf}\ d_0\ P\ \Omega\ d_R\ \beta\ (\textsf{tcons}\ h\ w\ a\ s\ r) \stackrel{\text{def}}{=}$$
$$\textsf{traj-pdf}\ d_0\ P\ \Omega\ d_R\ \beta\ h \cdot \Omega\ (\textsf{t-st}\ h)\ w \cdot \beta\ w\ a \cdot P\ (\textsf{t-st}\ h)\ a\ s \cdot d_R\ (\textsf{t-st}\ h)\ a\ s\ r;$$

$$\textsf{importance-ratio}\ \pi\ \beta\ (\textsf{init}\ s) \stackrel{\text{def}}{=} 1$$
$$\textsf{importance-ratio}\ \pi\ \beta\ (\textsf{tcons}\ h\ w\ a\ s\ r) \stackrel{\text{def}}{=} \textsf{importance-ratio}\ \pi\ \beta\ h \cdot \pi\ w\ a \cdot (\beta\ w\ a)^{-1}\text{; and}$$

$$\textsf{traj-return}\ \gamma\ (\textsf{init}\ s) \stackrel{\text{def}}{=} 0$$
$$\textsf{traj-return}\ \gamma\ (\textsf{tcons}\ h\ w\ a\ s\ r) \stackrel{\text{def}}{=} \textsf{traj-return}\ \gamma\ h\ +\ \gamma^{|h|} \cdot r.$$

There are similarly defined analogous functions for histories: hist-pdf, h-importance-ratio, and hist-return.

▶ **Definition 17** (History PDF). hist-pdf *is particularly interesting in that it requires a helper function that serves as a PDF of history and final state, where the non-final states are integrated away recursively.*

$$\text{hist-pdf } ms\ d_0\ P\ \Omega\ d_R\ \beta\ h \stackrel{\text{def}}{=} \int^+ ms\ (\text{hist-lst-pdf } ms\ d_0\ P\ \Omega\ d_R\ \beta\ h); \text{ and}$$

$$\text{hist-lst-pdf } ms\ d_0\ P\ \Omega\ d_R\ \beta\ \text{hnil}\ s' \stackrel{\text{def}}{=} d_0\ s'$$
$$\text{hist-lst-pdf } ms\ d_0\ P\ \Omega\ d_R\ \beta\ (\text{hcons } h\ o\ a\ r)\ s' \stackrel{\text{def}}{=}$$
$$\beta\ o\ a \cdot \int^+ ms\ (\lambda\, s.\ \text{hist-lst-pdf } ms\ d_0\ P\ \Omega\ d_R\ \beta\ h\ s \cdot \Omega\ s\ o \cdot P\ s\ a\ s' \cdot d_R\ s\ a\ s'\ r).$$

Many steps in later proofs require these (and their history analogs) to be measurable:

$$\vdash \text{traj-pdf } d_0\ P\ \Omega\ d_R\ \beta \in \text{Borel-measurable } (\text{traj-sig-alg-n } n);$$

$$\vdash (\forall\, w\ a.\ w \in \text{m-space } mo \wedge a \in \text{m-space } ma \wedge \beta\ w\ a = 0 \Rightarrow \pi\ w\ a = 0) \Rightarrow \text{importance-ratio } \pi\ \beta \in \text{Borel-measurable } (\text{traj-sig-alg-n } n); \text{ and}$$

$$\vdash \text{sig-alg } mr = \text{Borel} \Rightarrow \text{traj-return } \gamma \in \text{Borel-measurable } (\text{traj-sig-alg-n } n).$$

3.7 Importance Sampling

We now start to address OPE as described in Section 2.1. The first step is to shift the density measure space from the π PDF to the β PDF, by showing that the PDF ratio is a Radon-Nikodym derivative from one space to the other:

$$\vdash (\forall\, w\ a.\ w \in \text{m-space } mo \wedge a \in \text{m-space } ma \wedge \beta\ w\ a = 0 \Rightarrow \pi\ w\ a = 0) \wedge f \in \text{Borel-measurable } (\text{traj-sig-alg-n } n) \Rightarrow \int (\text{density } (\text{traj-measure-space-n } n)\ (\text{traj-pdf } d_0\ P\ \Omega\ d_R\ \pi))\ f = \int (\text{density } (\text{traj-measure-space-n } n)\ (\text{traj-pdf } d_0\ P\ \Omega\ d_R\ \beta))\ (\lambda\, h.\ \text{traj-pdf } d_0\ P\ \Omega\ d_R\ \pi\ h \cdot (\text{traj-pdf } d_0\ P\ \Omega\ d_R\ \beta\ h)^{-1} \cdot f\ h).$$

We next replace the ratio of PDFs with the importance ratio, *via* reasonably straightforward algebra:

$$\vdash h \in \text{traj-m-space-n } n \wedge \text{traj-pdf } d_0\ P\ \Omega\ d_R\ \beta\ h \neq 0 \Rightarrow \text{traj-pdf } d_0\ P\ \Omega\ d_R\ \pi\ h \cdot (\text{traj-pdf } d_0\ P\ \Omega\ d_R\ \beta\ h)^{-1} = \text{importance-ratio } \pi\ \beta\ h.$$

▶ **Theorem 18** (Importance Ratio). *We combine our last two results to allow us to calculate an expectation (integral, here) in π-weighted trajectory space given trajectories in β-weighted trajectory space:*

$$\vdash (\forall\, w\ a.\ w \in \text{m-space } mo \wedge a \in \text{m-space } ma \wedge \beta\ w\ a = 0 \Rightarrow \pi\ w\ a = 0) \wedge f \in \text{Borel-measurable } (\text{traj-sig-alg-n } n) \Rightarrow \int (\text{density } (\text{traj-measure-space-n } n)\ (\text{traj-pdf } d_0\ P\ \Omega\ d_R\ \pi))\ f = \int (\text{density } (\text{traj-measure-space-n } n)\ (\text{traj-pdf } d_0\ P\ \Omega\ d_R\ \beta))\ (\lambda\, h.\ \text{importance-ratio } \pi\ \beta\ h \cdot f\ h).$$

3.8 Off-Policy Evaluation

The next step is to shift from the β trajectory PDF to the β history PDF, justifying the use of the empirical estimator. To that end, after some involved changes in order of integration, we are able to show:

▶ **Theorem 19** (Trajectories to Histories). *Integrals in the trajectory measure space (assumed sigma finite), can be recast as integrals in the history space:*

$$\begin{aligned}&\vdash (\forall\, x.\ x \in \text{hist-m-space-n}\ n \Rightarrow 0 \le f\ x) \wedge \\ &\quad f \in \text{Borel-measurable}\ (\text{hist-sig-alg-n}\ n) \Rightarrow \\ &\quad \int^{+} (\text{density}\ (\text{traj-measure-space-n}\ n)\ (\text{traj-pdf}\ d_0\ P\ \Omega\ d_R\ \beta))\ (f \circ \text{t-hist}) = \\ &\quad \int^{+} (\text{density}\ (\text{hist-space}\ n)\ (\text{hist-pdf}\ ms\ d_0\ P\ \Omega\ d_R\ \beta))\ f.\end{aligned}$$

A useful corollary – used in later proofs – follows:

▶ **Corollary 20.** *Since the PDF-weighted trajectory space is a probability space, setting the positive function in the earlier result to be the constant 1, we derive that the PDF-weighted history space is also a probability space:*

$$\begin{aligned}&\vdash \text{prob-space}\ (\text{density}\ (\text{traj-measure-space-n}\ n)\ (\text{traj-pdf}\ d_0\ P\ \Omega\ d_R\ \beta)); \text{ and thus} \\ &\vdash \text{prob-space}\ (\text{density}\ (\text{hist-space}\ n)\ (\text{hist-pdf}\ ms\ d_0\ P\ \Omega\ d_R\ \beta)).\end{aligned}$$

Finally, we combine Theorems 18 and 19:

▶ **Theorem 21.** *It is possible to estimate the expected return in π-weighted trajectory space by appeal to the empirical return of β-weighted history space:*

$$\begin{aligned}&\vdash (\forall\, w\ a.\ w \in \text{m-space}\ mo \wedge a \in \text{m-space}\ ma \wedge \beta\ w\ a = 0 \Rightarrow \pi\ w\ a = 0) \wedge \\ &\quad f \in \text{Borel-measurable}\ (\text{hist-sig-alg-n}\ n) \Rightarrow \\ &\quad \int (\text{density}\ (\text{traj-measure-space-n}\ n)\ (\text{traj-pdf}\ d_0\ P\ \Omega\ d_R\ \pi))\ (f \circ \text{t-hist}) = \\ &\quad \int (\text{density}\ (\text{hist-space}\ n)\ (\text{hist-pdf}\ ms\ d_0\ P\ \Omega\ d_R\ \beta)) \\ &\qquad (\lambda\, h.\ \text{h-importance-ratio}\ \pi\ \beta\ h \cdot f\ h).\end{aligned}$$

3.9 Database Estimate

To complete OPE, we go from an empirical estimate based on a single history, to one based on an average over histories. We first prove a lemma to help go from product space back to the individual spaces:

▶ **Theorem 22.** *An integral (expectation) over a sum (thus average) in product space can be reduced to a sum (or average) of integrals over individual spaces:*

$$\begin{aligned}&\vdash (\forall\, i.\ i < n \Rightarrow \text{prob-space}\ mn_i\ ) \wedge (\forall\, i.\ i < n \Rightarrow \text{integrable}\ mn_i\ f_i\ ) \Rightarrow \\ &\quad \int (\text{pi-measure-space}_n\ mn)\ (\lambda\, x.\ \textstyle\sum_{i<n} (f_i\ x_i\ )) = \sum_{i<n} (\int\ mn_i\ f_i\ ).\end{aligned}$$

We use that to generalize Theorem 21:

▶ **Theorem 23.** *It is possible to estimate the expected return in π-weighted trajectory space by appeal to an average of empirical returns in β_i-weighted history spaces. Let*

$$\begin{aligned}p &= \text{pi-measure-space}_n\ (\lambda\, i.\ \text{density}\ (\text{hist-space}\ T)\ (\text{hist-pdf}\ ms\ d_0\ P\ Z\ d_R\ \beta_i\ )) \\ \tau &= \text{density}\ (\text{traj-measure-space-n}\ T)\ (\text{traj-pdf}\ d_0\ P\ Z\ d_R\ phi)\end{aligned}$$

and assume the various implicit measure spaces (ma, mo, etc.*) are sigma finite measure spaces; that n is strictly positive, and that π is zero whenever any of the β_i are, i.e.:*

$$\forall\, i\ o\ a.\ i < n \wedge o \in \text{m-space}\ mo \wedge a \in \text{m-space}\ ma \wedge \beta_i\ o\ a = 0 \Rightarrow \pi\ o\ a = 0$$

then:

$$\begin{aligned}&\vdash \text{integrable}\ (\text{density}\ (\text{hist-space}\ nT)\ (\text{hist-pdf}\ ms\ d_0\ P\ \Omega\ d_R\ \pi))\ f \Rightarrow \\ &\quad \int\ p\ (\lambda\, D.\ n^{-1} \cdot \textstyle\sum_{i<n} (\text{h-importance-ratio}\ \pi\ \beta_i\ D_i \cdot f\ D_i)) = \int\ \tau\ (f \circ \text{t-hist}).\end{aligned}$$

3.10 Confidence Interval

For brevity, we define the database-based estimate of the return as

$$\begin{array}{l}\mathsf{data\text{-}return}_n\ \pi\ \beta\ \gamma\ D \stackrel{\mathsf{def}}{=} \\ \quad n^{-1} \cdot \sum_{i<n} (\mathsf{h\text{-}importance\text{-}ratio}\ \pi\ \beta_i\ D_i \cdot \mathsf{hist\text{-}return}\ \gamma\ D_i\ )\end{array}$$

At this point we have shown that OPE gives an unbiased estimate of the expected return of the new policy. However, it is even more useful to have a confidence interval around this expectation. We develop a confidence interval using Hoeffding's inequality (other concentration inequalities could be used here). Using Hoeffding's inequality requires a further precondition: bounding almost everywhere the importance ratio times the return. The final result is:

▶ **Theorem 24** (Unbiasedness of Off-Policy Evaluation on Return). *Let*

$$\begin{array}{lcl} S & = & \sum_{i<n} (UB_i - LB_i)^2 / (2 \cdot n^2) \\ p & = & \mathsf{pi\text{-}measure\text{-}space}_n\ (\lambda\, i.\ \mathsf{density}\ (\mathsf{hist\text{-}space}\ T)\ (\mathsf{hist\text{-}pdf}\ ms\ d_0\ P\ \Omega\ d_R\ \beta_i\ )) \\ \tau & = & \mathsf{density}\ (\mathsf{traj\text{-}measure\text{-}space\text{-}n}\ T)\ (\mathsf{traj\text{-}pdf}\ d_0\ P\ \Omega\ d_R\ \pi) \end{array}$$

and assume the various implicit measure spaces (ma, mo, etc.*) are sigma finite measure spaces; that n and δ are both greater than 0, and that π is zero whenever any of the β_i are, i.e.:*

$$\forall\, i\ o\ a.\ i < n \wedge o \in \mathsf{m\text{-}space}\ mo \wedge a \in \mathsf{m\text{-}space}\ ma \wedge \beta_i\ o\ a = 0 \Rightarrow \pi\ o\ a = 0$$

then:

$$\begin{array}{l}\vdash \mathsf{sig\text{-}alg}\ mr = \mathsf{Borel} \wedge \\ \quad (\forall\, h.\ h \in \mathsf{hist\text{-}m\text{-}space\text{-}n}\ T \Rightarrow Gmin \le \mathsf{hist\text{-}return}\ \gamma\ h \wedge \mathsf{hist\text{-}return}\ \gamma\ h \le Gmax) \wedge \\ \quad (\forall\, i.\ i < n \Rightarrow \\ \qquad \mathsf{AE}h :: \mathsf{density}\ (\mathsf{hist\text{-}space}\ T)\ (\mathsf{hist\text{-}pdf}\ ms\ d_0\ P\ \Omega\ d_R\ \beta_i). \\ \qquad\quad LB_i \le \mathsf{h\text{-}importance\text{-}ratio}\ \pi\ \beta_i\ h \cdot \mathsf{hist\text{-}return}\ \gamma\ h \wedge \\ \qquad\quad \mathsf{h\text{-}importance\text{-}ratio}\ \pi\ \beta_i\ h \cdot \mathsf{hist\text{-}return}\ \gamma\ h \le UB_i) \Rightarrow \\ \quad 1 - \delta \le \mathrm{P}_p[\{\ D \mid \mathsf{data\text{-}return}_n\ \pi\ \beta\ \gamma\ D - \mathsf{sqrt}\ (\ln \delta^{-1} \cdot S) \le \mathbb{E}_\tau[\mathsf{traj\text{-}return}\ \gamma]\ \}\ ].\end{array}$$

In other words, the one-sided confidence interval whose lower bound is $\sqrt{\ln \delta^{-1} \cdot S}$ less than our estimator captures the expected return of the new policy with probability at least $1 - \delta$. As such, if we have reason to believe – such as an upper-bounded confidence interval – that the expected return under the current policy is below the estimator confidence interval with probability at least $1 - \epsilon$, then the probability that the new policy's expected return is better is at least $(1 - \delta)(1 - \epsilon)$

3.11 The HOL4 Mechanization

The proof of Hoeffding's inequality took on the order of two person-months of effort and OPE took about three-person months. In terms of lines of HOL4 code, our running library of important and useful general-purpose results is currently around 4800 lines, Hoeffding's inequality and variants take another 500 lines, the development of product measure spaces and isomorphisms requires about 1000 lines, and the rest of the OPE development is approximately 3500 lines long, for a total of about 10,000 lines. As may often be the case when adding new theories, it is helpful to refine and enhance the theorem prover's formula simplification capabilities to reduce the number of small, tedious, steps – though many remain.

4 Conclusion and Future Work

While the expected return, $J(\pi)$, is the most common performance metric in RL, for some high-risk applications parameters of the return distribution other than the expected value can better characterize the risk of applying the policy π. For example, RL researchers have studied using coherent risk measures [31, 7, 36, 30, 28] like the *conditional value at risk* (CVaR) [1] of the return distribution. The extension of our results to this setting will require an additional step: though we have shown off-policy pointwise convergence to the CDF of returns under π, we must show *uniform* convergence following the hand-checked proof of Chandak et al. [6]. Uniform convergence of off-policy estimates to the CDF of returns under π would allow for estimates and confidence intervals for all parameters of the return distribution [6], including CVaR, variance, and quantiles.

Our main result uses Hoeffding's inequality to obtain a confidence interval. As concentration inequalities go, Hoeffding's inequality is easy to prove, but also notoriously loose. Thus another fruitful direction for future work is mechanizing proofs of other, tighter, concentration inequalites, such as Maurer and Pontil's empirical Bernstein bound [24], Anderson's inequality [2] using Massart's tight constants [23] for the Dvoretsky-Kiefer-Wolfowitz inequality [9], or an extremely tight confidence interval conjectured independently by multiple researchers [11, 22].

There are places where our development might be tightened by relaxing preconditions. For example, the constraint that the distribution generation functions form probability spaces isn't necessary for showing that any of the PDF functions are measurable, but we still use the precondition valid-dist-gen-funs for brevity of proof statement in lemmas that are ultimately used when valid-dist-gen-funs in its entirety is necessary. There are also cases where a definition takes in extraneous information. For example, pi-m-space and traj-m-space-n take entire measure spaces, but use only the spaces of those measure spaces.

Finally, our histories are isomorphic to lists, and trajectories are in turn lists with extra information between adjacent elements, making them what the HOL4 library would call finite *paths*. It would be appealing to develop more generalized theories about measure spaces based on these library notions, allowing our trajectory and history results to fall out as special cases. Moreover, the product measure space results could be expressed in terms of list measure spaces.

More generally, given our development of measure spaces over histories and trajectories, a variety of other results in RL might be mechanized.

It is reassuring that our proof of the soundness of OPE brought no big surprises. In particular, it was encouraging that the ultimate pre-conditions were just the measure spaces being sigma finite and the distribution generating functions being non-negative, finite, measurable, and integrating to 1. At the same time, it is satisfying to record OPE's generalization to hybrid distributions and to lay groundwork for several future directions.

References

1 Carlo Acerbi and Dirk Tasche. On the coherence of expected shortfall. *Journal of Banking & Finance*, 26(7):1487–1503, 2002.

2 B. D. Anderson and J. B. Moore. *Optimal control: linear quadratic methods*. Courier Corporation, 2007.

3 Alexander Bagnall and Gordon Stewart. Certifying the true error: Machine learning in Coq with verified generalization guarantees. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):2662–2669, July 2019. doi:10.1609/aaai.v33i01.33012662.

4 M. Bastani. Model-free intelligent diabetes management using machine learning. Master's thesis, Department of Computing Science, University of Alberta, 2014.

5 Sylvie Boldo, François Clément, Florian Faissole, Vincent Martin, and Micaela Mayero. A Coq formalization of Lebesgue integration of nonnegative functions. *Journal of Automated Reasoning*, November 2021. doi:10.1007/s10817-021-09612-0.

6 Yash Chandak, Scott Niekum, Bruno Castro da Silva, Erik Learned-Miller, Emma Brunskill, and Philip S. Thomas. Universal off-policy evaluation. In *Advances in Neural Information Processing Systems*, 2021.

7 Yinlam Chow and Mohammad Ghavamzadeh. Algorithms for CVaR optimization in MDPs. In *Advances in Neural Information Processing Systems*, pages 3509–3517, 2014.

8 Aaron R. Coble. *Anonymity, information, and machine-assisted proof*. PhD thesis, Computer Lab., University of Cambridge, July 2010. doi:10.48456/tr-785.

9 A. Dvoretzky, J. Kiefer, and J. Wolfowitz. Asymptotic minimax character of the sample distribution function and of the classical multinomial estimator. *Annals of Mathematical Statistics*, 27(3):642–669, 1956. doi:10.1214/aoms/1177728174.

10 Ahmad El Sallab, Mohammed Abdou, Etienne Perot, and Senthil Yogamani. Deep reinforcement learning framework for autonomous driving. *Electronic Imaging*, 2017(19):70–76, 2017.

11 Norbert Gaffke. Nonparametric one-sided testing for the mean and related extremum problems. *Mathematical Methods of Statistics*, 13(4):369–391, 2004.

12 Arthur Guez, Robert D. Vincent, Massimo Avoli, and Joelle Pineau. Adaptive treatment of epilepsy via batch-mode reinforcement learning. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*, pages 1671–1678, 2008.

13 Paul R. Halmos. *Measure Theory*. Springer-Verlag, New York, NY, 1950.

14 W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58:13–30, 1963.

15 Johannes Hölzl and Armin Heller. Three chapters of measure theory in Isabelle/HOL. In Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Proceedings of Interactive Theorem Proving*, pages 135–151. Springer, 2011.

16 Joe Hurd. Formal verification of probabilistic algorithms. Technical Report UCAM-CL-TR-566, University of Cambridge, Computer Laboratory, May 2003. doi:10.48456/tr-566.

17 Joe Hurd. Verification of the Miller–Rabin probabilistic primality test. *The Journal of Logic and Algebraic Programming*, 56(1):3–21, 2003. doi:10.1016/S1567-8326(02)00065-6.

18 Nan Jiang and Lihong Li. Doubly robust off-policy value evaluation for reinforcement learning. In *International Conference on Machine Learning*, pages 652–661. PMLR, 2016.

19 Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

20 T. Kloek and H. K. van Dijk. Bayesian estimates of equation system parameters: An application of integration by Monte Carlo. *Econometrica*, 46(1):1–19, 1978. doi:10.2307/1913641.

21 Matthieu Komorowski, Leo A. Celi, Omar Badawi, Anthony C. Gordon, and A. Aldo Faisal. The artificial intelligence clinician learns optimal treatment strategies for sepsis in intensive care. *Nature Medicine*, 24(11):1716–1720, 2018.

22 Erik Learned-Miller and Philip S. Thomas. A new confidence interval for the mean of a bounded random variable, 2020. arXiv:1905.06208.

23 P. Massart. The tight constant in the Dvoretzky-Kiefer-Wolfowitz inequality. *Annals of Probability*, 18(3):1269–1283, July 1990. doi:10.1214/aop/1176990746.

24 A. Maurer and M. Pontil. Empirical Bernstein bounds and sample variance penalization. In *Proceedings of the Twenty-Second Annual Conference on Learning Theory*, pages 115–124, 2009.

25 Rozalina G McCoy, Holly K Van Houten, Jeanette Y Ziegenfuss, Nilay D Shah, Robert A Wermers, and Steven A Smith. Increased mortality of patients with diabetes reporting severe hypoglycemia. *Diabetes Care*, 35(9):1897–1901, 2012.

26 Tarek Mhamdi, Osman Hasan, and Sofiène Tahar. On the formalization of the Lebesgue integration theory in HOL. In Matt Kaufmann and Lawrence C. Paulson, editors, *Proceedings of Interactive Theorem Proving*, volume 6172, pages 387–402. Springer, July 2010. doi:10.1007/978-3-642-14052-5_27.

27 Tarek Mhamdi, Osman Hasan, and Sofiène Tahar. Formalization of entropy measures in HOL. In Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Proceedings of Interactive Theorem Proving*, volume 6898, pages 233–248. Springer, August 2011. doi:10.1007/978-3-642-22863-6_18.

28 Tetsuro Morimura, Masashi Sugiyama, Hisashi Kashima, Hirotaka Hachiya, and Toshiyuki Tanaka. Nonparametric return distribution approximation for reinforcement learning. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 799–806. Citeseer, 2010.

29 A. Papoulis. *The Fourier Integral and its Implications*. McGraw-Hill Book Company, Inc., New York, NY, 1962.

30 Lerrel Pinto, James Davidson, Rahul Sukthankar, and Abhinav Gupta. Robust adversarial reinforcement learning. *arXiv preprint*, 2017. arXiv:1703.02702.

31 LA Prashanth and Mohammad Ghavamzadeh. Actor-critic algorithms for risk-sensitive MDPs. In *Advances in Neural Information Processing Systems*, pages 252–260, 2013.

32 D. Precup, R. S. Sutton, and S. Singh. Eligibility traces for off-policy policy evaluation. In *Proceedings of the 17th International Conference on Machine Learning*, pages 759–766, 2000.

33 A. Shamilov, A. F. Yuzer, E. Agaoglu, and Y. Mert. A method of obtaining distributions of transformed random variables by using the Heaviside and the Dirac generalized functions. *Journal of Statistical Research*, 40(1):23–34, 2006.

34 Student. The probable error of a mean. *Biometrika*, pages 1–25, 1908.

35 R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

36 Aviv Tamar, Yonatan Glassner, and Shie Mannor. Optimizing the CVaR via sampling. In *AAAI*, pages 2993–2999, 2015.

37 P. S. Thomas, G. Theocharous, and M. Ghavamzadeh. High confidence off-policy evaluation. In *Proceedings of the Twenty-Ninth Conference on Artificial Intelligence*, 2015.

38 Philip S. Thomas. *Safe Reinforcement Learning*. PhD thesis, University of Massachusetts Libraries, 2015.

39 Philip S. Thomas and Emma Brunskill. Data-efficient off-policy policy evaluation for reinforcement learning. In *International Conference on Machine Learning*, pages 2139–2148. PMLR, 2016.

40 Philip S. Thomas, Bruno Castro da Silva, Andrew G. Barto, Stephen Giguere, Yuriy Brun, and Emma Brunskill. Preventing undesirable behavior of intelligent machines. *Science*, 366(6468):999–1004, 2019.

41 Chun Tian. The new HOL-Probability based on [0,+Inf]-measure theory, September 2019. URL: https://github.com/HOL-Theorem-Prover/HOL/commit/c4db120ba392910141cc83672cc9bd6435e17c9a.

Compositional Verification of Interacting Systems Using Event Monads

Bohua Zhan
State Key Lab. of Computer Science, Institute of Software,
Chinese Academy of Sciences, Beijing, China

Yi Lv
State Key Lab. of Computer Science, Institute of Software,
Chinese Academy of Sciences, Beijing, China

Shuling Wang
State Key Lab. of Computer Science, Institute of Software,
Chinese Academy of Sciences, Beijing, China

Gehang Zhao
School of Mathematical Sciences, Peking University, Beijing, China

Jifeng Hao
Aeronautics Computing Technique Research Institute, Xi'an, China

Hong Ye
Aeronautics Computing Technique Research Institute, Xi'an, China

Bican Xia
School of Mathematical Sciences, Peking University, Beijing, China

Abstract

Large software systems are usually divided into multiple components that interact with each other. How to verify interacting components in a modular way is one of the major problems in formal verification. In many cases, interaction between components can be modeled asynchronously, where events are sent without requiring a response in order to continue with execution of the component. In this paper, we propose a lightweight, event-based framework for verification of components with asynchronous interaction. We define event monads and event systems, and a Hoare logic-style calculus for reasoning about them. The framework is implemented in Isabelle and applied to several case studies, including models for distributed computing, cache-coherence protocols, and verification of partition scheduling in a real-time operating system.

2012 ACM Subject Classification Theory of computation

Keywords and phrases Hoare Logic, Compositional Verification, Events

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.33

Supplementary Material *Software (Source Code)*: `https://github.com/bzhan/EventSystem`
archived at `swh:1:dir:226d9c470f428a9e6c765fbea641d0b619b908cb`

Funding This work was partially supported by the National Natural Science Foundation of China under Grant Nos. 61732001, 62002351, 62032024, 61972385.

1 Introduction

In the verification of large-scale computer programs and systems, a major challenge is modular verification: how to verify components of a system independently, and then compose results from verification of each component into an overall correctness result for the entire system. Sometimes, only part of the system is available or needs to be verified, and the question arises of how to properly model the interaction points between the parts to be verified and the rest of the system.

Interaction between components of a computer system comes in many types. The simplest is when one component of the system makes function calls to another component. In this case, if the callee is completely verified, or at least if an abstraction of its behavior is available, then the caller can be verified in terms of the verified or assumed behavior of the callee. Somewhat more complicated is the situation where two or more components make function calls on each other. Verification approaches designed for such situations include assume-guarantee reasoning [1], where during the verification of each component, we make assumptions on the behavior of the components it relies on, and prove guarantees of its own behavior under these assumptions. Correctness of the composed system is then proved by showing that the guarantees of the components entail all of the assumptions.

Another way to describe interactions between components of a system or with the environment is via effects and effect handlers. There is a long line of work on the modeling and verification of effects and effect handlers [4, 24, 30], which will be reviewed in more detail in Section 6. The main idea is to model each interaction as an uninterpreted event that returns a result, and provide a continuation for each possible value of the result, hence modeling the program as an interaction tree [21, 34]. This technique has been applied successfully to the verification of swap servers [21] and an HTTP Key-Value Server [36].

In many applications, interactions between components are of a special kind, where each component sends out events without requiring an immediate response in order to continue with its execution. This can be used, for example, to model asynchronous function calls, commands to other components to carry out some action, or outputs to the environment. In such cases, interactions can be modeled by lists of output events, together with handlers for such events, which can result in change of state in other components as well as possibly further events. In situations where such a modeling technique is applicable, it provides a simpler and sometimes more accurate way of modeling interaction between components. In this paper, we propose a framework for modeling and verifying components with such asynchronous interactions, by defining event monads and event systems, as well as a Hoare logic-style calculus for reasoning about them. Using several case studies, we demonstrate that this framework can be applied to a wide range of situations, allowing verification of functional properties in a clear and modular way.

The main motivation for the current work comes from a project for verification of partition scheduling in a commercial real-time operating system implemented following the ARINC 653 standard. The standard requires that the physical resources of the computer are divided into several partitions, and the operating system enforces strong spatial and temporal separation between the partitions. To achieve this, scheduling between partitions is strictly deterministic, based on pre-specified time tables (which however can be switched at run time in response to specific events). As partition scheduling is critical to ensure temporal separation as well as real-time properties of the entire system, it is of strong interest to verify its correctness and precision.

While the scheduling policy based on time tables is itself quite simple, its actual implementation is complicated due to efficiency considerations and the need to support switching between time tables. The implementation involves two components, the scheduler and the watchdog, that interact with each other as well as with other parts of the system. The scheduler adds new tasks with deadlines to the watchdog, and the watchdog invokes the dispatch function of the scheduler when the deadline is reached. The scheduler also receives calls to switch time tables from the environment, and emits calls to change the partition. Likewise, the watchdog must handle tasks with deadlines from other modules. We show that the framework based on event monads can be used to model such bi-directional interactions as

well as interactions with the environment, resulting in the modular verification of correctness of partition scheduling based on time tables. The verification is at design-level, with parts of the model closely following the C implementation.

In addition to the application to partition scheduling in real-time operating systems, we also present two smaller case studies, demonstrating the applicability of the framework to other situations. First, we verify a model of distributed computing based on MapReduce. In this model, the client divides a large computing task into several parts, with each part sent to a different server. Each server asynchronously returns the result of the corresponding task after some time. The client then obtains the final result of computation by adding up the answers. Verification of the model requires reasoning about the bi-directional interaction between the client and the servers. Second, we verify a model of cache-coherence protocol. The protocol to be verified is first proposed by Steven German, and is widely used as a test case in parameterized verification (e.g. in the work of Chou et al. [6]). It involves a number of clients that can obtain either exclusive or shared access to some data, by interacting with the server through request and invalidate messages. We model such interactions using the event monad, and prove that the entire system does guarantee exclusive access when required.

1.1 Implementation in Isabelle/HOL

The work described in this paper is implemented[1] in Isabelle/HOL [29]. We base the implementation on the AutoCorres library [12], mainly to take advantage of its `wp` tactic for verification condition generation. The development of event monads in Isabelle is inspired by, but does not depend logically on the development of nondeterministic state monads in AutoCorres [7]. Nor do we make use of its translation facility from C code.

The rest of this paper will make free use of Isabelle notation. We will just review some frequently-used symbols. `'a × 'b` denotes the product of two types, with elements in the form `(a, b)`. Functions `fst` and `snd` return the first and second component of a pair. `f ‘ S` denotes the image of function `f` on the set `S`. The symbols `@` and `#` denote append and cons operations on lists, and `xs ! i` denotes taking the i^{th} element of a list. We will also make frequent use of inductive predicates, using the keyword **inductive** in Isabelle/HOL, or the version generating sets using **inductive_set**.

1.2 Outline of the paper

In Section 2, we motivate the theory in this paper using the example of partition scheduling based on time tables. In Section 3, we define event monads and its associated Hoare logic. In Section 4, we describe how to combine different components into a single event system, and additional rules for reasoning about event systems. In Section 5, we demonstrate the framework on two smaller examples: MapReduce and cache-coherence protocol, as well as the main application on partition scheduling. We discuss related work in Section 6 and finally conclude in Section 7.

2 Motivation: Partition Scheduling

In this section, we describe the motivating example of this paper: scheduling for a partitioned real-time operating system implementing the ARINC 653 standard.

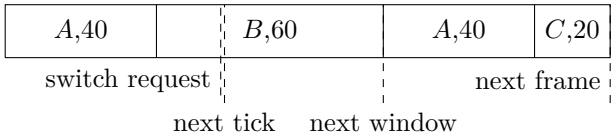
[1] Code available at `https://www.github.com/bzhan/EventSystem`

ARINC 653 is an international standard for real-time operating systems in the aerospace industry [2]. The standard specifies that computing resources are divided into several partitions, with each task running in a single partition. Strong spatial and temporal separation are enforced between partitions, so that failure in one partition will not propagate to affect tasks in other partitions. Part of the mechanism for enforcing temporal separation is a strictly deterministic scheduling policy between partitions based on time tables. A *time table* specifies the order and allotted time of partition executions in each *major time frame*. Here time is given in units of *ticks*. For example, the following time table

A,40	B,60	A,40	C,20

has a major time frame of 160 ticks and consists of four *windows*, where partition A executes for the first 40 ticks, partition B executes for the next 60 ticks, partition A executes again for the next 40 ticks, and finally partition C executes for the remaining 20 ticks.

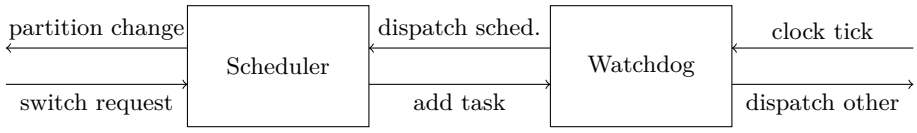
A secondary feature of the scheduling module is the support for switching between time tables. Switching to a new time table can be requested at any time, but is not carried out immediately. Instead the time when the actual switch occurs depends on the *switch mode*. There are three possible modes: next tick, next window, and next frame. Their meanings are straightforward, and are illustrated in the following diagram.



While the functionality described above is relatively simple, its actual implementation is more complex. This is mainly due to efficiency considerations. Within the operating system, there are many modules that require keeping time. It would be costly to invoke each module that requires time-keeping at every clock tick. Instead, time-keeping is centralized in a single module called the *watchdog*. Each module can register tasks on the watchdog, each with a specified deadline in the number of ticks. The watchdog will then *dispatch* each task exactly at its deadline.

Hence, we can consider partition scheduling as an interaction between two components: the scheduler and the watchdog. At initialization, the scheduler sets the partition to that of the first window in the time table, and registers a task to the watchdog with the number of ticks in the first window as deadline. When the task is dispatched, the scheduler sets the partition to that of the next window, and registers a new task to be dispatched after the number of ticks in the next window. This pattern continues, rotating to the first window after reaching the end of the frame. For the time table switch requests, the switch mode and the ID of the next time table are immediately recorded. The switch mode is checked at every dispatch, which will correctly handle the next window and next frame requests. A special check is needed at every clock tick in order to handle the next tick request.

While the scheduler and the watchdog can be viewed as independent modules, there are interactions between them in both directions: the scheduler sends requests to add tasks to the watchdog, while the watchdog dispatches functions in the scheduler. Moreover, both the scheduler and the watchdog interact with other parts of the system: the scheduler receives switch requests, and invokes change of partition. The watchdog receives clock ticks, and may dispatch tasks for other modules. These are illustrated in the following diagram.



All these interactions are of asynchronous nature: they do not require an immediate response in order to continue with the execution. Instead, we can accurately model the system as emitting all interaction events at the end of each function.

As illustration, we give the definition of events in this case study. First, we define some basic datatypes:

```
type_synonym partition = nat
type_synonym ttbl_id = nat
type_synonym task_id = nat
datatype switch_mode = NO_SWITCH | NEXT_TICK | NEXT_WINDOW | NEXT_FRAME | ONGOING
```

The datatype of events is given by

```
datatype event =
  TICK | DISPATCH nat | SWITCH ttbl_id switch_mode | PARTITION partition |
  WATCHDOG_ADD task_id × nat | WATCHDOG_TICK | WATCHDOG_REMOVE task_id
```

Here `TICK` is the global tick operation, which calls on `WATCHDOG_TICK` as well as checks whether the current switching mode is *next tick*. `DISPATCH i` dispatches the task with index i on the watchdog. We assume this dispatches the scheduler task if $i = 0$. `SWITCH tid mode` requests a switch to time table `tid` under switch mode `mode`. `PARTITION p` denotes change of partition to `p`. The events `WATCHDOG_ADD`, `WATCHDOG_TICK` and `WATCHDOG_REMOVE` corresponds to adding a task, time increment, and removing a task on the watchdog, respectively. The scheduling component handles events `DISPATCH 0` and `SWITCH`, and outputs events `PARTITION` and `WATCHDOG_ADD`. The watchdog component handles events `WATCHDOG_ADD`, `WATCHDOG_TICK` and `WATCHDOG_REMOVE`, and sends `DISPATCH` events. The environment provides implementation of the `TICK` events, which may output `WATCHDOG_TICK`, `WATCHDOG_REMOVE`, and `DISPATCH` events.

The overall approach is a *compositional* verification of the system. First, implementations of the scheduler and the watchdog are specified independently, and we verify their properties (as refinements of functional specifications, including the trace of interaction events). Then, the combined system is specified, and its property (a refinement of an overall functional specification) is verified using results proved about the two components.

3 Event Monad

3.1 Definitions of Event Monads

First, we review the concept of (nondeterministic) state monads, as formalized in Isabelle by Cock et al. in [7]. A state monad over a state of type `'s` and returning a value of type `'a` is given by the type `'s ⇒ ('a × 's) set × bool`. Given an input state s, it returns a pair (rs, b), where rs is a set of possible pairs of return value and output state, and b is a failure flag indicating whether it is possible for the computation to fail (including non-termination). The bind operation $f \gg\!= g$ executes f first, then executes g applied to the return value of f. It can fail if either f or g can fail.

The event monad is an extension of the state monad, where we also record a trace of *events* produced by the program. The formalism is parameterized over a type `'s` of states and a type `'e` of events. Then the event monad with return type `'a` is defined as follows.

```
type_synonym ('s,'e,'a) event_monad = 's ⇒ ('a×'s×'e list) set × bool
```

Given an input state s, the function returns a pair (rs, b), where rs is now a set of *triples* of return value, output state, and trace of interaction events. The meaning of the second component b is the same as before.

We begin by defining some basic event monads: `skip` does nothing, `return` returns the given value, and `signal` raises the given event:

```
return a = (λs. ({(a, s, [])}, False))
signal e = (λs. ({((), s, [e])}, False))
skip = return ()
```

The behavior of bind $f \gg\!= g$ is similar to that of usual state monads, except the trace of events produced by g is appended to the trace produced by f to give the overall trace of events. The formal definition is as follows. As preparation, we define `prepend_event` for prepending onto the event trace, and `bind_cont` for applying monad g to an intermediate result of computation:

```
prepend_event e1 (b, s2, e2) = (b, s2, e1 @ e2)
bind_cont g (a, s1, e1) = (let (rs2, f2) = g a s1 in (prepend_event e1 ‘ rs2, f2))
```

Then we define:

```
f ≫= g = (λs. let (rs1, f1) = f s in
           let rss2 = bind_cont g ‘ rs1 in
            (⋃(fst ‘ rss2), f1 ∨ True ∈ (snd ‘ rss2)))
```

Next, we define event monads for retrieving the state, setting the state to s, and modifying the state using a function f. They are similar to the analogous definitions for state monads.

```
get = (λs. ({(s, s, [])}, False))
put s = (λ_. ({((), s, [])}, False))
modify f = (λs. put (f s))
```

The while loop is defined similarly as in the state monad. Given a loop condition C of type `'r ⇒ 's ⇒ bool`, and a loop body of type `'r ⇒ ('s,'e,'r) event_monad`, the expression `whileLoop C B` has type `'r ⇒ ('s,'e,'r) event_monad`. Given an initial value r of type `'r` and state s, it repeatedly executes the loop body B until the condition C becomes false, using the return value of B to reset the value of r after each iteration. In addition, the traces of events produced by C at every iteration are appended in sequence to give the overall trace of events.

Finally, we give the definition of non-deterministic choice between two monads:

```
f ⊔ g = (λs.  let (rs1, f1) = f s; (rs2, f2) = g s in (rs1 ∪ rs2, f1 ∨ f2))
```

3.2 Hoare Logic for Event Monads

We now present a Hoare logic for reasoning about event monads. It will turn out that the definition of Hoare triples as well as the Hoare rules are very similar to that for nondeterministic state monads in [7]. For reference, we repeat these definitions here.

```
⦃P⦄ f ⦃Q⦄ = (∀s. P s ⟶ (∀(r,s') ∈ fst (f s). Q r s'))
```

```
no_fail P f = (∀s t. P s t ⟶ ¬(snd (f s)))
⦃P⦄ f ⦃Q⦄! ⟷ ⦃P⦄ f ⦃Q⦄ ∧ no_fail P f
```

For the extension to event monads, the main decision that needs to be made is where to include the dependence on the trace of events. We choose to allow *both* the precondition and postcondition to depend on traces. The definition for partial correctness in event monads is as follows, and the definition for total correctness is the same as before.

```
{|P|} f {|Q|} = (∀s es1. P s es1 ⟶ (∀(r,s',es2) ∈ fst (f s). Q r s' (es1 @ es2)))
```

In words, given an initial state s and trace `es1` satisfying precondition P, if the execution of f gives return value r, final state s' and trace `es2`, then the triple r, s' and `es1 @ es2` satisfies the postcondition Q. In a sense the trace is viewed as part of the state, with each signal operation appending onto it. The Hoare logic rules have mostly the same form as before. The only new rule is that for signal:

```
{|λs es. P () s (es @ [e])|} signal e {|P|}!
```

As another example, we show the total correctness Hoare triple for the while loop. Note the invariant is a function of the state s and the value r of variables modified at each iteration of the loop, as well as the current trace `es`, while the loop condition cannot depend on the current trace. Here `wf R` means `R` is a well-founded relation.

```
(⋀s es. P s es ⟹ I r s es) ⟹
(⋀r0 s0. {| λs es. I r0 s es ∧ C r0 s ∧ s = s0 |} B r0
          {| λr' s' es'. I r' s' es' ∧ ((r', s'), (r0, s0)) ∈ R |}!) ⟹
wf R ⟹ (⋀r s es. I r s es ∧ ¬C r s ⟹ Q r s es) ⟹
{|P|} whileLoop C B R {|Q|}!
```

What distinguishes event monads and its Hoare logic from simply recording the trace within the state is a *frame rule*, which reflects the fact that the trace of events can only be appended onto, not removed or modified. This rule allows us to show for each function only the Hoare triple where the precondition requires the trace to be empty, using the frame rule to cover other cases when necessary.

First, we define the nil and chop assertions on events:

```
nil_e = (λes. es = [])
P ^_e Q = (λes. ∃es1 es2. P es1 ∧ Q es2 ∧ es = es1 @ es2)
```

The frame rule is then as follows.

$$\frac{\{|\lambda s\ es.\ P\ s \wedge nil_e\ es|\}\ c\ \{|\lambda r\ s\ es.\ P'\ r\ s \wedge Q\ s\ es|\}}{\{|\lambda s\ es.\ P\ s \wedge R\ es|\}\ c\ \{|\lambda r\ s\ es.\ P'\ r\ s \wedge (R\ \hat{}_e\ Q\ s)\ es|\}}$$

An alternative form of the frame rule, more convenient when the list of output events is a function of the initial state, is given as follows.

$$\frac{\{|\lambda s\ es.\ s = s0 \wedge es = []|\}\ c\ \{|\lambda r\ s\ t.\ Q\ r\ s \wedge es = g\ s0|\}}{\{|\lambda s\ es.\ s = s0 \wedge es = es0|\}\ c\ \{|\lambda r\ s\ es.\ Q\ r\ es \wedge es = es0\ @\ g\ s0|\}}$$

The form of definitions of nil and chop, and the naming of the frame rule will remind the reader of separation logic [31]. However, there are some essential differences: compared to separating conjunction, the chop operator represents joining in the temporal rather than the spatial dimension. It should also be noted that the chop operator is not commutative. In fact our setting is closer to that of interval temporal logic [15] and duration calculus [5].

▶ Remark 1. Another possible choice is to always assume the trace to be empty in the precondition, and allow only the postcondition to depend on the trace, perhaps also separating it into two assertions, on the state and trace respectively. However, with this choice, the Hoare rules will no longer be in weakest precondition form, making verification condition generation more difficult. We also note that our approach allows postconditions to state relationships between the final state and the trace (e.g. there exists n such that the value of variable `x` is n and the additional trace contains exactly n events).

4 Event System

With event monads, we can specify and verify properties of programs that produce a trace of events. However, what makes events truly useful is in combination with event handlers. In this section, we define the concept of event system as a model of reactive system consisting of event monads, and a Hoare logic-style calculus for reasoning about event systems.

4.1 Definition of Event System

An event system models a reactive system as a partial mapping from events to their handlers, which take the form of event monads with the same event type, and no return value:

```
type_synonym ('e,'s) event_system = 'e ⇒ ('s,'e,unit) event_monad option
```

We now define the execution of an event e in event system `sys`. The intuitive idea is as follows. If e is not handled by `sys`, then it is simply output to the environment. Otherwise, the event monad handling e is executed. Suppose the resulting trace of events is `es`, then each event in `es` is recursively executed in sequence.

The formal definition is given by two inductive predicates defined by mutual recursion. The predicate `reachable sys e s (s', es')` means starting from state `s`, executing event `e` can reach state `s'` and output event trace `es'` to the environment. The predicate `reachable_list sys es s (s', es')` is similar, except `es` is a list of events to be executed in sequence. Note the output trace `es'` does *not* include events that are handled within the system.

```
reachable_None: sys e = None ⟹ reachable sys e s (s, [e])
reachable_Some: ⟦sys e = Some p; (r, s', es) ∈ fst (p s); ¬snd (p s);
             reachable_list sys es s' (s'', es')⟧ ⟹ reachable sys e s (s'', es')

reachable_list_Nil: reachable_list sys [] s (s, [])
reachable_list_Cons: ⟦reachable sys e s (s', es1); reachable_list sys es s' (s'', es2)⟧
  ⟹ reachable_list sys (e # es) s (s'', es1 @ es2)
```

We also define (by a similar induction) the concept of guaranteed termination when executing an event `e` or a sequence of events `es` starting from state `s`. These are written as `terminates sys e s` and `terminates_list sys es s`.

```
terminates_None: sys e = None ⟹ terminates sys e s
terminates_Some: sys e = Some p ⟹ ¬snd (p s) ⟹
   (∀(r,s',es') ∈ fst (p s). terminates_list sys es' s') ⟹ terminates sys e s

terminates_list_Nil: terminates_list sys [] s
terminates_list_Cons: terminates sys e s ⟹
   (∀s' es'. reachable sys e s (s', es') ⟶ terminates_list sys es s') ⟹
   terminates_list sys (e # es) s
```

4.2 Hoare Logic for Event Systems

Based on the Hoare logic for event monads, we present a Hoare logic-style calculus for reasoning about event systems. Since handlers for events are assumed to have no return values, both pre- and post-conditions are predicates on the pair of state and event trace only. We define partial and total correctness of an event system for a single event and a sequence of events as follows.

```
sValid sys P e Q =
 (∀s es1 s' es2.  P s es1 ⟶ reachable sys e s (s', es2) ⟶ Q s' (es1 @ es2))
sNo_fail sys P e = (∀s es.  P s es ⟶ terminates sys e s)
sValidNF sys P e Q ⟷ sValid sys P e Q ∧ sNo_fail sys P e

sValid_list sys P es Q =
 (∀s es1 s' es2.  P s es1 ⟶ reachable_list sys es s (s', es2) ⟶ Q s' (es1 @ es2))
sNo_fail_list sys P es = (∀s es'.  P s es' ⟶ terminates_list sys es s)
sValidNF_list sys P es Q ⟷ sValid_list sys P es Q ∧ sNo_fail_list sys P es
```

We now state rules for deriving Hoare triples for event systems. The rules are stated for total correctness only, with the partial correctness case being similar. First, if an event `e` is not handled by the system, it just appends to the trace:

$$\frac{\texttt{sys e = None}}{\texttt{sValidNF sys (}\lambda\texttt{s t. P s (t @ [e])) e P}}\ \text{SYS-NONE}$$

The central rule concerns the case where e is handled by the system. It makes use of the Hoare triple for the corresponding event monad c. To show the overall postcondition R, we need to show for each intermediate state s and event trace es that is allowed by the postcondition of c, that R is satisfied after executing es starting from s. This is expressed in the following:

$$\frac{\begin{array}{c}\texttt{sys e = Some c}\\ \{\!|\lambda\texttt{s es. P s} \wedge \texttt{nil}_e\ \texttt{es}|\!\}\ \texttt{c}\ \{\!|\lambda\texttt{r s es. Q s es}|\!\}!\\ \bigwedge\texttt{s es. Q s es} \Longrightarrow \texttt{sValidNF\_list sys (}\lambda\texttt{s' es'. s' = s} \wedge \texttt{nil}_e\ \texttt{es')\,es R}\end{array}}{\texttt{sValidNF sys (}\lambda\texttt{s es. P s} \wedge \texttt{nil}_e\ \texttt{es) e R}}\ \text{SYS-SOME}$$

This rule only considers the case where the initial trace is empty. The frame rule (to be described below) is then used for the general case.

The two rules for event lists are mostly straightforward:

$$\texttt{sValidNF\_list sys P [] P}\quad \text{SYS-NIL}$$

$$\frac{\texttt{sValidNF sys P e Q}\quad \texttt{sValidNF\_list sys Q es R}}{\texttt{sValidNF\_list sys P (e \# es) R}}\ \text{SYS-CONS}$$

We can then prove Hoare triples for general lists of events using induction rules in Isabelle. For example, the following rule is proved by induction on the length of `es`, for the case where none of the events in `es` is handled by the system:

$$\frac{\forall\texttt{e}\in\texttt{set es.}\quad \texttt{sys e = None}}{\texttt{sValidNF\_list sys (}\lambda\texttt{s t. P s (t @ es)) es P}}\ \text{SYS-ALL-NONE}$$

Another useful rule concerns append of event lists, stated simply as follows:

$$\frac{\texttt{sValidNF\_list sys P es1 Q}\quad \texttt{sValidNF\_list sys Q es2 R}}{\texttt{sValidNF\_list sys P (es1 @ es2) R}}\ \text{SYS-APPEND}$$

In addition to the above rules, there are also the usual rules for weakening the precondition, strengthening the postcondition, and dealing with the logical operations. We omit the details here.

Both partial and total correctness Hoare triples for event systems satisfy frame rules. Here we give the rules for total correctness and a single event. The rules for partial correctness and for a sequence of events are similar.

$$\frac{\texttt{sValidNF sys } (\lambda \texttt{s t. P s} \wedge \texttt{nil}_e\,\texttt{t})\ \texttt{e } (\lambda \texttt{s t. P' s} \wedge \texttt{Q s t})}{\texttt{sValidNF sys } (\lambda \texttt{s t. P s} \wedge \texttt{R t})\ \texttt{e } (\lambda \texttt{s t. P' s} \wedge (\texttt{R } \hat{}_e\ \texttt{Q s})\ \texttt{t})}$$

To verify properties of an event system, we first prove appropriate Hoare triples for each event handler as event monads. Then, these results are composed together using the above rules. Often, there is a logical order among events handled by the system, so that for each event handler, any output event that is handled occurs earlier in the order. In this case, the `sValid` and `sValidNF` statements can be proved in sequence following this logical order. This will be demonstrated in Section 5.3.2 (where the order is `WATCHDOG_ADD`, `DISPATCH 0`, `WATCHDOG_TICK`). In more general cases induction techniques would be needed to show several `sValid` and `sValidNF` statements at the same time.

4.3 Composition of Event Systems

Usually, event systems are composed of multiple subsystems, with most of the events acting on some of the subsystems only. In the case studies in Section 5, we will compose subsystems by pairing, as well as using parameterized array of subsystems. We provide support for this by defining functions that automatically transform monads acting on subsystems to monads acting on the entire state, as well as Hoare rules for dealing with monads defined in this way.

First, we consider composition of subsystems by pairing. Given two subsystems with state `'s` and `'t` respectively, we can form a new system with state `'s` × `'t`. Monads acting on `'s` and `'t` can be transformed into monads acting on the global system using the following functions:

```
apply_fst_st t (r, s, es) = (r, (s, t), es)
apply_fst c (s, t) = (let (rs, f) = c s in ((apply_fst_st t) ‘ rs, f))

apply_snd_st s (r, t, es) = (r, (s, t), es)
apply_snd c (s, t) = (let (rs, f) = c t in ((apply_snd_st s) ‘ rs, f))
```

Given a Hoare triple for program `c`, we automatically get a Hoare triple for program `apply_fst c` or `apply_snd c`, using the following Hoare rules (the rules for `apply_snd` and for total correctness are similar).

$$\frac{\{\!|\lambda \texttt{s es. P s es}|\!\}\ \texttt{c}\ \{\!|\lambda \texttt{r s es. Q r s es}|\!\}}{\{\!|\lambda \texttt{p es. P (fst p) es} \wedge \texttt{R (snd p)}|\!\}\ \texttt{apply\_fst c}\ \{\!|\lambda \texttt{r p es. Q r (fst p) es} \wedge \texttt{R (snd p)}|\!\}}$$

Another common way of composing systems is via parameterized array. This is used in both case studies on MapReduce and cache-coherence protocols. We begin by defining a function to transform a monad to apply on the i^{th} index of an array:

```
apply_idx_st slist i (r, s, es) = (r, slist[i := s], es)
apply_idx c i slist = (let (rs, f) = c (slist ! i) in ((apply_idx_st slist i) ‘ rs, f))
```

For reasoning rules about `apply_idx c i`, we provide two versions. The first version is well-suited to the case where the behavior of `c` is deterministic, giving by some function f. Then the behavior of `apply_idx c` is simply applying f to the i^{th} index of the array:

$$\frac{\bigwedge \texttt{s0}.\ \{\!|\lambda \texttt{s es}.\ \texttt{s = s0} \wedge \texttt{es = []}|\!\}\ \texttt{c}\ \{\!|\lambda \texttt{\_ s es}.\ \texttt{s = f s0} \wedge \texttt{es = g s0}|\!\}}{\begin{array}{l}\{\!|\lambda \texttt{s es}.\ \texttt{s = slist} \wedge \texttt{es = []}|\!\}\ \texttt{apply\_idx c i}\\ \{\!|\lambda \texttt{\_ p es}.\ \texttt{s = slist[i := f (slist ! i)]} \wedge \texttt{es = g (slist ! i)}|\!\}\end{array}}$$

The second version is better suited to the case where `c` is characterized by properties on the initial and final states, and there is a certain uniformity between properties satisfied by `s ! i` for each index `i`. This is used, for example, in the verification of MapReduce, where each server satisfies some uniform property parameterized by the data it contains.

$$\frac{\bigwedge \texttt{i}.\ \texttt{i < N} \Longrightarrow \{\!|\lambda \texttt{s es}.\ \texttt{P i s} \wedge \texttt{nil}_e\ \texttt{t}|\!\}\ \texttt{c}\ \{\!|\lambda \texttt{r s es}.\ \texttt{Q i r s es}|\!\}\quad \texttt{j < N}}{\begin{array}{l}\{\!|\lambda \texttt{s es}.\ \texttt{length s=N} \wedge (\forall \texttt{i}.\ \texttt{i < N} \longrightarrow \texttt{P i (s ! i)}) \wedge \texttt{nil}_e\ \texttt{es}|\!\}\ \texttt{apply\_idx c j}\\ \{\!|\lambda \texttt{r s es}.\ \texttt{length s=N} \wedge (\forall \texttt{i}.\ \texttt{i < N} \longrightarrow \neg \texttt{i=j} \longrightarrow \texttt{P i (s!i)} \wedge \texttt{Q j r (s!j) es}|\!\}\end{array}}$$

5 Case Study

In this section, we present three case studies applying the above framework to different scenarios. Two smaller case studies concern distributed computing based on MapReduce, and a cache-coherence protocol proposed by Steven German. The main case study concerns partition scheduling using time tables in a real-time operating system. In all of the case studies, we show that the use of event monads allow us to separately specify and verify each component of the system. The specifications can then be composed together to form an overall correctness result.

5.1 MapReduce

MapReduce is a method of distributed computing proposed by Dean and Ghemawat in [8]. The idea is to divide a large computation task into several smaller portions, each portion consisting of applying some function f (the *map* stage). The results are then combined together by applying another function g onto the initial value and each returned result in sequence (the *reduce* stage). We demonstrate the verification of MapReduce using a simple example, which nevertheless contains the main ingredients, including asynchronous communication and nondeterminism in the time when each machine returns its result.

Given a number `N` and a list of lists of numbers `data` of length `N`, we need to compute the total sum of numbers in `data`. This is done using `N` servers, where each server `i` computes the sum of `data ! i`. The results are then collected together in a client.

The state of each server is as follows:

```
record server =
  input :: nat list
  index :: nat
  cursum :: nat
  returned :: bool
```

Here `input` is the list of numbers whose sum is to be computed. `index` indicates the current progress of computation, and `cursum` records the sum of numbers between 0 and `index - 1`. Finally, `returned` indicates whether the computation is complete, with sum already returned to the client.

The state of the (unique) client is as follows:

```
record client =
  num_received :: nat
  acc :: nat
  alldone :: bool
```

Here `num_received` indicates the number of returned results received from the servers. `acc` is the accumulated value of the sum, and `alldone` indicates whether the entire computation has finished. Note the client does not record which servers it has received results from (hence it does not know the sum of which lists the value `acc` corresponds to), which presents additional challenges to verifying correctness of the system.

The events of the system are defined using the following datatype.

```
datatype event = QUERY nat × nat list | RESPOND nat | TICK | INIT
```

Handlers of the events are as follows. `QUERY` is handled by the server, and initializes its state:

```
query_impl xs = (put (⦇input = xs, index = 0, cursum = 0, returned = False⦈))
```

The event `TICK` applies the following monad to each server node. As long as the node has not returned its answer, it nondeterministically chooses to perform a step, which amounts to either progress the computation by one index, or returning the result when reaching the end.

```
tick_node_impl = (get ≫= (λs.
  (if ¬returned s then
     if index s = length (input s) then
       signal (RESPOND (cursum s)) ≫= (λ_. put (s⦇returned := True⦈))
     else put (s⦇index := index s + 1, cursum := cursum s + input s ! (index s)⦈)
   else skip) ⊔ skip))

tick_impl N = (whileLoop (λi _. i < N)
  (λi. apply_fst (apply_idx tick_node_impl i) ≫= (λ_. return (Suc i))) 0) ≫=
  (λ_. return ())
```

Response is handled by the client, and applies the following monad. It updates the number of received answers and the currently accumulated sum. Moreover, if the number of received answers reaches `N`, it sets the `alldone` flag.

```
respond_impl N a = (get ≫= (λs.
  put (s⦇acc := acc s + a, num_received := num_received s + 1⦈)) ≫= (λ_.
  get ≫= (λs. if num_received s = N then put (s⦇alldone := True⦈) else skip)))
```

The handler for `INIT` uses a while loop to send data to all server nodes:

```
init_impl N data = (whileLoop (λi _. i < N)
  (λi. signal (QUERY (i, data ! i)) ≫= (λ_. return (Suc i))) 0) ≫= (λ_. return ())
```

Finally, we define the event system with global state given by `server list × client` as follows:

```
fun system :: nat⇒nat list list⇒(event, server list×client) event_system where
  system N data (QUERY (i, xs)) = Some (apply_fst (apply_idx (query_impl xs) i))
| system N data (RESPOND a) = Some (apply_snd (respond_impl N a))
| system N data TICK = Some (tick_impl N)
| system N data INIT = Some (apply_snd (init_impl N data))
```

The condition on the initial state is:

```
init_state N (ss, c) ⟷
  length ss = N ∧ c = (|num_received = 0, acc = 0, alldone = False|)
```

We prove the correctness result that starting from the state satisfying `init_state`, after performing one `INIT` and any number of `TICK` events, if the client has set the `alldone` flag, then the value of `acc` in the client must be the total sum of `data`. This is stated formally as follows:

```
N ≥ 0 ⟹ sValidNF_list (system N data)
  (λp es. init_state N p ∧ nilₑ es)
  (INIT # replicate n TICK)
  (λp es. (alldone (snd p) ⟶ acc (snd p) = sum (λi. sum_list (data ! i)) {0 ..< N})
          ∧ nilₑ es)
```

Since whether a step is performed is nondeterministic according to the definition of `tick_node_impl`, it is impossible to predict how many `TICK`s are needed for the `alldone` flag to be set. The same property would hold if a more detailed specification about when to perform a step is used. The main idea of the proof is to verify that `TICK` preserves an invariant of the system, stating that each server node has either returned or is in progress of computing the sum, and the values of `num_received` and `acc` in the client correctly keeps track of the number and total sum of the returned answers. The proof makes key use of the second rule for `apply_idx` given in Section 4.3, lifting the property for each server to a property for the collection of servers.

5.2 Cache-Coherence Protocol

Our second example concerns a cache-coherence protocol proposed by Steven German, which has been widely used as a test case for parameterized verification [6]. The protocol consists of one server and multiple client nodes, and is intended to enforce either exclusive or shared access to some data. If a client node requires exclusive (resp. shared) access, it sends a `ReqE` (resp. `ReqS`) message to the server. On receiving a `ReqE` message, the server sends invalidation messages `Inv` to all client nodes that currently have exclusive or shared access. On receiving an invalidation message, the client sets its own state to `Invalid` and returns an `InvAck` message back to the server. On receiving `InvAck` messages from all clients with access, the server sends an `SendE` message to the client node that initially requested access. On receiving an `SendE` message, the client knows that it now has exclusive access, and sets its own state to `Exclusive`. The handling of `ReqS` is similar, except there is no need to send invalidation messages if no node has exclusive access, and the server sends `SendS` message to the client that initially requested access, who then sets its own state to `Shared`.

Hence, the interaction is mediated by six types of events, defined as follows:

```
datatype event = ReqS nat | ReqE nat | Inv nat | InvAck nat | SendS nat | SendE nat
```

The state of the server is modeled as follows:

```
record server =
  invset :: bool list
  shrset :: bool list
  curptr :: nat option
  grantE :: bool
```

Here `invset` records which client nodes the server is waiting for `InvAck` from. `shrset` indicates which client nodes currently have exclusive or shared access. `curptr` stores the currently requesting client node, and `grantE` indicates whether the requested access is exclusive. The state of the client is simply `record client = st :: state`, where `st` is one of `Invalid`, `Shared`, or `Exclusive`.

In this model, `Inv`, `InvAck`, `SendS`, and `SendE` are all generated by event handlers in either server or clients, whereas `ReqS` and `ReqE` can be seen as events coming from the environment. Correctness of the system can be stated as an invariant that is preserved by the `ReqS` and `ReqE` events, and which implies exclusive access when required. This can be stated for `ReqS` in the following theorem (the one for `ReqE` is similar).

```
i < N ⟹ sValidNF (system N)
  (λp es. system_inv N p ∧ nil_e es)
  (ReqS i)
  (λp es. system_inv N p ∧ nil_e es)
```

where `system_inv` contains a number of conditions, including the following:

```
∀i<N. (st (clist ! i) = Shared ∨ st (clist ! i) = Exclusive) ⟶
       (shrset s ! i ∨ invset s ! i)
∀i<N. ¬invset s0 ! i
grantE s0 ⟶ (∀i j. i < N ⟶ j < N ⟶ i≠j ⟶ ¬shrset s0 ! i ∨ ¬shrset s0 ! j)
```

Together, they state that a client node can be in shared or exclusive state only if the corresponding bit in the `shrset` or `invset` array is turned on. However, before and after each `ReqE` and `ReqS` event, none of the `invset` is turned on, while in the case when `grantE` equals true, at most one `shrset` is turned on. Hence in this case at most one client node has exclusive access.

5.3 Partition Scheduling

We now describe the application of our framework to verify partition scheduling using time tables. We perform two versions of verification: with and without allowing switching between time tables. The version without switching already contains interaction between the scheduler and watchdog in both directions, hence illustrates the main ideas of the framework. The version with switching shows scalability to examples of moderate complexity. Verification of the watchdog is shared between the two versions, and will be described first below.

5.3.1 Watchdog

The watchdog module maintains a set of tasks, each with its own deadline. Tasks are indexed by elements of type `task_id`. Abstractly, the state of a watchdog can be represented by a partial mapping from task ID to deadline (in the number of clock ticks):

```
type_synonym astate_watchdog = task_id ⇒ nat option
```

Concretely, a watchdog is implemented as a doubly-linked list (the *watchdog chain*), where each node represents a task, consisting of task ID and deadline relative to the previous task in the chain. The deadline at the first node of the chain is the actual deadline. The advantage of recording relative deadline is that at each tick, only the deadline at the head of the chain needs to be updated. In this paper, we model the watchdog chain as an array, which preserves most of the logical complexity, without requiring reasoning about linked lists. Hence, the type of concrete watchdog is given by

```
type_synonym watchdog_chain = (task_id × nat) list
```

For example, the watchdog chain $(1, 10), (2, 5), (4, 0), (3, 2)$ means the task with ID 1 is due in 10 ticks, Two tasks with IDs 2 and 4 are due in 15 ticks, and a task with ID 3 is due in 17 ticks.

We define `rel_w` as the refinement relation between abstract and concrete watchdog representations as follows:

```
valid_watchdog es ⟷
 (length es > 0 ⟶ snd (es ! 0) > 0) ∧ (∀evt_id. occurs_atmost_one es evt_id)
event_time [] i = None
event_time (e # es) i =
 (if fst e = i then Some (snd e)
  else case event_time es i of None ⇒ None | Some k ⇒ Some (k + snd e))
rel_w aw cw ⟷ valid_watchdog cw ∧ aw = event_time cw
```

Here `event_time cw i` returns `None` if `i` is not in the watchdog chain `cw`, and `Some k` if it is in the chain with actual deadline `k`. The condition `valid_watchdog` contains the invariant that the first deadline in the chain is always positive, and each task appears at most once in the chain.

The concrete watchdog operations are implemented as follows. Adding a new task requires traversing the chain, inserting the task at the correct location, and decrement the deadline of the next node accordingly. The tick operation first decrements the deadline at the head of the chain by 1, then removes all tasks from the head that have zero deadlines, and emitting their dispatch events. The remove operation first locates the task to be removed in the chain, removes it, then increments the deadline of the next node accordingly.

It is nontrivial to verify the correctness of the watchdog module (see the statistics in Section 5.4). This is stated in terms of refinement between abstract and concrete specifications, including correctness of the list of `DISPATCH` events emitted when handling `WATCHDOG_TICK`, as well as correct update of the watchdog chain when handling `WATCHDOG_ADD` and `WATCHDOG_REMOVE`.

5.3.2 Scheduler with No Switching

We now describe the verification of scheduler without considering switching between time tables. The abstract state of the scheduler consists of the time table (which stays unchanged), and the ID of the current window:

```
record astate_scheduler =
 as_ttbl :: time_table
 window_id :: nat
```

Dispatch increments `window_id` by one (modulo the total number of windows) and produces two output events: change of partition (which is output to the environment in the combined system) and adding to the watchdog (which is handled by the watchdog module).

The concrete state, which corresponds more closely to the actual implementation in C, maintains in addition the length of the current window (`window_time`) and the amount of time passed in the current frame (`cur_frame_time`). The variable `window_id` in the abstract state is renamed to `cur_window`:

```
record cstate_scheduler =
 cs_ttbl :: time_table
 window_time :: nat
 cur_window :: nat
 cur_frame_time :: nat
```

The invariant to be maintained is that `window_time` is the length of the window with index `cur_window`, and `cur_frame_time` is the total length up to (but not including) `cur_window`. During dispatch, first increment `cur_frame_time` by `window_time`; if the result equals the length of the major frame, then `cur_window` and `cur_frame_time` are reset to zero, otherwise `cur_window` is incremented by one; finally `window_time` is updated, and the events `PARTITION` and `WATCHDOG_ADD` are emitted. Again, correctness of the scheduler is stated and proved as refinement between the abstract and concrete specifications. The refinement relation `rel_s` requires that `window_id` in the abstract state equals `cur_window` in the concrete state, and the invariant to be maintained held for the concrete state.

5.3.3 Combined System

The state of the combined system is the product of concrete states for the scheduler and the watchdog. The event handlers are defined in terms of handlers in the two subsystems:

```
sys (DISPATCH 0) = Some (apply_fst dispatch_impl)
sys (WATCHDOG_ADD (ev, n)) = Some (apply_snd (wadd_impl (ev, n)))
sys WATCHDOG_TICK = Some (apply_snd wtick_impl)
```

The refinement relation in the product system is the product of the refinement relations on the two sides:

```
rel_total (as, aw) p ⟷ rel_s as (fst p) ∧ rel_w aw (snd p)
```

We then verify the overall system specifications stated as Hoare triples in the event system, following the method described at the end of Section 4.2. We briefly describe the proved specifications. `WATCHDOG_ADD` adds a new task with given deadline to the watchdog chain, without producing additional events. `DISPATCH 0` outputs a `PARTITION` event to indicate change of partition, as well as adding task 0 with a new deadline back to the watchdog chain. The input event `WATCHDOG_TICK` is the main entry point of the system. Its handler produces a list of `DISPATCH i` events for all tasks $i \neq 0$ that should be dispatched at the current step, as well as performing the action of `DISPATCH 0` if task 0 should be dispatched.

5.3.4 Top-Level Refinement

Based on the results proved in the previous section, it is possible to prove a more abstract specification of the combined system, stating that the scheduling follows the time table precisely. For this, we first define an overall abstract state as follows.

```
record astate =
 a_ttbl :: time_table
 frame_time :: nat
 wchain :: astate_watchdog
```

The state contains the constant time table (`a_ttbl`), the number of ticks spent in the current frame (`frame_time`), and the watchdog mapping excluding the scheduler task (`wchain`). The functional specifications `spec_atick` and `spec_atick_ev` (omitted here) state that `frame_time` increments by 1 at each tick (modulo frame length), change of partition is emitted only at window boundaries, and the usual dispatch events for other tasks are emitted at appropriate times. Then the theorem stating the top-level refinement is:

```
sValidNF sys
 (λp es. arel_full a p ∧ nil_e es)
  WATCHDOG_TICK
 (λp es. arel_full (spec_atick a) p ∧ distinct es ∧ set es = spec_atick_ev a)
```

Table 1 Statistics of the implementation and examples.

Description	Files	Number of lines
Foundations	EventSpecWhile, EventSystem	1131
MapReduce	MapReduce	726
Cache-coherence	Cache	1261
Time table	TimeTable	270
Watchdog	Watchdog, EventSystemWatchdog	2572
Scheduler (no switch)	EventSystemScheduler	723
Scheduler (switch)	EventSystemSwitchScheduler	866
	EventSystemSwitch	1399
Total		8948

Here `arel_full` is the refinement relation between `astate` and the pair of abstract states `(as, aw)` in the previous section. The theorem is proved directly from the previous result for `WATCHDOG_TICK`, by proving the refinement between the specification in the previous section and the specifications `spec_atick` and `spec_atick_ev`.

5.3.5 Scheduler with Switching

We also verified a version of the scheduler allowing switching between time tables, which forms a more substantial example showing the scalability of our framework. For reason of space, we only sketch the main additional features:

- There is an additional input event `SWITCH n mode`, which requests switching to a new time table with identifier n, with switch mode given by *mode*.
- The dispatch function in the scheduler tests for two of the switch modes: next window and next frame, and performs switching at the appropriate time.
- There is an event for overall clock tick which handles the next tick switch mode. If next tick is active, the handler sets the mode to next window, then emits three events: remove the scheduler task from watchdog chain, perform watchdog tick, and dispatch the scheduler. Otherwise, it simply emits the event to perform watchdog tick.

As with the case with no switching, we combined correctness of the scheduler and the watchdog to form correctness result of the overall system. It states that scheduling proceeds precisely according to the time table, and whenever a switch event arrives, the switch to a new time table will be performed at the appropriate time according to the switch mode. The correctness theorem is again stated in the form of a refinement between the concrete behavior of the system and an abstract functional specification.

5.4 Statistics

Statistics from the implementation of the framework and the examples are given in Table 1. Defining the event monad and event system, then setting up the Hoare logic take around 1000 lines in total. Verifying the watchdog is surprisingly complex, so it is a good thing that under the framework, it only need to be done once for verifying the two versions of the scheduler.

6 Related work

There is a large body of work on effects and effect handlers [4, 24, 30], game semantics [20], and verification methods using interaction trees [21, 34, 36], which study how to model and verify interacting systems where interactions are synchronous, and a response is needed immediately in order to proceed with execution of the program. Hence, programs are modeled using *interaction trees* which branch at every continuation in response to an event. Compared to these works, we consider a different model, which may be simpler or more accurate in some settings, where sending events is asynchronous. Closely related is the work of Ahman et al. on asynchronous effects [3]. However, they focused mostly on type checking of programs with asynchronous effects, rather than verification of functional correctness.

Zhao et al. [39] proposed a framework for rely-guarantee reasoning about reactive systems defined by events. In this work (and many earlier works that specify systems using events), each event has a guard, and is triggered whenever the guard is satisfied. The semantics in our case is quite different, where events are triggered explicitly, either by the environment or by other event handlers. In this respect, our semantics is closer to that of I/O automata [26, 27], but with sequential rather than concurrent execution.

There is a large number of frameworks for program verification in Isabelle and other proof assistants, many of which based on monads and/or refinement. We will only give some examples in Isabelle here. Our work builds upon the state monads of Cock et al. [7], which are used extensively in the seL4 project [19]. Lammich et al. developed the Isabelle Refinement Framework [23, 22], which was most recently used by Haslbeck and Lammich for verification of functional correctness and worst-case complexity of algorithms at the LLVM level [16]. Tuong et al. developed Clean [32], which implements a state-exception monad in Isabelle, and is used to verify a number of small programs. Foster et al. developed Isabelle/UTP [11], implementing Hoare and He's Unifying Theories of Programming [17], and applied it to the verification of reactive and hybrid systems [10, 9].

There have been many existing work on verifying operating systems or its components [13, 18, 19, 35, 37]. This includes much work on the specification and verification of separation kernels [40]. Zhao et al. [38] specified channel-based communication according to the ARINC 653 standard, and provided formal proofs about its information flow security. Verbeek et al. [33] formalized the API specification for PikeOS, and proved security properties as required by the MILS architecture. Murray et al. also extended the verification of the seL4 microkernel to prove information flow enforcement properties [28]. More recently, there have been focus on verification of the scheduling in real-time operating systems. In particular, the work of Guo et al. [14] and Liu et al. [25] verified the correctness of scheduling in a real-time version of CertiKOS. They verify both the correct implementation of a scheduling policy as well as schedulability under that policy.

7 Conclusion

In this paper, we introduced a framework for modular verification of interacting components using event monads. Procedures in each component are modeled by event monads which can produce a trace of events. Each event can then be handled by procedures in other components in the event system. We applied the framework to the verification of three examples, including partition scheduling based on time tables in a real-time operating system. These indicate that the framework is applicable in a wide range of scenarios.

While the current paper focuses on verification of distributed systems and operating system components, it appears likely that verification using event monads and event systems can also be applied in other contexts, such as network communication. Exploring these other applications is a goal of future work.

References

1 Martín Abadi and Leslie Lamport. Composing specifications. *ACM Trans. Program. Lang. Syst.*, 15(1):73–132, 1993.

2 Aeronautical Radio Inc. *ARINC Specification 653: Avionics Application Software Standard Interface, Part 1–Required Services*. ARINC Airlines Electronic Engineering Committee, 2015.

3 Danel Ahman and Matija Pretnar. Asynchronous effects. *Proc. ACM Program. Lang.*, 5(POPL):1–28, 2021.

4 Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebraic Methods Program.*, 84(1):108–123, 2015. doi:10.1016/j.jlamp.2014.02.001.

5 Zhou Chaochen, C. A. R. Hoare, and Anders P. Ravn. A calculus of durations. *Inf. Process. Lett.*, 40(5):269–276, 1991. doi:10.1016/0020-0190(91)90122-X.

6 Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. A simple method for parameterized verification of cache coherence protocols. In *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, pages 382–398, 2004.

7 David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, pages 167–182, 2008. doi:10.1007/978-3-540-71067-7_16.

8 Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

9 Simon Foster. Hybrid relations in isabelle/utp. In *Unifying Theories of Programming - 7th International Symposium, UTP 2019, Dedicated to Tony Hoare on the Occasion of His 85th Birthday, Porto, Portugal, October 8, 2019, Proceedings*, pages 130–153, 2019.

10 Simon Foster, James Baxter, Ana Cavalcanti, Alvaro Miyazawa, and Jim Woodcock. Automating verification of state machines with reactive designs and isabelle/utp. In *Formal Aspects of Component Software - 15th International Conference, FACS 2018, Pohang, South Korea, October 10-12, 2018, Proceedings*, pages 137–155, 2018.

11 Simon Foster, James Baxter, Ana Cavalcanti, Jim Woodcock, and Frank Zeyda. Unifying semantic foundations for automated verification tools in Isabelle/UTP. *Sci. Comput. Program.*, 197:102510, 2020.

12 David Greenaway. *Automated proof-producing abstraction of C code*. PhD thesis, University of New South Wales, Sydney, Australia, 2014. URL: http://handle.unsw.edu.au/1959.4/54260.

13 Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 653–669, 2016. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu.

14 Xiaojie Guo, Maxime Lesourd, Mengqi Liu, Lionel Rieg, and Zhong Shao. Integrating formal schedulability analysis into a verified OS kernel. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, pages 496–514, 2019. doi:10.1007/978-3-030-25543-5_28.

15 Joseph Y. Halpern, Zohar Manna, and Ben C. Moszkowski. A hardware semantics based on temporal intervals. In *Automata, Languages and Programming, 10th Colloquium, Barcelona, Spain, July 18-22, 1983, Proceedings*, pages 278–291, 1983. doi:10.1007/BFb0036915.

16 Maximilian P. L. Haslbeck and Peter Lammich. For a few dollars more - verified fine-grained algorithm analysis down to LLVM. In *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, pages 292–319, 2021.

17 C.A.R Hoare and Jifeng He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

18 Gerwin Klein. Operating system verification–an overview. *Sadhana*, 34:27–69, 2009.

19 Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1):2:1–2:70, 2014. doi:10.1145/2560537.

20 Jérémie Koenig and Zhong Shao. Refinement-based game semantics for certified abstraction layers. In *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, pages 633–647, 2020. doi:10.1145/3373718.3394799.

21 Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. From C to interaction trees: specifying, verifying, and testing a networked server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 234–248, 2019. doi:10.1145/3293880.3294106.

22 Peter Lammich. Refinement to imperative HOL. *J. Autom. Reason.*, 62(4):481–503, 2019.

23 Peter Lammich and Thomas Tuerk. Applying data refinement for monadic programs to hopcroft's algorithm. In *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, pages 166–182, 2012.

24 Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. Modular verification of programs with effects and effects handlers. *Formal Aspects Comput.*, 33(1):127–150, 2021. doi:10.1007/s00165-020-00523-2.

25 Mengqi Liu, Lionel Rieg, Zhong Shao, Ronghui Gu, David Costanzo, Jung-Eun Kim, and Man-Ki Yoon. Virtual timeline: a formal abstraction for verifying preemptive schedulers with temporal isolation. *Proc. ACM Program. Lang.*, 4(POPL):20:1–20:31, 2020. doi:10.1145/3371088.

26 Nancy A. Lynch and Eugene W. Stark. A proof of the Kahn principle for input/output automata. *Inf. Comput.*, 82(1):81–92, 1989. doi:10.1016/0890-5401(89)90066-7.

27 Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10-12, 1987*, pages 137–151, 1987. doi:10.1145/41840.41852.

28 Toby C. Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. sel4: From general purpose to a proof of information flow enforcement. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 415–429, 2013.

29 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.

30 Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, York, UK, March 22-29, 2009. Proceedings*, pages 80–94, 2009. doi:10.1007/978-3-642-00590-9_7.

31 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74, 2002. doi:10.1109/LICS.2002.1029817.

32 Frédéric Tuong and Burkhart Wolff. Clean - an abstract imperative programming language and its theory. *Arch. Formal Proofs*, 2019, 2019. URL: https://www.isa-afp.org/entries/Clean.html.

33 Freek Verbeek, Oto Havle, Julien Schmaltz, Sergey Tverdyshev, Holger Blasum, Bruno Langenstein, Werner Stephan, Burkhart Wolff, and Yakoub Nemouchi. Formal API specification of the pikeos separation kernel. In *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, pages 375–389, 2015.

34 Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.*, 4(POPL):51:1–51:32, 2020. doi:10.1145/3371119.

35 Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. A practical verification framework for preemptive OS kernels. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pages 59–79, 2016. doi:10.1007/978-3-319-41540-6_4.

36 Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. Verifying an HTTP key-value server with interaction trees and VST. In *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, pages 32:1–32:19, 2021.

37 Yongwang Zhao and David Sanán. Rely-guarantee reasoning about concurrent memory management in Zephyr RTOS. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, pages 515–533, 2019. doi:10.1007/978-3-030-25543-5_29.

38 Yongwang Zhao, David Sanán, Fuyuan Zhang, and Yang Liu. Reasoning about information flow security of separation kernels with channel-based communication. In *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pages 791–810, 2016.

39 Yongwang Zhao, David Sanán, Fuyuan Zhang, and Yang Liu. A parametric rely-guarantee reasoning framework for concurrent reactive systems. In *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*, pages 161–178, 2019. doi:10.1007/978-3-030-30942-8_11.

40 Yongwang Zhao, Zhibin Yang, and Dianfu Ma. A survey on formal specification and verification of separation kernels. *Frontiers Comput. Sci.*, 11(4):585–607, 2017. doi:10.1007/s11704-016-4226-2.