12th International Conference on Interactive Theorem Proving

ITP 2021, June 29–July 1, 2021, Rome, Italy (Virtual Conference)

Edited by

Liron Cohen
Cezary Kaliszyk

LIPICS

Editors

Liron Cohen
Ben-Gurion University, Be'er Sheva, Israel
cliron@cs.bgu.ac.il

Cezary Kaliszyk
University of Innsbruck, Austria
cezary.kaliszyk@uibk.ac.at

ACM Classification 2012
Theory of computation → Logic

ISBN 978-3-95977-188-7

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/978-3-95977-188-7.

Publication date
June, 2021

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.2021.0

LIPIcs – Leibniz International Proceedings in Informatics

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

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 twelfth to be held under the ITP name, is co-located with the 36th Annual Symposium on Logic in Computer Science (LICS 2021), and was to be held in Rome, Italy. However, due to the COVID-19 global pandemic, the conference will be held in an online format for the first time. 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 57 submissions (51 long papers and 6 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: 29 papers (28 long papers and 1 short). 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 Nadia Polikarpova (University of California, San Diego, joint talk with LICS), Andrei Popescu (University of Sheffield), and Magnus Myreen (Chalmers). The present volume collects all the accepted papers contributed to the conference as well as the latter two invited papers. This is the second 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, as well as the EasyConferences staff for their support in the logistics.

We are grateful to Daniele Gorla for offering to organize ITP in Rome and his flexibility in changing the format in face of the pandemic. We would like to also extend this thanks to all authors and speakers, as we are certain that adjusting to the new format of the conference required some additional effort. Finally we are thankful to the ITP Steering Committee for their guidance throughout.

The CakeML Project's Quest for Ever Stronger Correctness Theorems

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

Abstract

The CakeML project has developed a proof-producing code generation mechanism for the HOL4 theorem prover, a verified compiler for ML and, using these, a number of verified application programs that are proved correct down to the machine code that runs them (in some cases, even down to the underlying hardware). The purpose of this extended abstract is to tell the story of the project and to point curious readers to publications where they can read more about specific contributions.

2012 ACM Subject Classification Theory of computation → Higher order logic

Keywords and phrases Program verification, interactive theorem proving

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.1

Category Invited Paper

Supplementary Material `https://cakeml.org`

Funding This work was partially supported by the Swedish Foundation for Strategic Research, and by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

Acknowledgements I thank the ITP programme committee for inviting me to give this talk. I am grateful for comments on drafts of this text from Michael Norrish, Johannes Åman Pohjola, Andreas Lööw, Yong Kiam Tan, Oskar Abrahamsson and Konrad Slind. I want to thank the team of CakeML contributors for making the CakeML project such a pleasure to work on. Finally, I would like to thank the late Mike Gordon whose enthusiasm for the CakeML project and belief in its contributions are still felt and appreciated to this day.

1 Motivation and Beginning

Around the year 2012, the CakeML project started as a reaction to the practice of using the following steps to produce verified applications using interactive theorem provers (ITPs).

1. Write functions in the logic of an ITP;
2. prove correctness properties of the functions as theorems in the ITP; and
3. ask the ITP to generate SML/OCaml/Haskell code based on the functions in the logic.

Scott Owens and my reaction was the following: the code generators in Step 3 ought to show *with theorems proved in the ITP* that the generated code has the same behaviour as the supplied functions, and the theorems should ideally be expressed in terms of a formal semantics for the programming language in question (SML, OCaml or Haskell). However, the code generators[1] in use at the time did not provide users with such theorems.

The first publication of the CakeML project [32] showed that it is possible to build a proof-producing code generator that automatically proves a theorem stating the relationship between the given function's behaviour and the behaviour of the generated ML code with respect to an operational semantics of a subset of SML.

[1] In Coq terminology, this code generation is called code extraction.

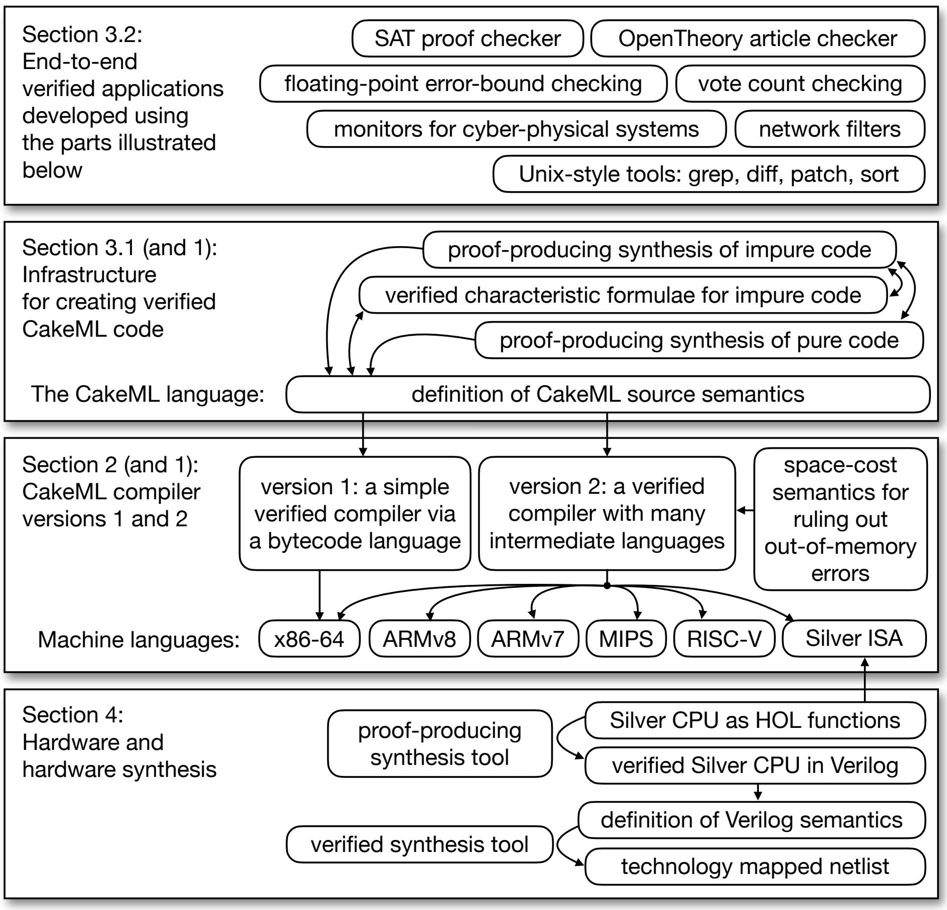

Section 3.2:
End-to-end
verified applications
developed using
the parts illustrated
below
SAT proof checker
OpenTheory article checker
floating-point error-bound checking
vote count checking
monitors for cyber-physical systems
network filters
Unix-style tools: grep, diff, patch, sort
Section 3.1 (and 1):
Infrastructure
for creating verified
CakeML code
proof-producing synthesis of impure code
verified characteristic formulae for impure code
proof-producing synthesis of pure code
The CakeML language:
definition of CakeML source semantics
Section 2 (and 1):
CakeML compiler
versions 1 and 2
version 1: a simple verified compiler via a bytecode language
version 2: a verified compiler with many intermediate languages
space-cost semantics for ruling out out-of-memory errors
Machine languages:
x86-64
ARMv8
ARMv7
MIPS
RISC-V
Silver ISA
Section 4:
Hardware and
hardware synthesis
proof-producing synthesis tool
Silver CPU as HOL functions
verified Silver CPU in Verilog
definition of Verilog semantics
verified synthesis tool
technology mapped netlist

Figure 1 A diagram of the parts of the CakeML project described in this paper.

The introduction to this first publication made it clear that there really ought to be two parts to code generation: A) proof-producing translation of functions in the logic to functions deeply embedded in a programming language, and B) verified compilation for programs written in that programming language down to executable machine code. At the time when Scott Owens and I wrote this, we did not envision that we would deliver on B. However, that changed when Ramana Kumar got involved.

In the build up to the next publication, the project got its name "CakeML" which was initially short for Cambridge and Kent ML, following the tradition of naming SML implementations based on location names. At the time, Ramana and I were at Cambridge and Scott was at Kent. However, as the project grew, we have decided that the name CakeML is to be regarded as just a name rather than an abbreviation.

A close collaboration between Ramana Kumar, Scott Owens, Michael Norrish and myself delivered on B and resulted in the second paper *CakeML: A Verified Implementation of ML* [22], which explains how we created a verified read-eval-print loop implementing the CakeML operational semantics. The second paper has become the canonical reference for the CakeML project as a whole, even though the CakeML compiler has changed significantly since this first version, as will be described in the next section.

This paper gives a tour of the CakeML project and its ambition to achieve end-to-end verified code. Figure 1 illustrates the different parts of the project: the CakeML compiler is at the centre of the figure (Section 2); infrastructure and verified applications are above the compiler (Section 3); below the compiler, we have work that dives into the hardware layers (Section 4). Future work is mentioned in Section 5, but is not part of Figure 1.

2 Verified Compiler and Runtime

The CakeML project has put much focus on its verified compiler, which has evolved from a simple compiler to one with eight intermediate languages and several compilation passes within each intermediate language [41].

2.1 In-Logic Execution and Transportation of Correctness Results

The purpose of the CakeML compiler is to provide a way to generate machine code from given source programs in a way that allows users to transport any properties proved of the source code down to the generated machine code. This transportation feature is, of course, a desire in most compiler verification projects, but the CakeML project has taken this desired feature perhaps more literally than most other compiler verification projects. A property is fully transported when the final theorem is about the generated machine code and does not mention the compiler. In-logic execution of the compiler is necessary for this.

The CakeML compiler is written in such a way that it can be fully executed inside of an ITP. Such in-logic executions result in theorems of the form compile input = machine_code, where input is a concrete source-level program and machine_code is a concrete list of bytes. With concrete compilation results as theorems in the logic, and a standard result that compile only produces machine code that has identical behaviour (up to out-of-memory errors), one can transport properties P proved of the source-level program input down to the machine_code. Crucially, the in-logic compilation allows us to state the resulting theorem in terms of P and the semantics of machine_code *without any mention* of the complicated compile function.

These ideas are explained in Kumar et al. [21], which argues for the importance of ITP code generation mechanisms that prove the correctness down to the level of machine code.

2.2 Compiler Bootstrapping inside an ITP

The CakeML project is perhaps most well known for the fact that it is the first project to bootstrap a verified compiler entirely inside an ITP. Bootstrapping entails running a compiler on itself to generate machine code implementing itself. This seemingly circular concept can seem confusing at first. However, it follows quite directly from the concepts we have seen above: given a compiler function compile defined as a normal function in an ITP, we can apply the proof-producing code generator [32] to generate a behaviourally equivalent source program compile_prog and, to this source program, we can apply the in-logic compilation method described in the previous section. The result is a verified implementation of the compile function as concrete machine code. Separately from CakeML, the idea of bootstrapping a compiler inside an ITP has been described in a simple minimal setting [30].

2.3 The First Version: Compilation in a Read-Eval-Print Loop

The CakeML compiler was, from the start, an end-to-end compiler consisting of lexing, parsing, type inference and code generation. Each part was proved correct: the parser implementation was proved sound and complete with respect to a context-free grammar for Standard ML syntax; our type inferencer was proved sound (and later complete [42] when Yong Kiam Tan joined the effort) w.r.t. the type system; we proved that no typed program will ever hit a runtime error in our operational semantics; and finally we also proved that any program that avoids runtime errors in the source semantics is compiled to identically behaving machine code, up to out-of-memory errors that can happen in the generated machine code. This caveat about out-of-memory errors is necessary because the CakeML source language does not pose any limits on the size of lists, arrays or integers, but the compilation target, x86-64 machine code operates in a finite state space.

The first version of the compiler was bootstrapped and used as a component in a verified x86-64 implementation of a read-eval-print loop for the CakeML language [22]. In this first version, the code generator was simple: it compiled to a stack-based bytecode language, which mapped quite directly to short snippets of x86-64 machine code. The implementation did include a bignum library [31] and a verified garbage collector [28]. The verification of the type inferencer built on prior work on unification [23].

The most challenging part of the first version was perhaps the dynamic compilation aspect: the compiler is repeatedly executed at runtime by the read-eval-print loop, which affected the compiler proofs [18]. We made use of proof methods developed in the context of verification of a just-in-time compiler [29].

2.4 A New Optimising Compiler Backend

Even before the first compiler version was ready, a plan was shaping up to do better both in terms of proof methodology and compiler implementation. We also decided to focus on more conventional static ahead-of-time compilation rather than dynamic compilation.

The proof methodology was improved by switching from relational big-step semantics to functional big-step semantics [35]. Semantics written in functional big-step style take the form of clocked interpreter functions that neatly fit with proof-by-rewriting in higher-order logic. This style of semantics allowed us to conveniently prove preservation of terminating and non-terminating behaviour using only one simulation proof per compiler phase. Most simulation proofs are in the direction of compilation, i.e. forward.

The compiler implementation was redone almost entirely. The new design had several intermediate languages that were designed with optimisations in mind. The new implementation avoided compiling via a stack-based bytecode, and instead used graph-colouring based register allocation. We also took care to compile curried functions into efficient code [36]. At this point Yong Kiam Tan and Anthony Fox had become active in the project. Yong Kiam Tan made significant contributions to the lower-level languages, including register allocation, compilation of calling conventions and the assembler. Anthony Fox was crucial in making the new compiler target five machine languages [11]: 64-bit x86-64, ARMv8, MIPS and RISC-V, and 32-bit ARMv7; later a sixth (the Silver ISA) was added, as will be described in Section 4.

The new compiler implementation has offered plenty of opportunities for student projects and experimental extensions. Students have, for example, contributed new optimisations [3], infrastructure for visualisation of the compiler's transformations [16] and even a generational copying garbage collector [9]. One of the experimental extensions has explored giving CakeML a flexible fast-math-compatible floating-point semantics [5].

The most up-to-date description of the CakeML compiler is Tan et al. [41]. That paper includes benchmarks comparing the performance of the CakeML generated code with code from other ML compilers. The new CakeML compiler generates code with performance numbers that are in the same ballpark as the performance numbers for code compiled with Poly/ML, SML/NJ and native-code compiled OCaml.

2.5 A Space Cost Semantics

The CakeML compiler's correctness statement has always contained an irritating get-out clause: the CakeML compiler's correctness statement allows the generated machine code to exit early due to out-of-memory errors. As mentioned earlier, out-of-memory errors cannot be ruled out in general since the source semantics has no bounds on the size of data and numbers, but the target machine code runs in settings where memory is finite.

To address this wart in the compiler correctness theorem, we have defined a space cost semantics [15] that can be used to prove tight memory bounds for concrete CakeML programs. The cost semantics has been proved sound w.r.t. CakeML's compiler and enables transfer of liveness properties proved for the source code down to liveness properties of the generated machine code.

3 Verified Applications

In parallel with the development of the compiler, we improved our techniques for generating CakeML source code and post hoc verification of manually written CakeML code.

3.1 Characteristic Formulae and Improved Code Generation

In the beginning, we could only produce verified CakeML code using the proof-producing synthesis tool mentioned in Section 1. This tool could initially only target the pure part of the CakeML language. However, the CakeML language includes SML-style impure features such as exceptions and mutable state in the form of references, arrays, and byte arrays. We did take some early steps towards support for impure CakeML code for a journal version [33] of the original synthesis paper, but the solution was ad hoc and unsatisfactory.

Separately from CakeML, Arthur Charguéraud developed a separation logic style reasoning framework called Characteristic Formulae for ML (CFML) for reasoning about OCaml code in Coq [8], and in 2016, I had the fortune of having Armaël Guéneau, who had worked with CFML, visit me for an internship. During his five-month internship, Guéneau produced a CFML variant in HOL for CakeML, and developed and proved it sound w.r.t. CakeML operational semantics. This CakeML adaption [13] of CFML covered every aspect of the CakeML language: (mutually recursive) functions as values, mutable state, exceptions and even reasoning about I/O through CakeML foreign function interface.

Guéneau's original work on Characteristic Formulae (CF) for CakeML has been extended and used for several applications. A file-system model was built on top of it in order to allow us to reason about file accessing programs [10]. For this work, Johannes Åman Pohjola improved CakeML's foreign-function interface. Later, Son Ho contributed an important forward simulation tactic for separation logic CF proofs (unpublished). Most recently, the CF framework was extended to handle correctness proofs of non-terminating programs [4]; this extension makes it possible to prove liveness properties for diverging (productive and non-productive) CakeML programs. A simplified version of this Hoare logic for non-termination has been proved sound and complete.

Once CF was well developed, we revisited our desire for proof-producing code generation that targets CakeML's impure features. We took inspiration from how effectful computations are modelled using monads in Haskell and developed techniques for creating impure CakeML code from monadic HOL functions [2]. Our method builds on the separation logic setup provided by CF for CakeML. The fact that CF for CakeML and our code synthesis tool share infrastructure meant that we were able to provide links by which results proved in one system could be transferred for use in the other. The speed of the CakeML compiler binary was significantly improved by this access to the impure features of CakeML. In a separate improvement to the code generator, some reduction in code bloat was provided by new techniques for code generated for case expressions [43].

In parallel to the work described above which took place in HOL4, Lars Hupel and Tobias Nipkow developed a different CakeML code generator [17] in Isabelle/HOL. This code generator differed from the one in HOL4 in that the Isabelle/HOL version was (for most part) a once-and-for-all verified tool, while the code generator in HOL4 was (for most part) a proof-producing tool. The bridge between Isabelle/HOL and HOL4 was provided by the Lem tool [27]. Most parts of the CakeML source language semantics are specified in the Lem language and, from this prover-independent specification, the Lem tool generates definitions in HOL4 and Isabelle/HOL.

3.2 Example Applications

The tools described above have been used to create end-to-end verified applications:

- We have verified a number of Unix-like tools such as grep, sort, cat, diff, patch, and a word frequence counter. These examples were mostly developed to showcase features of our file system models and CF setup.
- From early on [34], there was a drive to develop a verified proof assistant for higher-order logic based on CakeML. Ramana Kumar made significant progress in this direction [18, 20]. In the process, a new definition mechanism for higher-order logic (proposed by Rob Arthan) was proved sound [19]. Later, this line of work was continued when Arve Gengelbach and Johannes Åman Pohjola [37] proved consistency of ad-hoc overloading, as allowed by constant definitions in Isabelle/HOL.
- There has been work on verified checkers of different kinds: a checker for OpenTheory article files [1], a checker for proof certificates produced by SAT solvers [40], a checker for vote counting [12], and a checker for floating-point error bounds [6].
- Formally specified security components, such as filters, monitors, and attestation schemes, are being synthesized to CakeML running on seL4 [38, 39]; the intent is to thereby improve the security of legacy embedded systems.
- A verified CakeML application was also developed to be a verified monitor for cyber-physical systems [7].

4 Extending down into Hardware

The CakeML project has also dived into the hardware levels of computing systems. Our adventure in the hardware world was motivated by two questions:

1. Can our end-to-end correctness theorems be extended beyond the level of machine code and into the hardware layers? (And if so, down to what layer?)
2. Can the techniques that we have developed for proof-producing code generation and compiler verification be adapted to work for the creation of verified hardware?

Andreas Lööw has explored these questions in his research. For question 2, he formalised a subset of Verilog in HOL4 and built a proof-producing HOL-function-to-Verilog generator [25] using ideas from the proof-producing CakeML code generator.

For question 1, Andreas Lööw defined a small CPU, called Silver, as HOL functions in the subset understood by his Verilog code generator, and proved correctness of the CPU w.r.t. a specification of the custom instruction set architecture (ISA) that it support. The ISA that the Silver CPU implements was added as a target to the CakeML compiler by Anthony Fox. Finally, a number of CakeML developers helped verify a minimal implementation of a file system written in Silver machine code. Together these components allowed us to create *a verified stack* where end-to-end correctness theorems extend from high-level specifications all the way down to the Verilog implementation of the CPU that executes the machine code that the CakeML compiler produces, with all layers in between verified [26]. In short, the result was *a Silver platform for CakeML programs to stand on.*

The exploration of questions 1 and 2 continues: Andreas Lööw has built a verified Verilog-to-netlist compiler [24] that can compile (a subset of) the Verilog produced by his first tool. The netlists are technology mapped netlists for FPGAs.

5 Conclusions and Future Directions

The CakeML project started off as a simple reaction to the state of code generators/extractors and has since snowballed into an ambitious compiler verification project. The core aim has remained the same: to provide users with tools that allow transfer of properties proved about functions in logic down to concrete executable code (machine code or even hardware).

5.1 Reflection

What made it possible for the CakeML project to develop so far? There are, of course, many factors at play and no definite answers, but here are some points I believe are important:

- *Struck a chord.* I believe the end-to-end correctness theorems that the project has as its core aim inspires people to join, and functional programming (and reasoning about functional programs) is of interest to many with relevant backgrounds.
- *Luck.* We were fortunate early on: the initial group of people worked well together which was vital for getting the project off the ground.
- *No initial funding and no named leader.* The CakeML project started and ran for several years without any CakeML-specific funding, which meant that there was no named project leader and people were contributing to the project due to their own interest rather than because they were paid to work on this project.

5.2 What Next?

When something has been built well, I believe it has potential to be a platform to build on. One aspect that I am keen to explore is the use of CakeML and its compiler as part of other projects, e.g. projects that develop verification tools or compilers for languages other than ML. In this direction, there is on-going work within the CakeML project to develop a compiler for a clean low-level imperative language using parts of the CakeML compiler; and, similarly, there is another on-going project developing a verified complier for a Haskell-inspired functional language using the CakeML compiler as a component. There is also on-going work on developing a verified compiler for a choreographic language [14].

References

1 Oskar Abrahamsson. A verified proof checker for higher-order logic. *Journal of Logical and Algebraic Methods in Programming*, 112:100530, 2020. doi:10.1016/j.jlamp.2020.100530.

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

3 Oskar Abrahamsson and Magnus O. Myreen. Automatically introducing tail recursion in CakeML. In Meng Wang and Scott Owens, editors, *Trends in Functional Programming (TFP)*, volume 10788 of *LNCS*, pages 118–134. Springer, 2017. doi:10.1007/978-3-319-89719-6_7.

4 Johannes Aman Pohjola, Henrik Rostedt, and Magnus O. Myreen. Characteristic formulae for liveness properties of non-terminating CakeML programs. In *Interactive Theorem Proving (ITP)*. LIPICS, 2019.

5 Heiko Becker, Eva Darulova, Magnus O. Myreen, and Zachary Tatlock. Icing: Supporting fast-math style optimizations in a verified compiler. In *Computer Aided Verification (CAV)*. Springer, 2019. doi:10.1007/978-3-030-25543-5_10.

6 Heiko Becker, Nikita Zyuzin, Raphael 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)*. IEEE, 2018.

7 Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. VeriPhy: verified controller executables from verified cyber-physical system models. In Jeffrey S. Foster and Dan Grossman, editors, *Programming Language Design and Implementation (PLDI)*, pages 617–630. ACM, 2018. doi:10.1145/3192366.3192406.

8 Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In *International Conference on Functional Programming (ICFP)*. ACM, 2011. doi:10.1145/2034773.2034828.

9 Adam Sandberg Ericsson, Magnus O. Myreen, and Johannes Åman Pohjola. A verified generational garbage collector for CakeML. *Journal of Automated Reasoning (JAR)*, 63, 2019. doi:10.1007/s10817-018-9487-z.

10 Hugo Férée, Johannes Åman Pohjola, Ramana Kumar, Scott Owens, Magnus O. Myreen, and Son Ho. Program verification in the presence of I/O – semantics, verified library routines, and verified applications. In *Verified Software: Theories, Tools, Experiments (VSTTE)*, 2018. doi:10.1007/978-3-030-03592-1_6.

11 Anthony C. J. Fox, Magnus O. Myreen, Yong Kiam Tan, and Ramana Kumar. Verified compilation of CakeML to multiple machine-code targets. In Yves Bertot and Viktor Vafeiadis, editors, *Certified Programs and Proofs (CPP)*, pages 125–137. ACM, 2017. doi:10.1145/3018610.3018621.

12 Milad Ketab Ghale, Dirk Pattinson, and Ramana Kumar. Verified certificate checking for counting votes. In Ruzica Piskac and Philipp Rümmer, editors, *Verified Software. Theories, Tools, and Experiments (VSTTE)*, volume 11294 of *LNCS*. Springer, 2018. doi:10.1007/978-3-030-03592-1_5.

13 Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. Verified characteristic formulae for CakeML. In Hongseok Yang, editor, *European Symposium on Programming (ESOP)*, volume 10201 of *LNCS*. Springer, 2017. doi:10.1007/978-3-662-54434-1_22.

14 Alejandro Gómez-Londoño. Choreographies and cost semantics for reliable communicating systems, 2020. Licentiate thesis, Chalmers University of Technology.

15 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. (OOPSLA)*, 4:204:1–204:29, 2020. doi:10.1145/3428272.

16 Rikard Hjort, Jakob Holmgren, and Christian Persson. The CakeML compiler explorer - tracking intermediate representations in a verified compiler. In Meng Wang and Scott Owens, editors, *Trends in Functional Programming (TFP)*, volume 10788 of *LNCS*, pages 135–148. Springer, 2017. doi:10.1007/978-3-319-89719-6_8.

17 Lars Hupel and Tobias Nipkow. A verified compiler from Isabelle/HOL to CakeML. In Amal Ahmed, editor, *European Symposium on Programming (ESOP)*, volume 10801 of *LNCS*, pages 999–1026. Springer, 2018. doi:10.1007/978-3-319-89884-1_35.

18 Ramana Kumar. *Self-compilation and self-verification.* PhD thesis, University of Cambridge, 2016.

19 Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. HOL with definitions: Semantics, soundness, and a verified implementation. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving (ITP)*, volume 8558 of *LNCS*, pages 308–324. Springer, 2014. `doi:10.1007/978-3-319-08970-6_20`.

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

21 Ramana Kumar, Eric Mullen, Zachary Tatlock, and Magnus O. Myreen. Software verification with ITPs should use binary code extraction to reduce the TCB (short paper). In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving (ITP)*, volume 10895 of *LNCS*, pages 362–369. Springer, 2018. `doi:10.1007/978-3-319-94821-8_21`.

22 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *Principles of Programming Languages (POPL)*. ACM, 2014. `doi:10.1145/2535838.2535841`.

23 Ramana Kumar and Michael Norrish. (nominal) unification by recursive descent with triangular substitutions. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *LNCS*, pages 51–66. Springer, 2010. `doi:10.1007/978-3-642-14052-5_6`.

24 Andreas Lööw. Lutsig: a verified Verilog compiler for verified circuit development. 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 46–60. ACM, 2021. `doi:10.1145/3437992.3439916`.

25 Andreas Lööw and Magnus O. Myreen. A proof-producing translator for Verilog development in HOL. In Stefania Gnesi, Nico Plat, Nancy A. Day, and Matteo Rossi, editors, *Proceedings of the 7th International Workshop on Formal Methods in Software Engineering, FormaliSE@ICSE 2019, Montreal, QC, Canada, May 27, 2019*, pages 99–108. IEEE / ACM, 2019. `doi:10.1109/FormaliSE.2019.00020`.

26 Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. Verified compilation on a verified processor. In *Programming Language Design and Implementation (PLDI)*. ACM, 2019. `doi:10.1145/3314221.3314622`.

27 Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering of real-world semantics. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *International Conference on Functional Programming (ICFP)*. ACM, 2014. `doi:10.1145/2628136.2628143`.

28 Magnus O. Myreen. *Formal verification of machine-code programs*. PhD thesis, University of Cambridge, 2009.

29 Magnus O. Myreen. Verified just-in-time compiler on x86. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Principles of Programming Languages (POPL)*. ACM, 2010.

30 Magnus O. Myreen. A minimalistic verified bootstrapped compiler (proof pearl). In Catalin Hritcu and Andrei Popescu, editors, *Conference on Certified Programs and Proofs (CPP)*, pages 32–45. ACM, 2021. `doi:10.1145/3437992.3439915`.

31 Magnus O. Myreen and Gregorio Curello. Proof pearl: A verified bignum implementation in x86-64 machine code. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs (CPP)*, volume 8307 of *LNCS*, pages 66–81. Springer, 2013. `doi:10.1007/978-3-319-03545-1_5`.

32 Magnus O. Myreen and Scott Owens. Proof-producing synthesis of ML from higher-order logic. In *International Conference on Functional Programming (ICFP)*, pages 115–126. ACM Press, 2012. `doi:10.1145/2364527.2364545`.

33 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):284–315, 2014. `doi:10.1017/S0956796813000282`.

34 Magnus O. Myreen, Scott Owens, and Ramana Kumar. Steps towards verified implementations of HOL light. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving (ITP)*, volume 7998 of *LNCS*, pages 490–495. Springer, 2013. doi:10.1007/978-3-642-39634-2_38.

35 Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional big-step semantics. In Peter Thiemann, editor, *European Symposium on Programming (ESOP)*, LNCS. Springer, 2016. doi:10.1007/978-3-662-49498-1_23.

36 Scott Owens, Michael Norrish, Ramana Kumar, Magnus O. Myreen, and Yong Kiam Tan. Verifying efficient function calls in CakeML. *Proc. ACM Program. Lang.*, 1(ICFP), September 2017. doi:10.1145/3110262.

37 Johannes Aman Pohjola and Arve Gengelbach. A mechanised semantics for HOL with ad-hoc overloading. In Elvira Albert and Laura Kovacs, editors, *Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 73 of *EPiC Series in Computing*, pages 498–515. EasyChair, 2020.

38 Konrad Slind. Specifying message formats with contiguity types. In *Interactive Theorem Proving (ITP)*, Leibniz International Proceedings in Informatics. Schloss Dagstuhl Leibniz-Zentrum für Informatik, Dagstuhl Publishing, 2021.

39 Konrad Slind, David S. Hardin, Johannes Åman Pohjola, and Michael Sproul. Synthesis of verified architectural components for autonomy hosted on a verified microkernel. In *Hawaii International Conference on System Sciences*, January 2020.

40 Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. cake_lpr: Verified propagation redundancy checking in CakeML. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2021. doi:10.1007/978-3-030-72013-1_12.

41 Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29, 2019. doi:10.1017/S0956796818000229.

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

43 Thomas Tuerk, Magnus O. Myreen, and Ramana Kumar. Pattern matches in HOL: A new representation and improved code generation. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving (ITP)*, volume 9236 of *LNCS*, pages 453–468. Springer, 2015. doi:10.1007/978-3-319-22102-1_30.

Synthesis of Safe Pointer-Manipulating Programs

Nadia Polikarpova
University of California in San Diego, La Jolla, CA, USA

Abstract

Low-level pointer-manipulating code is ubiquitous in operating systems, networking stacks, and browsers, which form the backbone of our digital infrastructure. Unfortunately, this code is susceptible to many kinds of bugs, which lead to crashes and security vulnerabilities. A promising approach to eliminating bugs and reducing programmer effort at the same time is to use *program synthesis* technology to generate provably correct low-level code automatically from high-level specifications.

In this talk I will present a program synthesizer SuSLik, which accepts as input a specification written in *separation logic*, and produces as output a provably correct C program. SuSLik is the first synthesizer capable of generating a wide range of operations on linked data structures (such as singly- and doubly-linked lists, binary trees, and rose trees) without additional hints from the user. It is also the first synthesizer to automatically discover recursive auxiliary functions required for nested data structure traversal. To make this possible, SuSLik relies on a novel proof system – *synthetic separation logic* – to derive correct-by-construction programs directly from their specifications. Program proofs generated by SuSLik can be automatically translated into three foundational verification frameworks embedded in Coq: Hoare Type Theory (HTT), Iris, and Verified Software Toolchain (VST).

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Automated reasoning; Software and its engineering → Automatic programming

Keywords and phrases Program Synthesis, Separation Logic, Proof Search

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.2

Category Invited Talk

Funding *Nadia Polikarpova*: This work was supported by the National Science Foundation under Grant No. 1911149.

Bounded-Deducibility Security

Andrei Popescu
Department of Computer Science, University of Sheffield, UK

Thomas Bauereiss
Department of Computer Science and Technology, University of Cambridge, UK

Peter Lammich
Department of Computer Science, University of Twente, The Netherlands

Abstract

We describe Bounded-Deducibility (BD) security, an expressive framework for the specification and verification of information-flow security. The framework grew by confronting concrete challenges of specifying and verifying fine-grained confidentiality properties in some realistic web-based systems. The concepts and theorems that constitute this framework have an eventful history of such "confrontations", often involving trial and error, which are reported in previous papers. This paper is the first to focus on the framework itself rather than the case studies, gathering in one place all the abstract results about BD security.

2012 ACM Subject Classification Security and privacy → Formal security models; Security and privacy → Logic and verification; Security and privacy → Security requirements

Keywords and phrases Information-flow security, Unwinding proof method, Compositionality, Verification

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.3

Category Invited Paper

Funding The work presented here has been supported by: EPSRC through the grant "Verification of Web-based Systems (VOWS)" (EP/N019547/1); DFG through the grants "Security Type Systems and Deduction" (Ni 491/13-2) and "MORES – Modelling and Refinement of Security Requirements on Data and Processes (Hu 737/5-2)", part of "RS^3 – Reliably Secure Software Systems" (SPP 1496); VeTSS through the grant "Formal Verification of Information Flow Security for Relational Databases"; Innovate UK through the Knowledge Transfer Partnership 010041 between Caritas Anchor House and Middlesex University: "The Global Noticeboard (GNB): a verified social media platform with a charitable, humanitarian purpose".

Acknowledgements We are fortunate to have collaborated with some excellent researchers and developers on various parts of the implementation and verification work based on BD security: Sergey Grebenshchikov, Ping Hou, Sudeep Kanav and Ondřej Kunčar have contributed to CoCon, while Armando Pesenti Gritti and Franco Raimondi have contributed to CoSMed and CoSMeDis. We thank this paper's reviewers for their helpful comments and suggestions.

1 Introduction

Bounded-Deducibility (BD) security is a framework we have developed recently for the specification and verification of information-flow security. It is applicable widely, to systems described as nondeterminisic I/O automata, and caters for the fine-grained specification of restrictions on their flows of information. We formalized the framework in the proof assistant Isabelle/HOL [31, 32] and used it in the verification of confidentiality properties of some web applications.

Information-flow security has a rich history, with many formal definitions having been proposed, differing in how systems, attackers, and flow policies are modeled [18, 26, 29, 30, 33, 34, 42–44, 46, 47]. Nevertheless, a new notion seemed necessary because the existing notions (Section 4) were not expressive enough for our case studies: multi-user web-based systems with flows of information requiring fine-grained control. For example, about a multi-user multi-conference management system, we wanted to prove a property such as the following, which refers to the series of uploads of a document's versions in a system: "A group of users learn nothing about a paper beyond the absence of any upload unless one of them becomes an author of that paper or a PC member at the paper's conference." (Importantly, the property is about not only what can be *directly accessed*, but also what can be *learned* by interacting with the system – this distinguishes information-flow control from mere access control.) Every abstract definition and theorem in the BD security framework was inspired from, and refined based on, the needs of concrete interactive systems. This ended up contributing to the area of information-flow security an increased level of precision in specification and proof, of the kind that we believe can make a difference in practical system verification.

In previous papers [7–9, 23, 37], BD security has only been discussed in the context of verifying these concrete systems. This helps with intuition and motivation, but makes it easy to miss the forest from the trees, i.e., miss the abstract level of the development. The current paper is the first to collect in one place all our abstract results, and to present them independently of any case studies (Section 2). They include the BD unwinding proof method (Section 2.5), as well as theorems on proof (Section 2.6) and system compositionality (Section 2.7). We hope that this paper will better demonstrate the scope of the framework and help identify potential new applications. The framework is open-ended and open-source [10, 35], and new contributions are welcome.

Three major verification case studies will also be briefly described while recalling their contribution to the framework's design (Section 3). These are the CoCon conference management system (Section 3.1, [23, 37]), the CoSMed social media platform (Section 3.2, [7, 9]), and the CoSMeDis distributed extension of CoSMed (Section 3.3, [8]).

Notations

We write function application by juxtaposition, without placing the argument in parentheses, as in $f\ a$, unless required for disambiguation, e.g., $f\ (g\ a)$. Multiple-argument functions will usually be considered in curried form – e.g., we think of $f : A \to B \to C$ as a two-argument function, and $f\ a\ b$ denotes its application to a and b. We write "$\circ$" for function composition. Bool denotes the two-element set of Booleans, $\{\mathsf{true}, \mathsf{false}\}$. Predicates and relations will be modeled as functions to Bool. For example, $P : A \to \mathsf{Bool}$ is a (unary) predicate on A and $Q : A \to A \to \mathsf{Bool}$ is a binary relation on A. Given $a \in A$, we write "$P\ a$ holds", or simply "$P\ a$", to mean that $P\ a = \mathsf{true}$; and similarly for binary relations.

Given a set A, we write $\mathsf{Set}(A)$ for the powerset (i.e., set of all subsets) of A, and $\mathsf{List}(A)$ for the set of lists with elements in A. We write $[a_1, \ldots, a_n]$ for the list consisting of the indicated elements; in particular, $[]$ is the empty list and $[a]$ is a singleton list. As a general convention, if a, b denote elements in A, then al, bl will denote elements in $\mathsf{List}(A)$. An exception will be the system traces – even though they are lists of transitions t, for them we will use the customized notation tr. We write "$\cdot$" for list concatenation. Applied to a non-empty list $[a_1, \ldots, a_n]$, the function head returns its first element a_1. Given a function $f : A \to B$ and $[a_1, \ldots, a_n] \in \mathsf{List}(A)$, $\mathsf{map}\ f\ [a_1, \ldots, a_n]$ returns $[f\ a_1, \ldots, f\ a_n]$. Given a partial function $f : A \rightharpoonup B$ and $[a_1, \ldots, a_n] \in \mathsf{List}(A)$, let $[a_{i_1}, a_{i_2}, \ldots, a_{i_k}]$ be the sublist of $[a_1, \ldots, a_n]$ that keeps only elements on which f is defined (where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$); then $\mathsf{map}\ f\ [a_1, \ldots, a_n]$

returns $[f\ a_{i_1}, \dots, f\ a_{i_k}]$. In other words, partial functions are mapped while omitting the elements on which they are not defined. Given a predicate P, $\mathsf{filter}\ P\ [a_1, \dots, a_n]$ returns the sublist of $[a_1, \dots, a_n]$ that keeps only the elements satisfying P.

2 Specification and Reasoning Framework

Our framework is developed around a simple and general notion of system: nondeterministic I/O automata. It also provides a notion of policy to describe the (dis)allowed flows of information in these systems. A policy has several parameters that regulate the tension between observations (what can be seen) and secrets (what needs to be protected). The judicious use of these parameters allows fine-tuning not only *what*, but also *how much* needs to be protected, and *when*, or even *for how long*. The framework offers methods to prove that the policies are satisfied by systems, and to manage proof and system complexity via compositionality results.

2.1 System model

The systems whose information-flow security properties will be studied are nondeterministic I/O automata. Namely, we call *system* a tuple $\mathcal{A} = (\mathsf{State}, \mathsf{Act}, \mathsf{Out}, \mathsf{istate}, \mathsf{Trans})$, where:

- State, ranged over by σ, σ' etc., is the set of states;
- Act, ranged over by a, b etc., is the set of actions;
- Out, ranged over by ou, ou' etc., is the set of outputs;
- $\mathsf{istate} \in \mathsf{State}$ is the initial state;
- $\mathsf{Trans} \subseteq \mathsf{State} \times \mathsf{Act} \times \mathsf{Out} \times \mathsf{State}$ is the set of transitions.

(Note that we call "action" what is usually called "input" for I/O automata.) A transition $t = (\sigma, a, ou, \sigma') \in \mathsf{Trans}$ has the following interpretation: If action a is taken while the system is in state σ, the system may respond by producing output ou and changing the state to σ'. We call σ the source, a the action, ou the output, and σ' the target of t. The transition's action a is also denoted by $\mathsf{actOf}\ t$. We will write $\sigma \xRightarrow{t} \sigma'$ to express that $t \in \mathsf{Trans}$, σ is the source of t and σ' is the target of t.

A *trace* is any non-empty list of transitions $[t_1, \dots, t_n]$ such that the source of t_1 is istate and, for all $i \in \{2, \dots, n\}$, the source of t_i is the target of t_{i-1}. We let Trace, ranged over by tr, be the set of traces. A *trace fragment* has the form $[t_i, \dots, t_j]$ with $1 \le i < j \le n$, where $[t_1, \dots, t_n]$ is a trace. We write TraceF_σ for the set of trace fragments that start in σ, i.e., have σ as the source of their first transition. Note that all these concepts are relative to a system $\mathcal{A}$. When we want to emphasize the underlying system, we may write $\mathsf{Trace}_\mathcal{A}$ instead of Trace, $\mathsf{TraceF}_{\mathcal{A},\sigma}$ instead of TraceF_σ, etc.

2.2 Flow policies

Given a system $\mathcal{A} = (\mathsf{State}, \mathsf{Act}, \mathsf{Out}, \mathsf{istate}, \mathsf{Trans})$, our goal is to express its information-flow security via policies that are capable of fine-grained distinctions between desirable flows (which are important for the system's functionality) and undesirable flows (which constitute information leaks possibly exploitable by attackers). To achieve such surgical precision, a policy should accurately identify the following: (1) What observations can be made on the system, (2) Which data constitute secrets that need protection, (3) How much of these secrets should be protected (and how much can be revealed), and (4) Under which conditions protection is required.

For accommodating these requirements, we define a *flow policy* $\mathcal{F}$ to consist of:

(1) an *observation infrastructure* $(\mathsf{Obs}, \mathsf{isObs}, \mathsf{getObs})$, where
- Obs, ranged over by o, o' etc., is a chosen domain of observations,
- $\mathsf{isObs} : \mathsf{Trans} \to \mathsf{Bool}$ is a predicate identifying observation-producing transitions,
- $\mathsf{getObs} : \mathsf{Trans} \to \mathsf{Obs}$ is a function for producing observations from transitions;

(2) a *secrecy infrastructure* $(\mathsf{Sec}, \mathsf{isSec}, \mathsf{getSec})$, where
- Sec, ranged over by s, s' etc., is a chosen domain of secrets,
- $\mathsf{isSec} : \mathsf{Trans} \to \mathsf{Bool}$ is a predicate identifying secret-producing transitions,
- $\mathsf{getSec} : \mathsf{Trans} \to \mathsf{Sec}$ is a function for producing secrets from transitions;

(3) a *declassification bound*, i.e., a relation on lists of secrets, $\mathsf{B} : \mathsf{List}(\mathsf{Sec}) \to \mathsf{List}(\mathsf{Sec}) \to \mathsf{Bool}$;

(4) a *declassification trigger*, i.e., a predicate on transitions, $\mathsf{T} : \mathsf{Trans} \to \mathsf{Bool}$.

Note that the observation and secrecy infrastructures have the same form. We define $\mathsf{O} : \mathsf{Trace} \to \mathsf{List}(\mathsf{Obs})$ by $\mathsf{O} = \mathsf{map\ getObs} \circ \mathsf{filter\ isObs}$, and $\mathsf{S} : \mathsf{Trace} \to \mathsf{List}\,(\mathsf{Sec})$ by $\mathsf{S} = \mathsf{map\ getSec} \circ \mathsf{filter\ isSec}$. Thus, O uses filter to select the transitions in a trace that are observable according to isObs, and then applies getObs to each selected transition. Similarly, S produces lists of secrets by filtering with isSec and applying getSec. Thus, when applied to a trace tr, O and S give the lists of observations and respectively secrets produced by tr.

2.3 Bounded-Deducibility security

For the rest of Section 2, let us fix a system $\mathcal{A} = (\mathsf{State}, \mathsf{Act}, \mathsf{Out}, \mathsf{istate}, \mathsf{Trans})$ and a flow policy $\mathcal{F}$, where $(\mathsf{Obs}, \mathsf{isObs}, \mathsf{getObs})$ is its observation infrastructure, $(\mathsf{Sec}, \mathsf{isSec}, \mathsf{getSec})$ its secrecy infrastructure, B its declassification bound and T its declassification trigger. Furthermore, let $\mathsf{O} : \mathsf{Trace} \to \mathsf{List}(\mathsf{Obs})$ and $\mathsf{S} : \mathsf{Trace} \to \mathsf{List}(\mathsf{Sec})$ be the functions on traces induced by these observation and secrecy infrastructures.

A system $\mathcal{A}$ is said to be *Bounded-Deducibility (BD) secure with respect to the flow policy* $\mathcal{F}$, written $\mathcal{A} \models \mathcal{F}$, provided that for all $tr_1 \in \mathsf{Trace}$ and $sl_1, sl_2 \in \mathsf{List}(\mathsf{Sec})$,
- if $\mathsf{never}\ \mathsf{T}\ tr_1$, $\mathsf{S}\ tr_1 = sl_1$ and $\mathsf{B}\ sl_1\ sl_2$,
- then there exists $tr_2 \in \mathsf{Trace}$ such that $\mathsf{O}\ tr_2 = \mathsf{O}\ tr_1$ and $\mathsf{S}\ tr_2 = sl_2$.

The predicate $\mathsf{never}\ \mathsf{T}\ tr_1$ says that T holds for no transition in tr_1.

Here is how to interpret the above definition: tr_1 is a trace that occurs when running the system, and sl_1 is the list of secrets that it produces. BD security says that, if the trigger T is never fired during tr_1, it is impossible for an observer (potential attacker) to distinguish tr_1 from any other trace tr_2 that produces some secrets sl_2 that are B-related to (i.e., located within bound B from) sl_1. Hence, for all the observer knows (via the observation function O), the trace tr_1 might as well have been tr_2.

When referring to the items in this definition, we will call tr_1 "the original trace" and tr_2 "the alternative trace". We will also apply the qualifiers "original" and "alternative" to the produced lists of observations and secrets. Note that BD security is a $\forall\exists$-statement: quantified universally over the original trace tr_1 and the alternative secrets sl_2, and then existentially over the alternative trace tr_2. (The universal quantification over sl_1 is done only for clarity; it can be avoided, since $sl_1 = \mathsf{S}\ tr_1$.)

We can think of B negatively, as a lower bound for uncertainty, or positively, as an upper bound for the amount of information release, also known as *declassification*. For example, if B is an equivalence, then the observers learn the equivalence class of the secret, but nothing more. On the other hand, T is a trigger removing the bound B: As soon as T becomes true, the containment of declassification is no longer guaranteed. In summary, BD security says: An observer O cannot learn about the secrets anything *beyond* B *unless* T occurs.

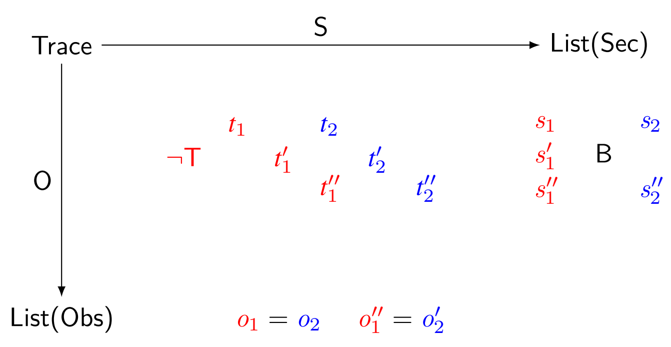


Figure 1 BD security illustrated.

Fig. 1 contains a visual illustration of BD security's two-dimensional nature: The system traces (displayed on the top left corner) produce observations (on the bottom left), as well as secrets (on the top right). The figure also includes an abstract example of traces and their observation and secret projections. The original trace tr_1 consists of three transitions, $tr_1 = [t_1, t'_1, t''_1]$, of which all produce secrets, $[s_1, s'_1, s''_1]$, and only the first and the third produce observations, $[o_1, o''_1]$ – all these are depicted in red. The alternative trace tr_2 also consists of three transitions, $tr_2 = [t_2, t'_2, t''_2]$, of which the first and the third produce secrets, $[s_2, s''_2]$, and the first two produce observations, $[o_2, o'_2]$ – all these are depicted in blue. Thus, the figure's functions O and S are given by filters and producers behaving as follows:

	isObs	getObs	isSec	getSec
t_1	true	o_1	true	s_1
t'_1	false		true	s'_1
t''_1	true	o''_1	true	s''_1

	isObs	getObs	isSec	getSec
t_2	true	o_2	true	s_2
t'_2	true	o'_2	false	
t''_2	false		true	s''_2

The empty slots in the tables correspond to values of getObs and getSec that are irrelevant, since the corresponding values of isObs and isSec are false. The ∀∃ statement expressing BD security is illustrated on the figure by making a choice of the ∀-quantified entities and the ∃-quantified entities: Given the original trace, here $[t_1, t'_1, t''_1]$ (which produces the shown observations and secrets and has all its transitions satisfying ¬T) and given some alternative secrets, here $[s_2, s''_2]$, located within the bound B of the original secrets, BD security requires the existence of the alternative trace, here $[t_2, t'_2, t''_2]$, producing the same observations and producing the alternative secrets.

2.4 From nondeducibility to bounded deducibility

BD security is a natural evolution of the idea of *nondeducibility* introduced in pioneering work by Sutherland [46]: by refining the notion of "nothing being deducible" to that of "nothing being deducible beyond a certain bound and unless a certain trigger occurs".

Indeed, nondeducibility can be expressed in terms of operators O : Trace → List(Obs) and S : Trace → List(Sec) by requiring that, for all $tr_1 \in$ Trace and $sl_1, sl_2 \in$ List(Sec), if S $tr_1 = sl_1$ then there exists $tr_2 \in$ Trace such that O $tr_2 =$ O tr_1 and S $tr_2 = sl_2$. Thus, BD security becomes nondeducibility when B is everywhere true and T everywhere false – meaning no declassification, i.e., maximum uncertainty.

2.5 Unwinding proof method

To prove that the system is BD secure with respect to the flow policy, $\mathcal{A} \models \mathcal{F}$, one needs to do the following: Given

- the original trace tr_1 for which $\mathsf{never}\ \mathsf{T}$ holds and which produces the list of secrets sl_1,
- and an alternative list of secrets sl_2 such that $\mathsf{B}\ sl_1\ sl_2$ holds,

one should provide an alternative trace tr_2 whose produced list of secrets is exactly sl_2 and whose produced list of observations is the same as that of tr_1.

Following the tradition of unwinding for noninterference-like properties [19,26,41], we want to construct tr_2 from tr_1 incrementally: As tr_1 grows, tr_2 should grow nearly synchronously. Unwindings are traditionally binary relations Δ on State that bookkeep the states reached by tr_1 and tr_2, say σ_1 and σ_2, and show how these can evolve transition by transition in the process of constructing tr_2 from tr_1; they guarantee that any Δ-related states σ_1 and σ_2 evolve via transitions $\sigma_1 \xRightarrow{t_1} \sigma_1'$ and $\sigma_2 \xRightarrow{t_2} \sigma_2'$ to Δ-related states σ_1' and σ_2'. In our case, unlike in the traditional case, we have a significantly more complex infrastructure to deal with: Since the produced observations of tr_1 and tr_2 will have to be equal, it is reasonable to track them synchronously; but the produced secrets are regulated by arbitrary bounds B, hence they will have to track them more flexibly.

To address the above, an unwinding for BD security will be not just a binary relation between states, but a binary relation between pairs consisting of a state and a list of secrets. Let us introduce some convenient notation to describe this. For any pairs (σ, sl) and (σ', sl') in $\mathsf{State} \times \mathsf{List}(\mathsf{Sec})$ and any transition t, we will write $(\sigma, sl) \xRightarrow{t} (\sigma', sl')$ as a shorthand for the following two statements: (1) $\sigma \xRightarrow{t} \sigma'$, and (2) either $\neg\ \mathsf{isSec}\ t$ and $sl' = sl$, or $\mathsf{isSec}\ t$ and there exists s such that $sl = [s] \cdot sl'$. The second statement means that the transition t either does not produce a secret thus leaving sl unchanged ($sl' = sl$), or produces the secret from the beginning of sl thus reducing it to sl'; we can think of this as a transition between lists of secrets that are still to be produced. Moreover, for any two transitions t_1 and t_2, we will write $t_1 =_{\mathsf{Obs}} t_2$ as a shorthand for the following two statements: (1) $\mathsf{isObs}\ t_1$ if and only if $\mathsf{isObs}\ t_2$, and (2) if $\mathsf{isObs}\ t_1$ then $\mathsf{getObs}\ t_1 = \mathsf{getObs}\ t_2$. In other words, t_1 and t_2 produce either the same observation or no observation.

A relation $\Delta : (\mathsf{State} \times \mathsf{List}(\mathsf{Sec})) \to (\mathsf{State} \times \mathsf{List}(\mathsf{Sec})) \to \mathsf{Bool}$ is said to be a *BD unwinding* if, for all (σ_1, sl_1), $(\sigma_2, sl_2) \in \mathsf{State} \times \mathsf{List}(\mathsf{Sec})$ such that σ_1 is $(\neg\mathsf{T})$-reachable, σ_2 is reachable and $\Delta\ (\sigma_1, sl_1)\ (\sigma_2, sl_2)$, we have that one of the following three cases holds:

(1) $sl_1 \neq []$ or $sl_2 = []$, and $\mathsf{reaction}\ \Delta\ (\sigma_1, sl_1)\ (\sigma_2, sl_2)$; or

(2) $\mathsf{iaction}\ \Delta\ (\sigma_1, sl_1)\ (\sigma_2, sl_2)$; or

(3) $sl_1 \neq []$ and $\mathsf{exit}\ \sigma_1\ (\mathsf{head}\ sl_1)$.

Above, a state being reachable means that there exists a trace tr leading to it; and $(\neg\mathsf{T})$-reachability additionally requires that all transitions in tr satisfy $\neg\mathsf{T}$.

The predicates $\mathsf{reaction}$, $\mathsf{iaction}$ (read "independent action") and exit will be defined below. The first two describe possible evolution patterns for the pairs (σ_1, sl_1) and (σ_2, sl_2) so that the result is still in Δ. By contrast, the exit predicate provides a shortcut for an early finish during a proof by unwinding. When reading the definitions of these predicates, the reader should keep in mind what we want from a BD unwinding: to manage the incremental growth of an alternative trace (that has currently reached state σ_2), in response to the growth of an original trace (that has currently reached state σ_1), while considering the list of secrets sl_1 that the remainder of the original trace *is assumed to produce* and the list of secrets sl_2 that the remainder of the alternative trace *will have to produce.*

$\mathsf{reaction}\ \Delta\ (\sigma_1, sl_1)\ (\sigma_2, sl_2)$ is defined to mean that, for all $t_1 \in \mathsf{Trans}$ and $(\sigma_1', sl_1') \in \mathsf{State} \times \mathsf{List}(\mathsf{Sec})$ such that $(\sigma_1, sl_1) \xrightarrow{t_1} (\sigma_1', sl_1')$, one of the following two cases holds:

(1) $\neg\ \mathsf{isObs}\ t_1$ and $\Delta\ (\sigma_1', sl_1')\ (\sigma_2, sl_2)$; or

(2) there exist $t_2 \in \mathsf{Trans}$ and $(\sigma_2', sl_2') \in \mathsf{State} \times \mathsf{List}(\mathsf{Sec})$ such that $(\sigma_2, sl_2) \xrightarrow{t_2} (\sigma_2', sl_2')$, $t_1 =_{\mathsf{Obs}} t_2$ and $\Delta\ (\sigma_1', sl_1')\ (\sigma_2', sl_2')$.

Thus, $\mathsf{reaction}\ \Delta\ (\sigma_1, sl_1)\ (\sigma_2, sl_2)$ describes two ways in which one can "react" to a transition t_1 taken by the original trace: (1) either ignoring it (if it is unobservable), or (2) matching it with a transition t_2 of the alternative trace. In both cases, we must stay in Δ.

$\mathsf{iaction}\ \Delta\ (\sigma_1, sl_1)\ (\sigma_2, sl_2)$ is defined to mean that there exist $t_2 \in \mathsf{Trans}$ and $(\sigma_2', sl_2') \in \mathsf{State} \times \mathsf{List}(\mathsf{Sec})$ such that $(\sigma_2, sl_2) \xrightarrow{t_2} (\sigma_2', sl_2')$, $\neg\ \mathsf{isObs}\ t_2$, $\mathsf{isSec}\ t_2$ and $\Delta\ (\sigma_1, sl_1)\ (\sigma_2', sl_2')$.

Thus, $\mathsf{iaction}$ describes the possibility of an "independent" (i.e., non-reactive) action by taking an unobservable secret-producing transition in the alternative trace. While the unobservability requirement ($\neg\ \mathsf{isObs}\ t_2$) is justified by the desire to keep the observations synchronized, the reason for the secret-producing requirement ($\mathsf{isSec}\ t_2$) is more subtle: Repeating unobservable *and non-secret-producing* independent actions could indefinitely delay the growth of the original trace while making no progress with the alternative list of secrets, rendering unwinding reasoning unsound.

$\mathsf{exit}\ \sigma\ s$ is defined to mean that, for all states σ' that are $(\neg\mathsf{T})$-reachable from σ and all transitions t with source σ' such that $\neg\ \mathsf{T}\ t$, if $\mathsf{isSec}\ t$ then $\mathsf{getSec}\ t \neq s$.

The idea behind exit is that BD security holds trivially for original traces that are unable to produce their due list of secrets sl_1; and exit detects this (thus closing that branch of the unwinding proof) by noticing that not even the first secret in sl_1 can be produced starting from the current state σ_1 – indeed, in the definition of unwinding, exit is invoked with σ_1 and $\mathsf{head}\ sl_1$.

Left unexplained so far are the (non)emptiness conditions guarding the invocations of the $\mathsf{reaction}$ and exit predicates in the definition of BD unwinding. For exit, it is obvious that we need $sl_1 \neq []$ for talking about the first element in sl_1. But for $\mathsf{reaction}$, why require that $sl_1 \neq []$ or $sl_2 = []$? Again, this decision has to do with the soundness of BD unwinding as a proof method: If the negation of this condition is true, it means that the original trace is done with producing its secrets ($sl_1 = []$) and the alternative trace still has some secrets to produce ($sl_2 \neq []$). In that case, we want to enforce an $\mathsf{iaction}$ move which, being secret-producing, would make progress through the remaining alternative list of secrets sl_2; this is achieved by preventing a $\mathsf{reaction}$ move, which would be the only alternative (since an exit move needs $sl_1 \neq []$). With these definitions, BD unwinding fulfills its goal:

▶ **Lemma 1.** [23, 37] Assume Δ is a BD unwinding and let $\sigma_1, \sigma_2 \in \mathsf{State}$ such that $\mathsf{reach}_{\neg\mathsf{T}}\ \sigma_1$ and $\mathsf{reach}\ \sigma_2$. Then, for all $tr_1 \in \mathsf{TraceF}_{\sigma_1}$ and $sl_1, sl_2 \in \mathsf{List}(\mathsf{Sec})$,

- if $\mathsf{never}\ \mathsf{T}\ tr_1$, $\mathsf{S}\ tr_1 = sl_1$ and $\Delta\ (\sigma_1, sl_1)\ (\sigma_2, sl_2)$,
- then there exists $tr_2 \in \mathsf{TraceF}_{\sigma_2}$ such that $\mathsf{O}\ tr_2 = \mathsf{O}\ tr_1$ and $\mathsf{S}\ tr_2 = sl_2$.

In other words, assuming $\Delta\ (\sigma_1, sl_1)\ (\sigma_2, sl_2)$ holds and given the remaining part tr_1 of the original trace (starting in σ_1) which produces secrets sl_1, there exists a trace tr_2 that produces the same observations and produces the desired secrets sl_2. The lemma's proof goes by induction on the sum of the lengths of tr_1 and sl_2. The induction step either reaches a contradiction (if exit is invoked), or consumes a transition from tr_1 (if $\mathsf{reaction}$ is invoked) or a secret from sl_2 (if $\mathsf{iaction}$ is invoked).

To connect this result to BD security, in particular to factor in the bound B as well, we additionally require that a BD unwinding Δ includes the bound B in the initial state. So we can think of Δ as generalizing and strengthening the bound, and then maintaining it all the way to the successful production of the alternative trace required by BD security. We are closing in on the main result about BD unwinding, a consequence of the lemma taking $\sigma_1 = \sigma_2 = \mathsf{istate}$. It states that BD unwinding is a *sound proof method* for BD security.

▶ **Theorem 2.** (Unwinding Theorem [23, 37]) Assume that the following hold:
(1) For all $sl_1, sl_2 \in \mathsf{List(Sec)}$, if $\mathsf{B}\ sl_1\ sl_2$ then $\Delta\ (\mathsf{istate}, sl_1)\ (\mathsf{istate}, sl_2)$.
(2) Δ is a BD unwinding.
Then $\mathcal{A} \models \mathcal{F}$.

According to this theorem, to prove BD security of a system, it suffices to define a relation Δ and show that (1) it includes the bound B in the initial state and (2) it is a BD unwinding.

2.6 Proof compositionality

When verifying a BD security policy for a large system, defining a single monolithic BD unwinding could be daunting. We can alleviate this by working not with a single unwinding relation, but with a network of relations, such that any relation may “unwind” into any number of relations in the network.

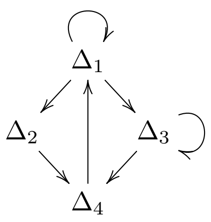
To this end, we refine the notion of BD unwinding. Given a relation Δ and a set of relations $\Delta\mathsf{s}$, Δ is a said to be a *BD unwinding into* $\Delta\mathsf{s}$ if it satisfies the same conditions as in the definition of BD unwinding, just that $\mathsf{iaction}\ \Delta$ and $\mathsf{reaction}\ \Delta$ are replaced by $\mathsf{iaction}\ (\bigvee \Delta\mathsf{s})$ and $\mathsf{reaction}\ (\bigvee \Delta\mathsf{s})$, where $\bigvee \Delta\mathsf{s}$ is the disjunction (i.e., union) of all the relations in $\Delta\mathsf{s}$. Namely, for all (σ_1, sl_1), $(\sigma_2, sl_2) \in \mathsf{State} \times \mathsf{List(Sec)}$ such that σ_1 is $(\neg\mathsf{T})$-reachable, σ_2 is reachable and $\Delta\ (\sigma_1, sl_1)\ (\sigma_2, sl_2)$, one of the following three cases holds:
(1) $sl_1 \neq []$ or $sl_2 = []$, and $\mathsf{reaction}\ (\bigvee \Delta\mathsf{s})\ (\sigma_1, sl_1)\ (\sigma_2, sl_2)$; or
(2) $\mathsf{iaction}\ (\bigvee \Delta\mathsf{s})\ (\sigma_1, sl_1)\ (\sigma_2, sl_2)$; or
(3) $sl_1 \neq []$ and $\mathsf{exit}\ \sigma_1\ (\mathsf{head}\ sl_1)$.

This enables a form of sound compositional reasoning: If we verify a condition as above for each component relation, we obtain an overall secure system.

▶ **Theorem 3.** (Multiplex Unwinding Theorem [37]) Let $\Delta\mathsf{s}$ be a set of relations. For each $\Delta \in \Delta\mathsf{s}$, let $\mathsf{next}_\Delta \subseteq \Delta\mathsf{s}$ be a (possibly empty) set of “successors” of Δ, and let $\Delta_{\mathsf{init}} \in \Delta\mathsf{s}$ be a chosen “initial” relation. Assume the following hold:
(1) For all $sl_1, sl_2 \in \mathsf{List(Sec)}$, if $\mathsf{B}\ sl_1\ sl_2$ then $\Delta_{\mathsf{init}}\ (\mathsf{istate}, sl_1)\ (\mathsf{istate}, sl_2)$.
(2) Each $\Delta \in \Delta\mathsf{s}$ is a BD unwinding into next_Δ.
Then $\mathcal{A} \models \mathcal{F}$.

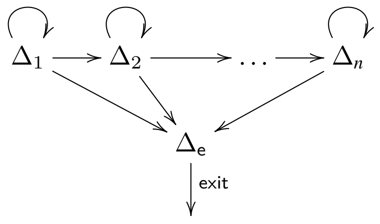
The network of components can form any directed graph – Fig. 2 shows an example. However, when doing concrete proofs by unwinding, we found that the following essentially linear network often suffices (Fig. 3): Each Δ_i unwinds either into itself, or into Δ_{i+1} (if $i \neq n$), or into an exit component Δ_{e} that always chooses the “exit” unwinding condition. (In practice, Δ_{e} will collect “error” situations that break invariants, hence preventing the original trace from producing its due secrets.) To express this, we define the notion of Δ being a *BD continuation-unwinding into* $\Delta\mathsf{s}$ similarly to that of “BD unwinding into” but excluding the exit case, i.e., requiring that either (1) $sl_1 \neq []$ or $sl_2 = []$, and $\mathsf{reaction}\ (\bigvee \Delta\mathsf{s})\ (\sigma_1, sl_1)\ (\sigma_2, sl_2)$, or (2) $\mathsf{iaction}\ (\bigvee \Delta\mathsf{s})\ (\sigma_1, sl_1)\ (\sigma_2, sl_2)$ hold. And Δ is said to be a *BD exit-unwinding* if the exit case, (3) $sl_1 \neq []$ and $\mathsf{exit}\ \sigma_1\ (\mathsf{head}\ sl_1)$, holds. We obtain:

▶ **Theorem 4.** (Sequential Multiplex Unwinding Theorem [37]) Consider the indexed set of relations $\{\Delta_1, \ldots, \Delta_n\}$ and the relation Δ_{e} such that the following hold:



$$\begin{tikzcd} & \Delta_1 \arrow[loop above] \arrow[dl] \arrow[dr] & \\ \Delta_2 \arrow[dr] & & \Delta_3 \arrow[loop right] \arrow[dl] \\ & \Delta_4 \arrow[uu] & \end{tikzcd}$$

Figure 2 A network of unwinding components.



$$\begin{tikzcd} \Delta_1 \arrow[loop above] \arrow[r] \arrow[drr] & \Delta_2 \arrow[loop above] \arrow[r] \arrow[dr] & \ldots \arrow[r] & \Delta_n \arrow[loop above] \arrow[dl] \\ & & \Delta_e \arrow[d, "\text{exit}"] & \\ & & {} & \end{tikzcd}$$

Figure 3 A linear network with exit.

(1) For all $sl_1, sl_2 \in \mathsf{List}(\mathsf{Sec})$, if $\mathsf{B}\ sl_1\ sl_2$ then $\Delta_1\ (\mathsf{istate}, sl_1)\ (\mathsf{istate}, sl_2)$.
(2) Δ_i is a BD continuation-unwinding into $\{\Delta_i, \Delta_{i+1}, \Delta_e\}$.
(3) Δ_e is a BD exit-unwinding.
Then $\mathcal{A} \models \mathcal{F}$.

Although the Multiplex Unwinding Theorems are easy consequences of the (plain) Unwinding Theorem, we found them to be very useful tools for managing proof complexity.

2.7 System compositionality

A complexity management desideratum equally important to proof compositionality is system compositionality: the possibility to infer BD security for a compound system from BD security of the components. Next, we will describe a compositionality result for a communicating network of systems. We start with two, then we generalize to n systems.

2.7.1 Product systems

Let $\mathcal{A}_1 = (\mathsf{State}_1, \mathsf{Act}_1, \mathsf{Out}_1, \mathsf{istate}_1, \mathsf{Trans}_1)$ and $\mathcal{A}_2 = (\mathsf{State}_2, \mathsf{Act}_2, \mathsf{Out}_2, \mathsf{istate}_2, \mathsf{Trans}_2)$ be two systems. We want to model communication between $\mathcal{A}_1$ and $\mathcal{A}_2$ by matching certain transitions that these systems must take synchronously while exchanging data. This is captured by a relation $\mathsf{match} : \mathsf{Trans}_1 \to \mathsf{Trans}_2 \to \mathsf{Bool}$. Transition matching gives a very flexible communication scheme: It can model message-passing communication using the transitions' actions and outputs, but also shared-state communication using the transitions' source and target states.

We will distinguish between separate (local) component actions and communication actions. We write $\mathsf{isCom}_i\ a$ (for $i \in \{1, 2\}$) to indicate that an action a is in the latter category for $\mathcal{A}_i$. Namely, $\mathsf{isCom}_i\ a$ holds whenever there exist t_1 and t_2 such that $\mathsf{match}\ t_1\ t_2$ holds and a is the action of t_i.

We define the match-*communicating product of* $\mathcal{A}_1$ *and* $\mathcal{A}_2$, written $\mathcal{A}_1 \times^{\mathsf{match}} \mathcal{A}_2$, as the following system $(\mathsf{State}, \mathsf{Act}, \mathsf{Out}, \mathsf{istate}, \mathsf{Trans})$:

- $\mathsf{State} = \mathsf{State}_1 \times \mathsf{State}_2$;
- $\mathsf{Act} = \mathsf{Act}_1 + \mathsf{Act}_2 + \mathsf{Act}_1 \times \mathsf{Act}_2$; thus, Act is a disjoint union of Act_1 (representing separate actions of the first component), Act_2 (for separate actions of the second component), and $\mathsf{Act}_1 \times \mathsf{Act}_2$ (for joint communicating actions); we write $(1, a_1)$, $(2, a_2)$, and (a_1, a_2) for actions of the first, second and third kind, respectively;
- $\mathsf{Out} = \mathsf{Out}_1 + \mathsf{Out}_2 + \mathsf{Out}_1 \times \mathsf{Out}_2$; thus, like Act, Out is a disjoint union, and we use similar notations for its elements: $(1, ou_1)$, $(2, ou_2)$ and (ou_1, ou_2);
- $\mathsf{istate} = (\mathsf{istate}_1, \mathsf{istate}_2)$;
- Trans contains three kinds of transitions:

- separate $\mathcal{A}_1$-transitions $((\sigma_1, \sigma_2), (1, a_1), (1, ou_1), (\sigma_1', \sigma_2))$, where $(\sigma_1, a_1, ou_1, \sigma_1') \in \mathsf{Trans}_1$ and $\neg\ \mathsf{isCom}_1\ a_1$;
- separate $\mathcal{A}_2$-transitions $((\sigma_1, \sigma_2), (2, a_2), (2, ou_2), (\sigma_1, \sigma_2'))$, where $(\sigma_2, a_2, ou_2, \sigma_2') \in \mathsf{Trans}_2$ and $\neg\ \mathsf{isCom}_2\ a_2$;
- communication transitions $((\sigma_1, \sigma_2), (a_1, a_2), (ou_1, ou_2), (\sigma_1', \sigma_2'))$, where $(\sigma_1, a_1, ou_1, \sigma_1') \in \mathsf{Trans}_1$, $(\sigma_2, a_2, ou_2, \sigma_2') \in \mathsf{Trans}_2$ and $\mathsf{match}\ (\sigma_1, a_1, ou_1, \sigma_1')\ (\sigma_2, a_2, ou_2, \sigma_2')$.

Thus, a transition t of $\mathcal{A}_1 \times^{\mathsf{match}} \mathcal{A}_2$ has exactly one of the following three forms shown above. In the first case, t is completely determined by an $\mathcal{A}_1$-transition $t_1 = (\sigma_1, a_1, ou_1, \sigma_1')$ and an $\mathcal{A}_2$-state σ_2 – we write $t = \mathsf{sep}_1\ t_1\ \sigma_2$, marking that t is given by the separate transition t_1. Similarly, in the second case we write $t = \mathsf{sep}_2\ \sigma_1\ t_2$, where $t_2 = (\sigma_2, a_2, ou_2, \sigma_2')$. In the third case, we write $t = \mathsf{com}\ t_1\ t_2$, marking that t proceeds as a communication transition. Thus, in our new notation, any transition of $\mathcal{A}_1 \times^{\mathsf{match}} \mathcal{A}_2$ has either the form $\mathsf{sep}_1\ t_1\ \sigma_2$, or $\mathsf{sep}_2\ \sigma_1\ t_2$, or $\mathsf{com}(t_1, t_2)$.

2.7.2 Product flow policies

Let $\mathcal{F}_1$ and $\mathcal{F}_2$ be flow policies for $\mathcal{A}_1$ and $\mathcal{A}_2$. Given $i \in \{1, 2\}$, we write $(\mathsf{Obs}_i, \mathsf{isObs}_i, \mathsf{getObs}_i)$ for the observation infrastructure, $(\mathsf{Sec}_i, \mathsf{isSec}_i, \mathsf{getSec}_i)$ for the secrecy infrastructure, B_i for the declassification bound and T_i for the declassification trigger of $\mathcal{F}_i$. We want to compose the policies $\mathcal{F}_1$ and $\mathcal{F}_2$ in a natural way, forming a policy for the product $\mathcal{A}_1 \times^{\mathsf{match}} \mathcal{A}_2$. To achieve this, we need observation and secret counterparts of the transition-matching predicate match, in the form of predicates $\mathsf{matchO} : \mathsf{Obs}_1 \to \mathsf{Obs}_2 \to \mathsf{bool}$ and $\mathsf{matchS} : \mathsf{Sec}_1 \to \mathsf{Sec}_2 \to \mathsf{bool}$. Triples $(\mathsf{match}, \mathsf{matchO}, \mathsf{matchS})$ will be called *communication infrastructures.*

A sanity property that we will assume about our communication infrastructures is that its matching operators are compatible with (i.e., preserved by) the secrecy and observation infrastructure operators.

Compatible Communication: For all $t_1 \in \mathsf{Trans}_1$ and $t_2 \in \mathsf{Trans}_2$, if $\mathsf{match}\ t_1\ t_2$ then:

- $\mathsf{isSec}_1\ t_1$ if and only if $\mathsf{isSec}_2\ t_2$, and in this case we have $\mathsf{matchS}\ (\mathsf{getSec}_1\ t_1)\ (\mathsf{getSec}_2\ t_2)$;
- $\mathsf{isObs}_1\ t_1$ if and only if $\mathsf{isObs}_2\ t_2$, and in this case we have $\mathsf{matchO}\ (\mathsf{getObs}_1\ t_1)\ (\mathsf{getObs}_2\ t_2)$.

The *product of* $\mathcal{F}_1$ *and* $\mathcal{F}_2$ *along a communication infrastructure* $(\mathsf{match}, \mathsf{matchO}, \mathsf{matchS})$, written $\mathcal{F}_1 \times^{(\mathsf{match},\mathsf{matchO},\mathsf{matchS})} \mathcal{F}_2$, is defined as the following flow policy for $\mathcal{A}_1 \times^{\mathsf{match}} \mathcal{A}_2$.

We start with its observation and secrecy infrastructures, which are naturally defined considering that observations and secrets can be produced either separately or in communication steps. The observation infrastructure $(\mathsf{Obs}, \mathsf{isObs}, \mathsf{getObs})$ is the following:

- $\mathsf{Obs}_1 + \mathsf{Obs}_2 + \mathsf{Obs}_1 \times \mathsf{Obs}_2$; thus, an element of Obs will have either the form $(1, o_1)$, or $(2, o_2)$, or (o_1, o_2), where $o_i \in \mathsf{Obs}_i$.
- For any $t \in \mathsf{Trans}$, $\mathsf{isObs}\ t$ and $\mathsf{getObs}\ t$ are defined as follows:
  - if t has the form $\mathsf{sep}_1\ t_1\ \sigma_2$, then $\mathsf{isObs}\ t = \mathsf{isObs}_1\ t_1$ and $\mathsf{getObs}\ t = (1, \mathsf{getObs}_1\ t_1)$;
  - if t has the form $\mathsf{sep}_2\ \sigma_1\ t_2$, then $\mathsf{isObs}\ t = \mathsf{isObs}_2\ t_2$ and $\mathsf{getObs}\ t = (2, \mathsf{getObs}_2\ t_2)$;
  - if t has the form $\mathsf{com}\ t_1\ t_2$, then $\mathsf{isObs}\ t = (\mathsf{isObs}_1\ t_1$ and $\mathsf{isObs}_2\ t_2)$ and $\mathsf{getObs}\ t = (\mathsf{getObs}_1\ t_1, \mathsf{getObs}_2\ t_2)$.

One could argue that, when t has the form $\mathsf{com}\ t_1\ t_2$, $\mathsf{isObs}\ t$ should be defined not as (1) $\mathsf{isObs}_1\ t_1$ and $\mathsf{isObs}_2\ t_2$, but as (2) $\mathsf{isObs}_1\ t_1$ or $\mathsf{isObs}_2\ t_2$, thus making the compound transition observable if either component transition is observable. However, we will only work under the assumption of Compatible Communication (introduced above), which makes (1) and (2) equivalent.

$$\textsc{Sep}_1\,\frac{sl \in sl_1 \times^{\mathsf{matchS}} sl_2 \qquad \neg\,\mathsf{isComS}_1\ s_1}{sl \cdot [(1, s_1)] \ \in\ (sl_1 \cdot [s_1]) \times^{\mathsf{matchS}} sl_2} \qquad \textsc{Sep}_2\,\frac{sl \in sl_1 \times^{\mathsf{matchS}} sl_2 \qquad \neg\,\mathsf{isComS}_2\ s_2}{sl \cdot [(2, s_2)] \ \in\ sl_1 \times^{\mathsf{matchS}} (sl_2 \cdot [s_2])}$$

$$\textsc{Empty}\,\frac{\cdot}{[] \in [] \times^{\mathsf{matchS}} []} \qquad \textsc{Com}\,\frac{sl \in sl_1 \times^{\mathsf{matchS}} sl_2 \qquad \mathsf{matchS}\ s_1\ s_2}{sl \cdot [(s_1, s_2)] \ \in\ (sl_1 \cdot [s_1]) \times^{\mathsf{matchS}} (sl_2 \cdot [s_2])}$$

Figure 4 Shuffle product for lists of secrets.

The secrecy infrastructure $(\mathsf{Sec}, \mathsf{isSec}, \mathsf{getSec})$ is defined similarly to the observation infrastructure: Sec is taken to be $\mathsf{Sec}_1 + \mathsf{Sec}_2 + \mathsf{Sec}_1 \times \mathsf{Sec}_2$, and isSec and getSec are defined correspondingly.

The trigger T of the product flow policy is also the natural one: Any firing of the trigger on either side, either separately or during communication, will fire the composite trigger. Formally, we take $\mathsf{T}\ t$ to mean the following: (1) if t has the form $\mathsf{sep}_1\ t_1\ \sigma_2$, then $\mathsf{T}_1\ t_1$ holds; (2) if t has the form $\mathsf{sep}_2\ \sigma_1\ t_2$, then $\mathsf{T}_2\ t_2$ holds; (3) if t has the form $\mathsf{com}\ t_1\ t_2$, then $\mathsf{T}_1\ t_1$ holds or $\mathsf{T}_2\ t_2$ holds.

It remains to define the bound B of the product flow policy. Let $sl \in \mathsf{List}(\mathsf{Sec})$ be a list of secrets in the composite secret domain. Intuitively, the most restrictive bound B we can hope for will forbid the declassification, for any lists of secrets $sl_1 \in \mathsf{List}(\mathsf{Sec}_1)$ and $sl_2 \in \mathsf{List}(\mathsf{Sec}_2)$ into which sl can be decomposed (i.e., which can be combined to make up sl), of anything beyond what can be declassified about sl_1 and sl_2 within the components' bounds B_1 and B_2.

To capture this, we collect all valid ways of combining sl_1 and sl_2, via the matchS-shuffle product operator $\times^{\mathsf{matchS}} : \mathsf{List}(\mathsf{Sec}_1) \to \mathsf{List}(\mathsf{Sec}_2) \to \mathsf{Set}(\mathsf{List}(\mathsf{Sec}))$ whose inductive definition is shown in Fig. 4. The set $sl_1 \times^{\mathsf{matchS}} sl_2$ contains all possible interleavings of sl_1 and sl_2, achieved by separate individual steps (rule $\textsc{Sep}_1$ and $\textsc{Sep}_2$) and communication steps (rule $\textsc{Com}$). For $i \in \{1, 2\}$, $\mathsf{isComS}_i\ s$ is the secret counterpart of the predicate isCom_i, expressing that the secret s participates in a matchS-relationship. We define $\mathsf{B}\ sl\ sl'$ to mean that, for all $sl_1, sl'_1 \in \mathsf{List}(\mathsf{Sec}_1)$ and $sl_2, sl'_2 \in \mathsf{List}(\mathsf{Sec}_2)$, if $sl \in sl_1 \times^{\mathsf{matchS}} sl_2$ and $sl' \in sl'_1 \times^{\mathsf{matchS}} sl'_2$, then $\mathsf{B}_1(sl_1, sl'_1)$ and $\mathsf{B}_2(sl_2, sl'_2)$ hold.

2.7.3 Compositionality result

We next introduce some properties that refer to the flow policies $\mathcal{F}_1$ and $\mathcal{F}_2$ and the communication infrastructure $(\mathsf{match}, \mathsf{matchO}, \mathsf{matchS})$. Together with Compatible Communication, they will be sufficient for compositionality.

Strong Communication: For all $t_1 \in \mathsf{Trans}_1$ and $t_2 \in \mathsf{Trans}_2$, if the following hold:

- $\mathsf{isCom}_1(\mathsf{actOf}_1\ t_1)$ and $\mathsf{isCom}_2\ (\mathsf{actOf}_2\ t_2)$,
- $\mathsf{isObs}_1\ t_1$, $\mathsf{isObs}_2\ t_2$ and $\mathsf{matchO}\ (\mathsf{getObs}_1\ t_1)\ (\mathsf{getObs}_2\ t_2)$,
- $\mathsf{isSec}_1\ t_1$ and $\mathsf{isSec}_2\ t_2$ imply $\mathsf{matchS}\ (\mathsf{getSec}_1\ t_1)\ (\mathsf{getSec}_2\ t_2)$,

then $\mathsf{match}\ t_1\ t_2$ holds.

The property says that, for observable communicating transitions, observation matching together with secret matching (the latter conditional on secrecy) causes the matching of the entire transitions.

Observable Communication: For all $t_1 \in \mathsf{Trans}_1$, $\mathsf{isCom}_1\ (\mathsf{actOf}_1\ t_1)$ implies $\mathsf{isObs}_1\ t_1$; and for all $t_2 \in \mathsf{Trans}_2$, $\mathsf{isCom}_2\ (\mathsf{actOf}_2\ t_2)$ implies $\mathsf{isObs}_2\ t_2$.

The property says that all communicating transitions are observable (i.e., isObs is true for them), although it does not say anything about what can actually be observed about them (via getObs).

Secret Polarization: For all $t_2 \in \mathsf{Trans}_2$, $\mathsf{isSec}_2\ t_2$ implies $\mathsf{isCom}_2\ (\mathsf{actOf}_2\ t_2)$.

The property says that any $\mathcal{A}_2$-transition that is secret-producing must be a communicating transition, which means that only $\mathcal{A}_1$ is able to produce secrets independently.

We are now ready to state our system compositionality result about BD security:

▶ **Theorem 5.** (System Compositionality Theorem [8]) Assume that the flow policies $\mathcal{F}_1$ and $\mathcal{F}_2$ and the communication infrastructure (match, matchO, matchS) satisfy all the above properties, namely Compatible, Strong and Observable Communication, and Secret Polarization. Moreover, assume $\mathcal{A}_1 \models \mathcal{F}_1$ and $\mathcal{A}_2 \models \mathcal{F}_2$. Then $\mathcal{A}_1 \times^{\mathsf{match}} \mathcal{A}_2 \ \models\ \mathcal{F}_1 \times^{(\mathsf{match},\mathsf{matchO},\mathsf{matchS})} \mathcal{F}_2$.

In [8], we discuss in great detail this theorem's assumptions in the context of verifying a concrete distributed system. The main strength of the theorem is that it allows composing general bounds and triggers. For this to work, we put restrictions on the observation and secrecy infrastructures. Among these, Compatible Communication seems to occur naturally in communicating systems – at least in our case studies of interest, which are multi-user web-based systems. When targeting such systems, Strong and Observable Communication seem to be achievable for a given desired policy via a uniform process of strengthening the observation and secrecy infrastructures: allowing one to observe as much non-sensitive information as possible, and making minor adjustments to the bounds and triggers to accommodate the additional harmless information unblocked [8, App. B].

On the other hand, Secret Polarization is the major limitation of the theorem.[1] For multi-user systems, this means that, for the notion of secret defined by the flow policies $\mathcal{F}_1$ and $\mathcal{F}_2$, only users of one of the two component systems, $\mathcal{A}_1$, can be allowed to upload secrets. However, this does not prevent us from considering another notion of secret, where the other component is the issuer, as part of a different pair of flow policies $\mathcal{F}_1'$ and $\mathcal{F}_2'$.[2]

Finally, an inconvenience of applying the theorem is the somewhat artificial nature of the composite bound. While by design the composite bound is as restrictive as possible (which is good for accuracy), in practice we would prefer a less restrictive but more readable bound, referring to secrets of a simpler nature than the composite secrets. To obtain this, we can perform an adjustment using a general-purpose theorem that transports a BD security property between different observation and secret domains, possibly loosening the bound and weakening the trigger, i.e., overall weakening the flow policy.

This works as follows. Let $\mathcal{F}$ and $\mathcal{F}'$ be two flow policies for a system $\mathcal{A}$, where we write ($\mathsf{Obs}, \mathsf{isObs}, \mathsf{getObs}$) and ($\mathsf{Obs}', \mathsf{isObs}', \mathsf{getObs}'$) for their observation infrastructures, and similarly for their secrecy infrastructures, bounds and triggers. $\mathcal{F}'$ is said to be *weaker* than $\mathcal{F}$, written $\mathcal{F}' \leq \mathcal{F}$, if there exist two partial functions $f : \mathsf{Sec} \rightharpoonup \mathsf{Sec}'$ and $g : \mathsf{Obs} \rightharpoonup \mathsf{Obs}'$ that preserve the secrecy and observation infrastructures, the bounds and the triggers, i.e., such that the following hold:

- $\mathsf{isSec}'\ t$ if and only if $\mathsf{isSec}\ t$ and f is defined on $\mathsf{getSec}\ t$, and in this case $f\ (\mathsf{getSec}\ t) = \mathsf{getSec}'\ t$;
- $\mathsf{isObs}'\ t$ if and only if $\mathsf{isObs}\ t$ and g is defined on $\mathsf{getObs}\ t$, and in this case $g\ (\mathsf{getObs}\ t) = \mathsf{getObs}'\ t$;
- $\mathsf{T}\ t$ implies $\mathsf{T}'\ t$;
- $\mathsf{B}'\ sl'\ tl'$ and $\mathsf{map}\ f\ sl = sl'$ imply that there exists tl such that $\mathsf{map}\ f\ tl = tl'$ and $\mathsf{B}\ sl\ tl$.

[1] In [8, Sec. V.8], we discuss in great detail the technical reasons for requiring Secret Polarization, which have to do with BD security favoring the under-specification of the time ordering between observations and secrets.

[2] See also [8, App. E] for a discussion on combining independent secret sources for more holistic multi-policy security guarantees.

▶ **Theorem 6.** (Transport Theorem [8]) If $\mathcal{A} \models \mathcal{F}$ and $\mathcal{F}' \leq \mathcal{F}$, then $\mathcal{A} \models \mathcal{F}'$.

In conclusion, one can use the System Compositionality Theorem to obtain for the composite system $\mathcal{A}_1 \times^{\mathsf{match}} \mathcal{A}_2$ a flow policy $\mathcal{F} = \mathcal{F}_1 \times^{(\mathsf{match},\mathsf{matchO},\mathsf{matchS})} \mathcal{F}_2$ with a strong bound, and the Transport Theorem to produce from this a perhaps weaker but more natural flow policy $\mathcal{F}'$ (for the same system $\mathcal{A}_1 \times^{\mathsf{match}} \mathcal{A}_2$). [8, App.A] gives more intuition on using the two theorems in tandem.

2.7.4 The *n*-ary case

The System Compositionality Theorem generalizes quite smoothly from the binary to the n-ary case. Let $(\mathcal{A}_k = (\mathsf{State}_k, \mathsf{Act}_k, \mathsf{Out}_k, \mathsf{istate}_k, \mathsf{Trans}_k))_{k\in\{1,\ldots,n\}}$ be a family of n systems. We fix, for each k, k' with $k \neq k'$, a matching predicate $\mathsf{match}_{k,k'} : \mathsf{Trans}_k \times \mathsf{Trans}_{k'} \to \mathsf{Bool}$. We write match for the family $(\mathsf{match}_{k,k'})_{k,k'}$ and $\mathsf{isCom}_{k,k'} : \mathsf{Act}_k \to \mathsf{Bool}$ for the corresponding notion of communication action (belonging to $\mathcal{A}_k$ and pertaining to communication with $\mathcal{A}_{k'}$). We will make the sanity assumption that a system cannot use the same action to communicate with different systems.

Pairwise-Dedicated Communication: If $k' \neq k''$, then for all k the predicates $\mathsf{isCom}_{k,k'}$ and $\mathsf{isCom}_{k,k''}$ are disjoint, in that there exists no $a \in \mathsf{Act}_k$ such that $\mathsf{isCom}_{k,k'}\ a$ and $\mathsf{isCom}_{k,k''}\ a$. The match-communicating product of the family of systems $(\mathcal{A}_k)_{k\in\{1,\ldots,n\}}$, written $\prod^{\mathsf{match}}_{k\in\{1,\ldots,n\}} \mathcal{A}_k$, generalizes of the binary case. Namely, it is the following system $(\mathsf{State}, \mathsf{Act}, \mathsf{Out}, \mathsf{istate}, \mathsf{Trans})$:

- $\mathsf{State} = \prod_{k\in\{1,\ldots,n\}} \mathsf{State}_k$; so the states are families $(\sigma_k)_{k\in\{1,\ldots,n\}}$, or $(\sigma_k)_k$ for short;
- $\mathsf{Act} = \sum_{k\in\{1,\ldots,n\}} \mathsf{Act}_k + \sum_{k,k'\in\{1,\ldots,n\},k\neq k'} \mathsf{Act}_k \times \mathsf{Act}_{k'}$; we write (i, a_i) for elements of the i'th summand on the left (separate actions by component $\mathcal{A}_i$), and $((i, a_i), (j, a_j))$ for elements of the (i, j)'th summand on the right (joint communicating actions by components $\mathcal{A}_i$ and $\mathcal{A}_j$);
- $\mathsf{Out} = \sum_{k\in\{1,\ldots,n\}} \mathsf{Out}_k + \sum_{k,k'\in\{1,\ldots,n\},k\neq k'} \mathsf{Out}_k \times \mathsf{Out}_{k'}$ (similarly to Act);
- $\mathsf{istate} = (\mathsf{istate}_k)_{k\in\{1,\ldots,n\}}$;
- Trans contains two kinds of transitions:
  - for $i \in \{1, \ldots, n\}$, separate $\mathcal{A}_i$-transitions $((\sigma_k)_k, (i, a_i), (i, ou_i), (\sigma_k)_k[i := \sigma'_i])$, where $(\sigma_i, a_i, ou_i, \sigma'_i) \in \mathsf{Trans}_i$ and $\neg\ \mathsf{isCom}_i\ a_i$;
  - for $i, j \in \{1, \ldots, n\}$ such that $i \neq j$, communication transitions (between $\mathcal{A}_i$ and $\mathcal{A}_j$) $((\sigma_k)_k, ((i, a_i), (j, a_j)), ((i, ou_i), (j, ou_j)), (\sigma_k)_k[i := \sigma'_i, j := \sigma'_j])$, where $(\sigma_i, a_i, ou_i, \sigma'_i) \in \mathsf{Trans}_i$, $(\sigma_j, a_j, ou_j, \sigma'_j) \in \mathsf{Trans}_j$ and $\mathsf{match}_{i,j}(\sigma_i, a_i, ou_i, \sigma'_i)(\sigma_j, a_j, ou_j, \sigma'_j)$.

Above, we wrote $(\sigma_k)_k[i := \sigma'_i]$ for the family of states that is the same as $(\sigma_k)_k$, except for the index i where it is updated from σ_i to σ'_i; and similarly for $(\sigma_k)_k[i := \sigma'_i, j := \sigma'_j]$.

Given the flow policies $\mathcal{F}_k$ for the component systems $\mathcal{A}_k$ and the families of matching predicates for transitions, $\mathsf{match} = (\mathsf{match}_{k,k'})_{k,k'}$, observations, $\mathsf{matchO} = (\mathsf{matchO}_{k,k'})_{k,k'}$, and secrets, $\mathsf{matchS} = (\mathsf{matchS}_{k,k'})_{k,k'}$, the product flow policy $\prod^{(\mathsf{match},\mathsf{matchO},\mathsf{matchS})}_{k\in\{1,\ldots,n\}} \mathcal{F}_k$ is defined as a straightforward generalization of the binary case. For example, its observation domain is $\sum_{k\in\{1,\ldots,n\}} \mathsf{Obs}_k + \sum_{k,k'\in\{1,\ldots,n\},k\neq k'} \mathsf{Obs}_k \times \mathsf{Obs}_{k'}$, so that it contains either separate observations (k, o_k) or joint observations $((k, o_k), (k', o_{k'}))$. Its trigger T is defined on separate i-transitions to be the trigger of the i component, and on (i, j)-communication transitions to be the disjunction of the triggers of the i and j component. And its bound $\mathsf{B}\ sl\ sl'$ is defined from the component bounds: For all $(sl_k)_k, (sl'_k)_k \in \prod_{k\in\{1,\ldots,n\}} \mathsf{List}(\mathsf{Sec}_k)$, if $sl \in \bigtimes^{\mathsf{matchS}}(sl_k)_k$ and $sl' \in \bigtimes^{\mathsf{matchS}}(sl'_k)_k$, then, for all k, $\mathsf{B}_k\ sl_k\ sl'_k$ holds – where $\bigtimes^{\mathsf{matchS}}$ is the n-ary matchS-shuffle product operator, which applied to a family of lists of secrets $(sl_k)_k$ gives all possible interleavings of these lists achieved by separate individual steps and communication steps.

Now we can formulate an n-ary generalization of the System Compositionality Theorem. Most of its assumptions will be those of the binary version, applied to all pairs of components (k, k') for $k, k' \in \{1, \ldots, n\}$ and $k \neq k'$. The only exception is Secret Polarization, which must be strengthened. It is not sufficient to have a single secret issuer for every pair (k, k'), but we need a unique secret issuer for the entire system of n components.

Unique Secret Polarization: There exists $i \in \{1, \ldots, n\}$ such that for all $k \in \{1, \ldots, n\}$ with $k \neq i$ and for all $t \in \mathsf{Trans}_k$, $\mathsf{isSec}_k\ t$ implies $\mathsf{isCom}_{k,i}\ (\mathsf{actOf}_k\ t)$.

▶ **Theorem 7.** (System Compositionality Theorem, n-ary case [8]) Assume the following:

- For all $k, k' \in \{1, \ldots, n\}$ such that $k \neq k'$, the flow policies $\mathcal{F}_k$ and $\mathcal{F}_{k'}$ and their communication infrastructure $(\mathsf{match}_{k,k'}, \mathsf{matchO}_{k,k'}, \mathsf{matchS}_{k,k'})$ satisfy the properties of Pairwise-Dedicated, Compatible, Strong and Observable Communication.
- The families $(\mathcal{F}_k)_{k \in \{1,\ldots,n\}}$ and $(\mathsf{match}_{k,k'}, \mathsf{matchO}_{k,k'}, \mathsf{matchS}_{k,k'})_{k,k' \in \{1,\ldots,n\}, k \neq k'}$ (as a whole) satisfy Unique Secret Polarization.
- $\mathcal{A}_k \models \mathcal{F}_k$ for all $k \in \{1, \ldots, n\}$.

Then $\prod^{\mathsf{match}}_{k \in \{1,\ldots,n\}} \mathcal{A}_k \models \prod^{(\mathsf{match},\mathsf{matchO},\mathsf{matchS})}_{k \in \{1,\ldots,n\}} \mathcal{F}_k$.

In conclusion, the generalization of the System Compositionality Theorem to the n-ary case proceeds almost pairwise, but with an additional sanity assumption (Pairwise-Dedicated Communication) and a strengthened assumption (Unique Secret Polarization).

3 Verified Systems

We have formalized in Isabelle/HOL the BD security framework (consisting of Section 2's concepts and theorems) [10, 35]. Recall that the framework operates on nondeterministic I/O automata. We have instantiated it to particular (deterministic) automata representing the functional kernels of some web-based systems. Fig. 5 shows the high-level architecture of these systems, which follows a paradigm of security by design:

- The kernel is formalized and verified in Isabelle.
- The formalization is automatically translated into a functional programming language – which in all our case studies was Scala, one of the target languages of Isabelle's code generator [20, 21].
- The translated program is wrapped in a user-friendly web application.

3.1 CoCon

CoCon [23, 36, 37] is an EasyChair-like conference management system, which was deployed to two international conferences: TABLEAUX 2015 and ITP 2016 [37, §5]. The web application

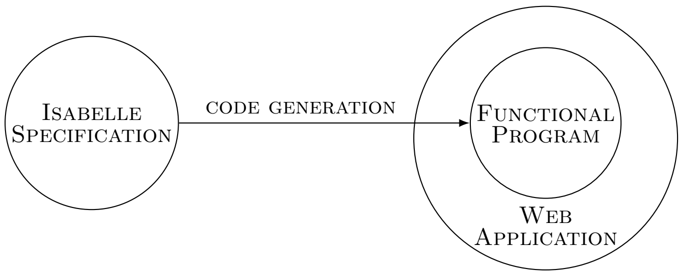


Figure 5 High-level architecture of the verified systems.

Table 1 Confidentiality properties for CoCon. The observations are made by a group of users G.

Secrets	Declassification Trigger	Declassification Bound
Paper	Some user in G is one of the paper's authors	Last uploaded version
	Some user in G is one of the paper's authors or a PC member$^{\mathsf{B}}$	Absence of any upload
Review	Some user in G is the review's author	Last edited version before Discussion and all the later versions
	Some user in G is the review's author or a non-conflicted PC member$^{\mathsf{D}}$	Last edited version before Notification
	Some user in G is the review's author or a non-conflicted PC member$^{\mathsf{D}}$ or the reviewed paper's author$^{\mathsf{N}}$	Absence of any edit
Discussion	Some user in G is a non-conflicted PC member	Absence of any edit
Decision	Some user in G is a non-conflicted PC member	Last edited version
	Some user in G is a non-conflicted PC member or a PC member$^{\mathsf{N}}$ or the decided paper's author$^{\mathsf{N}}$	Absence of any edit
Reviewer assignment	Some user in G is a non-conflicted PC member$^{\mathsf{R}}$	Reviewers being non-conflicted PC members, and number of reviewers
	Some user in G is a non-conflicted PC member$^{\mathsf{R}}$ or one of the reviewed paper's authors$^{\mathsf{N}}$	Reviewers being non-conflicted PC members

Phase Stamps: B = Bidding, D = Discussion, N = Notification, R = Review

layer of Fig. 5 was realized as a thin REST API implemented in Scalatra [45] wrapped around the verified kernel together with a stateless GUI written in AngularJS [2] that communicates with the API.

CoCon was our first case study, which motivated the initial design and formalization of the BD security framework. Our goal to express, let alone verify, fine-grained policies concerning the flow of information in CoCon between users and documents, could not be supported by the existing concepts in the literature. (See [23, §4.1] for a discussion.) Examples of properties we wanted to express are:

(1) A group of users learn nothing about a paper *beyond* the last uploaded version *unless* one of them becomes an author of that paper.

(2) A group of users learn nothing about a paper *beyond* the absence of any upload *unless* one of them becomes an author of that paper or a PC member at the paper's conference.

(3) A group of users learn nothing about the content of a review *beyond* the last edited version before Discussion phase and the later versions *unless* one of them is that review's author.

The BD security trigger and bound were born out of the need to formally capture the "unless" and "beyond" components of such properties. Tab. 1 summarizes informally the CoCon properties we have expressed in our framework as flow policies. The observation

Table 2 Confidentiality properties for the original CoSMed. The observations are made by a group of users G. The trigger is vacuously false.

Secrets	**Declassification Bound**
Content of a given post	Updates performed while or last before one of the following holds: – Some user in G is the admin, is the post owner or is friends with its owner – The post is marked as public
Friendship status between two given users U and V	Status changes performed while or last before the following holds: – Some user in G is the admin or is friends with U or V
Friendship requests between two given users U and V	Existence of accepted requests while or last before the following holds: – Some user in G is the admin or is friends with U or V

Table 3 Confidentiality properties for CoSMeDis, lifted from CoSMed. The observations are made by n groups of users – one group G_i at each node i. The declassification trigger is vacuously false.

Secrets	**Declassification Bound**
Content of a given post at node i	Updates performed while or last before one of the following holds: – Some user in G_i is the node's admin, is the post owner or is friends with its owner – The post is marked as public – Some user in G_j for $j \neq i$ is the admin at node j or is remote friends with the post's owner
Friendship status between two given users U and V at node i	Status changes performed while or last before the following holds: – Some user at node i is the node's admin or is friends with U or V
Friendship requests between two given users U and V at node i	Existence of accepted requests while or last before the following holds: – Some user at node i is the node's admin or is friends with U or V

infrastructure is always the same, given by the actions and outputs of a fixed group G of users. The secrecy infrastructures are given by the various documents managed by the system (paper content, review, discussion, decision) but also, in the table's last two rows, by information about the reviewers assigned to a paper. These properties should be read as follows: A group of users learns nothing about the given secret (more precisely, about all the uploads or edits performed on a document in the indicated "secret" category) beyond the indicated bound, unless the indicated trigger becomes true. For example, the above properties (1)–(3) are the first three shown in the table, with slightly stronger triggers factoring in the conference phase as well, which we indicate succinctly via "phase stamps" – e.g., the presence of the phase stamp "D" indicates the requirement that the conference must have moved into the Discussion phase. For each type of secret, we have a range of increasingly restrictive bounds matched by increasingly weaker triggers – indeed, the more we tighten the bound (meaning we allow less information to flow), the weaker the trigger becomes (since there are more events that could break the bound). This bound–trigger dynamics exhaustively characterizes the possible flows in the system.

The notion of BD unwinding was developed and refined during the verification of CoCon's policies. The opportunity to take proof shortcuts (via the exit predicate) was discovered during practical "proof hacking" sessions, and led to major simplifications in the development. The different unwinding components in the Sequential Multiplex Unwinding Theorem were naturally mapped to the different phases of a conference's workflow.

3.2 CoSMed

CoSMed [9, 11] is a simple Facebook-style social media platform, where users can register, create posts and establish friendship relationships. It was implemented following the same high-level architecture as CoCon. But unlike CoCon, CoSMed is only a research prototype, not intended for practical use.

CoSMed's confidentiality properties raised new challenges and inspired a more expressive way of modeling flows. In the style of CoCon, we could have specified and proved properties such as:

A group of users learn nothing about a post *unless* one of them is the admin, or is the post's owner, or becomes friends with the owner, or the post gets marked as public.

Remember that the trigger introduced via "unless" expresses a condition in whose presence the property stops guaranteeing anything – in other words, a trigger opens an access window indefinitely. While true, such a property is not strong enough to be useful for CoSMed, where both friendship and public visibility can be freely switched on and off by the owner at any time (e.g., by "unfriending" a user, and later "friending" them again). Instead, we wanted to prove more dynamic flow policies, reflecting any number of successive openings and closings of the access windows during system execution.

Tab. 2 summarizes informally the BD security properties that we ended up proving for CoSMed. The observation infrastructure is again given by a group G of users, and the secrecy infrastructure refers to either the content of a given post, or to information on the friendship status between two users or on the issued friendship requests. For example, the property on the first row is the dynamic-flow refinement of the coarser property discussed above:

A group of users learn nothing about a post beyond the updates performed while (or last before) one of them is the admin, or is the post's owner, or becomes friends with the owner, or the post is marked as public.

Thus, the "beyond–unless" bound-trigger combination we had employed for CoCon gave way to a "beyond–while" scheme for CoSMed, where "while" refers to the allowed access windows. To achieve this formally, we made the triggers vacuously false (i.e., deactivated them completely) and incorporated the opening and closing of access windows in inductively defined bounds. [9] discusses in detail this paradigm shift, which however did not require adjustments to the framework itself.

3.3 CoSMeDis

CoSMeDis [8, 12] is a multi-node distributed extension of CoSMed that follows a Diaspora-style scheme [1]: Different nodes can be deployed independently at different internet locations. The admins of any two nodes can initiate a protocol to connect these nodes, after which the users of one node can establish friendship relationships and share data with users of the other. Thus, a node of CoSMeDis consists of CoSMed plus actions for connecting nodes and cross-node post sharing and friending.

Our goal was to extend the confidentiality properties we had verified for CoSMed first to one CoSMeDis node, then to the multi-node CoSMeDis network. [8] describes in great detail this verification extension effort, which led to the discovery of the System Compositionality Theorems. The outcome was the properties shown in Tab. 3, which are natural multi-node generalizations of CoSMed's properties (from Tab. 2). They were obtained by applying the n-ary System Compositionality Theorem, then the Transport Theorem to switch to more readable secrets and bounds.

4 Related Work

We only discuss briefly the most related work, focusing on the general framework rather than the verification case studies. For more comprehensive literature comparisons (which also cover verification), we refer to our earlier papers [8, 9, 37].

Since we aimed for high expressiveness and precision, we defined BD security by quantifying over execution traces of general systems. This "heavy duty" approach, sometimes called *system-based security* [26], can be contrasted with *language-based security* [42], concerned with coarser-grained but tractable notions that can be automatically analyzed on programming language syntax.

BD security provides an expressive realization of Sabelfeld and Sands's dimensions of declassification [44] in a system-based setting. It descends from the epistemic logic [40] inspired tradition of modeling information-flow security, pioneered by Sutherland with Nondeducibility [46] and continued with Halpern and O'Neill's Secrecy Maintenance [22] and with Askarov et al.'s Gradual Release [3–6], the latter developed in a language-based setting. Our BD unwinding is a non-trivial generalization of unwinding proof methods going back to Goguen and Meseguer [19] and Rushby [41], which have been extensively studied as part of Mantel's MAKS framework [24, 26]. Unlike these predecessors which use safety-like unwinding conditions, BD unwinding combines safety with liveness: In the BD unwinding game, the "defender", who builds the alternative trace tr_2, must

- not only be able to *always* stay in the game – a safety-like property,
- but also be able to *eventually* produce the alternative secrets sl_2 (provided the "attacker", who controls the original trace tr_1, has produced all the original secrets sl_1) – a liveness-like property.

Because of the restrictive way of handling the liveness part of the aforementioned game, BD unwinding is not a complete proof method, in that it cannot prove every instance of BD security. We leave a complete extension of BD unwinding as future work.

Our system compositionality result joins a body of technically delicate work in system-based security, where the difficult terrain was recognized early on [27]. Several frameworks have been developed in various settings, e.g., event systems [25], reactive systems [39] and process calculi [13, 17]. Some of these focus on formulating very restricted classes of security properties that are always guaranteed to be preserved under a given notion of composition, such as McCullough's Restrictiveness [28]. Others, such as Mantel's MAKS framework [24, 25], formulate side conditions on the components' security properties that guarantee compositionality. Our result is in the latter category, and refers to a significantly more expressive notion of information-flow security than its predecessors (which is not to say that our result subsumes these previous results).

Temporal logics designed for information-flow security, such as SecLTL [15] and HyperCTL* [14, 16, 38], can express similar-looking properties to the instances of BD security we verified for CoCon – though semantically they differ by interpreting trace quantification synchronously.

References

1 The Diaspora project. `https://diasporafoundation.org/`, 2016.

2 The AngularJS Web Framework, 2021. URL: `https://angularjs.org/`.

3 Aslan Askarov and Stephen Chong. Learning is change in knowledge: Knowledge-based security for dynamic policies. In *CSF*, pages 308–322, 2012.

4 Aslan Askarov and Andrew C. Myers. Attacker control and impact for confidentiality and integrity. *Logical Methods in Computer Science*, 7(3), 2011.

5 Aslan Askarov and Andrei Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *IEEE Symposium on Security and Privacy*, pages 207–221, 2007.

6 Aslan Askarov and Andrei Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *CSF*, pages 43–59, 2009.

7 Thomas Bauereiss, Armando Pesenti Gritti, Andrei Popescu, and Franco Raimondi. CoSMed: A Confidentiality-Verified Social Media Platform. In *ITP*, 2016.

8 Thomas Bauereiss, Armando Pesenti Gritti, Andrei Popescu, and Franco Raimondi. CoSMeDis: A distributed social media platform with formally verified confidentiality guarantees. In *IEEE Symposium on Security and Privacy*, pages 729–748, 2017.

9 Thomas Bauereiss, Armando Pesenti Gritti, Andrei Popescu, and Franco Raimondi. CoSMed: A Confidentiality-Verified Social Media Platform. *J. Autom. Reasoning*, 61(1-4):113–139, 2018.

10 Thomas Bauereiss and Andrei Popescu. Compositional BD Security. *Archive of Formal Proofs*, 2021. URL: `https://www.isa-afp.org/entries/Compositional_BD_Security.html`.

11 Thomas Bauereiss and Andrei Popescu. CoSMed: A confidentiality-verified social media platform. *Archive of Formal Proofs*, 2021. URL: `https://www.isa-afp.org/entries/CoSMed.html`.

12 Thomas Bauereiss and Andrei Popescu. CoSMeDis: A confidentiality-verified distributed social media platform. *Archive of Formal Proofs*, 2021. URL: `https://www.isa-afp.org/entries/CoSMeDis.html`.

13 Annalisa Bossi, Damiano Macedonio, Carla Piazza, and Sabina Rossi. Information flow in secure contexts. *Journal of Computer Security*, 13(3):391–422, 2005. URL: `http://content.iospress.com/articles/journal-of-computer-security/jcs235`.

14 Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In *POST*, pages 265–284, 2014.

15 Rayna Dimitrova, Bernd Finkbeiner, Máté Kovács, Markus N. Rabe, and Helmut Seidl. Model checking information flow in reactive systems. In *VMCAI*, pages 169–185, 2012.

16 Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for model checking HyperLTL and HyperCTL ˆ*. In *International Conference on Computer Aided Verification*, pages 30–48. Springer, 2015.

17 Riccardo Focardi and Roberto Gorrieri. Classification of security properties (Part I: Information flow). In *FOSAD*, pages 331–396, 2000.

18 Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

19 Joseph A. Goguen and José Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pages 75–87, 1984.

20 Florian Haftmann. *Code Generation from Specifications in Higher-Order Logic*. Ph.D. thesis, Technische Universität München, 2009.

21 Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In *FLOPS 2010*, pages 103–117, 2010.

22 Joseph Y. Halpern and Kevin R. O'Neill. Secrecy in multiagent systems. *ACM Trans. Inf. Syst. Secur.*, 12(1), 2008.

23 Sudeep Kanav, Peter Lammich, and Andrei Popescu. A conference management system with verified document confidentiality. In *CAV*, pages 167–183, 2014.

24 Heiko Mantel. Possibilistic definitions of security - an assembly kit. In *CSFW*, pages 185–199, 2000.

25 Heiko Mantel. On the composition of secure systems. In *IEEE Symposium on Security and Privacy*, pages 88–101, 2002.

26 Heiko Mantel. *A Uniform Framework for the Formal Specification and Verification of Information Flow Security*. PhD thesis, University of Saarbrücken, 2003.

27 Daryl McCullough. Specifications for multi-level security and a hook-up property. In *IEEE Symposium on Security and Privacy*, 1987.

28 Daryl McCullough. A hookup theorem for multilevel security. *IEEE Trans. Software Eng.*, 16(6):563–568, 1990.

29 John McLean. Security models. In *Encyclopedia of Software Engineering*, 1994.

30 Toby C. Murray, Andrei Sabelfeld, and Lujo Bauer. Special issue on verified information flow security. *Journal of Computer Security*, 25(4-5):319–321, 2017.

31 Tobias Nipkow and Gerwin Klein. *Concrete Semantics: With Isabelle/HOL*. Springer, 2014.

32 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.

33 Andrei Popescu, Johannes Hölzl, and Tobias Nipkow. Proving concurrent noninterference. In *CPP*, pages 109–125, 2012.

34 Andrei Popescu, Johannes Hölzl, and Tobias Nipkow. Formalizing probabilistic noninterference. In *CPP*, pages 259–275. Springer, 2013.

35 Andrei Popescu and Peter Lammich. Bounded-deducibility security. *Archive of Formal Proofs*, 2014. URL: `https://www.isa-afp.org/entries/Bounded_Deducibility_Security.html`.

36 Andrei Popescu and Peter Lammich. CoCon: A confidentiality-verified conference management system. *Archive of Formal Proofs*, 2021. URL: `https://www.isa-afp.org/entries/CoCon.html`.

37 Andrei Popescu, Peter Lammich, and Ping Hou. CoCon: A conference management system with formally verified document confidentiality. *J. Autom. Reason.*, 65(2):321–356, 2021.

38 Markus N. Rabe, Peter Lammich, and Andrei Popescu. A shallow embedding of HyperCTL. *Archive of Formal Proofs*, 2014, 2014.

39 Willard Rafnsson and Andrei Sabelfeld. Compositional information-flow security for interactive systems. In *CSF*, pages 277–292, 2014.

40 Yoram Moses Ronald Fagin, Joseph Y. Halpern and Moshe Vardi. *Reasoning about knowledge*. MIT Press, 2003.

41 John Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, Computer Science Laboratory SRI International, December 1992. URL: `http://www.csl.sri.com/papers/csl-92-2/`.

42 Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

43 Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *CSFW*, pages 200–214, 2000.

44 Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

45 The Scalatra Web Framework, 2021. URL: `http://scalatra.org/`.

46 D. Sutherland. A model of information. In *9th National Security Conf.*, pages 175–183, 1986.

47 Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.

A Graphical User Interface Framework for Formal Verification

Edward W. Ayers
DPMMS, University of Cambridge, UK

Mateja Jamnik
Department of Computer Science and Technology, University of Cambridge, UK

W. T. Gowers
DPMMS, University of Cambridge, UK

Abstract

We present the "ProofWidgets" framework for implementing general user interfaces (UIs) within an interactive theorem prover. The framework uses web technology and functional reactive programming, as well as metaprogramming features of advanced interactive theorem proving (ITP) systems to allow users to create arbitrary interactive UIs for representing the goal state. Users of the framework can create GUIs declaratively within the ITP's metaprogramming language, without having to develop in multiple languages and without coordinated changes across multiple projects, which improves development time for new designs of UI. The ProofWidgets framework also allows UIs to make use of the full context of the theorem prover and the specialised libraries that ITPs offer, such as methods for dealing with expressions and tactics. The framework includes an extensible structured pretty-printing engine that enables advanced interaction with expressions such as interactive term rewriting. We exemplify the framework with an implementation for the leanprover-community fork of Lean 3. The framework is already in use by hundreds of contributors to the Lean mathematical library.

2012 ACM Subject Classification Software and its engineering → Software usability

Keywords and phrases User Interfaces, ITP

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.4

Supplementary Material The supplementary material presented with this paper is incorporated into the leanprover-community GitHub repositories.
Software (Server): `https://github.com/leanprover-community/lean/tree/master/library/init/meta/widget`; archived at `swh:1:dir:65d6fe171a7697793be204922aba83f2a94f5d20`
Software (Client): `https://github.com/leanprover/vscode-lean/blob/master/infoview/widget.tsx`; archived at `swh:1:cnt:e713dbc30927867464effc1e51fd1230cd961cbd`

Funding *Edward W. Ayers*: EPSRC 1804138.

Acknowledgements Enormous thanks to Gabriel Ebner and Brian Gen-ge Chen for reviewing the PRs and polishing the implementation. Also thanks for using the framework to create their own widgets, contributing and suggesting improvements: Daniel Fabian; Robert Y. Lewis; Markus Himmel; Minchao Wu; Kendall Frey; Patrick Massot.

1 Introduction

Modern ITP systems such as Isabelle, Coq and Lean use advanced language servers and protocols to interface with text editors to produce a feature-rich proving experience. These systems have helpful features such as syntax highlighting and code completion suggestions as would be found in normal programming language tooling. They additionally include prover-specific features such as displaying the goal state and providing interactive suggestions of tactics to apply in proof construction. ITP offers some additional GUI[1] challenges above what one might find in developing an editor extension for a standard programming language, because the process of proving is inherently interactive: the user is constantly observing the goal state of the prover and using this information to inform their next command in the construction of a proof.

During research into new ways of interacting within Lean 3 theorem prover, we became frustrated with the development workflow for prototyping GUIs for ITP. Each time the interface design changes, one needs to coordinate changes across three different codebases; the Lean core, the VSCode[2] editor extension and our project directory. It became clear that any approach to creating GUIs in which the editor code needs to be aware of the datatypes used within the ITP metalogic is doomed to require many coordinated changes across multiple codebases. This inspired our alternative approach; we designed a full-fledged GUI framework in the metalogic of the ITP itself. This approach has the advantage of tightening the development loop and has more general use outside of our particular project, and across different ITP systems in general.

In this paper we present the ProofWidgets framework, which enables implementation of UIs with a wide variety of features, for example:

- Interactive term inspection: the ability to inspect the tree structure of expressions by hovering the mouse over different parts of a pretty printed expression.
- Discoverable tactics: interactive suggestions of available tactics for a given goal.
- Discoverable term rewriting: the ability to inspect the equational rewrites at a particular position in the expression tree.
- Custom visualisations of structures like matrices, plots and graphs as well as rendering LaTeX mathematical typesetting.
- The ability to implement language features such as go-to-definition for pretty-printed expressions (as opposed to just text that appears in the editor).

Some of these features have been implemented to an extent within other provers (see Section 5). However, ProofWidgets provides a unified, underlying framework for implementing user interfaces in general, which can be used to implement these features in a customisable, extensible and portable way.

The contributions presented in this paper are:

- A new and general framework for creating portable, web-based, graphical UIs within a theorem prover.
- A functional API for creating widgets within the meta-programming framework of a theorem prover.
- An implementation of this framework for the Lean theorem prover.
- A new representation of structured expressions for use with widgets.
- A description and implementation of a goal-state widget used to interactively show and explore goal states within the Lean theorem prover.

This paper is structured as follows. In Section 2 we provide an overview of some background topics to contextualise the work. Section 3 details the design goals and specification of the ProofWidgets framework. Section 4 presents the ProofWidgets implementation for Lean 3. Section 5 discusses related approaches. Section 6 is the conclusion and contains some potential ideas for future work and extensions to the framework.

[1] Graphical User Interface
[2] Visual Studio Code `code.visualstudio.com`

2 Background

2.1 Web-apps

Web-apps are ubiquitous in modern software. By a web-app, we mean any software that uses modern browser technology to implement a graphical application. Web-apps are attractive targets for development because they are platform independent and can be delivered from a server on the internet using a browser or be packaged as an entirely local app using a packaging framework such as electron. Many modern desktop and mobile applications such as VSCode are thinly veiled browser windows.

To summarise the anatomy of a web-app: the structure of a web-page is dictated by a tree structure called the Document Object Model (DOM). The DOM is an abstract representation of the tree structure of an XML or HTML document with support for event handling as might occur as a result of user interaction. The word *fragment* is used to denote a valid subtree structure that is not the entire document. So for example, an "HTML fragment" is used to denote a snippet of HTML that could be embedded within an HTML document. With the help of a CSS style sheet, the web browser paints this DOM to the screen in a way that can be viewed and interacted with by a user. Through the use of JavaScript and event handlers, a webpage may manipulate its own DOM in response to events to produce interactive web-applications. Modern browsers support W3C standards for many advanced features: video playback, support for touch and ink input methods, drag and drop, animation, 3D rendering and many more. HTML also has a set of widely supported accessibility features called ARIA which can be used to ensure that apps are accessible to all. The power of web-apps to create portable, fully interactive user interfaces has clear applications for ITP and indeed many have already been created (see Section 5 for a review).

2.2 Code editors and client-server protocols

Some modern code editors such as Atom and VSCode are built using web technology. In order to support tooling features such as go-to-definition and hover information, these editors act as the client in a client/server relationship with an independently running *language server*. As the user modifies the code in the client editor, the client communicates with the server: notifying it when the document changes and sending requests for specific information based on the user's interactions. As noted in the introduction, in ITP this communication is more elaborate than in a normal programming language.

The most important thing to note here is that changing the communication protocol between the client and the server is generally hard, because the developer has to update the protocol in both the server and the client. There may even be multiple clients. This makes it difficult to quickly iterate on new UI designs. A way of solving this protocol problem is to offer a much tighter integration by combining the codebases for the editor and the ITP. This is the approach taken by Isabelle/PIDE/jEdit [21] and has its own trade-offs as discussed further in Section 5.

2.3 Functional GUI frameworks

Most meta-level programming languages for ITPs are functional programming languages.[3] However GUIs are inherently mutable objects that need to react to user interaction. Reactive programming [1] enables the control of the inherently mutating GUI within a pure functional

[3] ML and Scala for Isabelle, OCaml for Coq, Lean for Lean.

programming interface.[4] The ideas of reactive programming have achieved a wide level of adoption in web-app development thanks to the React JavaScript library and the Elm programming language [6].

The programming model used by these reactive frameworks is to model a user interface as a pure *view* function from a data source (e.g., a shopping list) to a DOM tree and an *update* function for converting user input events to a new version of the data (e.g., adding an item to the shopping list). Once the update function is applied and the data has been updated, the system reevaluates the view function on the new data and mutates the DOM to update. Although this may sound inefficient - recomputing the entire tree each time - there is an optimisation available: if the view function contains nested view functions, one can memoise these functions and avoid updating the parts of the DOM that have not changed.

A performance bottleneck in web-apps is the layout and painting stages of the browser rendering pipeline; the process by which the abstract DOM is converted to pixels on a screen.[5] One way to reduce the processing time spent on layout and painting is to minimise the number of changes made to the DOM. Both Elm and React achieve this through use of a "Virtual DOM" (VDOM). This is where a shadow tree isomorphic to the DOM is kept. When the data updates, the view function creates a new VDOM tree. This tree is then *diffed* with the previous VDOM tree to produce a minimal set of changes to the real DOM. In React, this diffing algorithm is called *reconciliation*.[6]

2.4 Lean

Lean [8] is an interactive theorem prover whose underlying logic is a dependent type theory called the calculus of inductive constructions. Lean verifies the correctness of proofs using its kernel, which type-checks proof terms (terms whose type is a proposition). Most users of Lean prove theorems using its tactic language. This language amounts to a sequence of invocations of the `tactic` monad, which enables one to write proof scripts in a similar style to that popularised by the LCF provers [12]. Most notably for our purposes, between each tactic invocation, Lean stores the *goal state* at that point, which amounts to a list of contexts and types that need to be inhabited to complete the proof. This goal state is pretty printed and sent for viewing in the client editor as plaintext, with some additional formatting (i.e., syntax highlighting) applied in the client. Lean also allows users to write custom tactic languages.

3 Framework architecture

The ProofWidgets framework has the following design goals:

- Programmers write GUIs using the metaprogramming framework of the ITP.
- Programmers are given an API that can produce arbitrary DOM fragments, including inline CSS styles.
- No cross-compilation to JavaScript or WebAssembly: the GUI-generating code must run in the same environment as the tactic system. This ensures that the user interaction handlers have full access to the tactic execution context, including the full database of definitions

[4] A similar paradigm is that of *functional reactive programming* (FRP) first invented by Elliot [9]. This is distinguished from general reactive programming by the explicit modelling of time.

[5] See this chromium documentation entry for more information on critical paths in browser rendering. `https://www.chromium.org/developers/the-rendering-critical-path`

[6] `https://reactjs.org/docs/reconciliation.html`

and lemmas, as well as all of the metaprogramming library. In a cross-compilation based approach (implementation difficulty notwithstanding), the UI programmer would have to choose which parts of this context to export to the client.
- To support interactively discoverable tactics, the system needs to be able to command the client text editor to modify its sourcetext.
- The pretty printer must be extended to allow for "interactive expressions": expressions whose tree structure may be explored interactively.
- Programmers should be able to create visualisations of their data.
- It should be convenient for programmers to be able to style their GUIs in a consistent manner.
- The GUI programming model should include some way of managing local UI state, for example, whether or not a tooltip is open.
- The GUI should be presented in the same output panel that the plaintext goal state was presented in.
- The framework should be backwards compatible with the plaintext goal state system. Users should be able to opt out of the GUI if they do not like it or want to use a non web-app editor such as Emacs.

These goal specifications led us to design ProofWidgets to use a declarative VDOM-based architecture similar to that used in the Elm programming language [6] and the React JavaScript library. By using the same programming model, we can leverage the familiarity with commonly used React and Elm. In the following subsections we will detail the design of ProofWidgets, starting with the UI programming model (Section 3.1) and the client/server protocol (Section 3.2).

3.1 UI programming model

New user interfaces are created using the `Html` and `Component` types. A user may define an HTML fragment by constructing a member of the inductive datatype `Html`, which is either an element (e.g., `<div></div>`), a string or an object called a component to be discussed shortly.

These fragments can have *event handlers* attached to them. For example, a button could have an event attribute `onclick` (as used in Listing 1) which accepts a handler $h : (\texttt{Unit} \to \alpha)$ sending the unit type to a member of some type α. When this interface is rendered in the client and the button is clicked, the server is notified and causes the node to "emit" the element $h() : \alpha$. The value of $h()$ is then propagated towards the root of the `Html` tree until it reaches a component.

A component is an inductive datatype taking two type parameters: π (the props type) and α (the action type).[7] It represents a stateful object in the user interface tree where the state $s : \sigma$ can change as a result of a user interaction event. By "stateful" we mean an object which holds some mutating state for the lifetime of the user interface. Through the use of components, it is possible to describe the behaviour of this state without having to leave the immutable world of a pure functional programming language. Three functions determine the behaviour of the component:
- $\texttt{init} : \pi \to \sigma$ initialises the state.
- $\texttt{view} : \pi \to \sigma \to \texttt{Html}\ \alpha$ maps the state to a VDOM tree.

[7] This is designed to be familiar to those who use React components `https://reactjs.org/docs/components-and-props.html`.

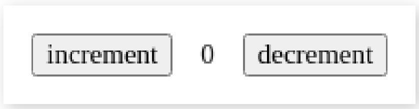


Figure 1 The output rendering of `counter` created in Listing 1.

- `update` : $\pi \to \alpha \to \sigma \to \sigma \times$ `Option` β is run when a user event is triggered in the child HTML tree returned by `view`. The emitted value $a : \alpha$ is used to produce a tuple $\sigma \times$ `Option` β consisting of a new state $s : \sigma$ and optionally, a new event $b : \beta$ to emit. If the new event is provided, it will propagate further towards the root of the VDOM tree and be handled by the next component in the sequence.

For example, a simple counter component (see Listing 1 and Figure 1) has an integer s for a state, and updating the state is done through clicking on the "increment" and "decrement" buttons which will emit 1 and -1 when clicked. The values a are used to update the state to $a + s$. Creating stateful components in this way has a variety of practical uses when building user interfaces for inspecting and manipulating the goal state. We will see in Section 4.1 that a state is used to represent which expression the user has clicked. Indeed, an entire tactic state can be stored as the state of the component. Then the update function runs various tactics to update the tactic state and output the new result.

Listing 1 Pseudocode listing showing a simple counter app showcasing statefulness. The output is shown in Figure 1.

```
counter : Component Unit Empty
counter := with_state
  ( init = (p ↦ 0)
  , view = (p ↦ i ↦
    <div>
      <button onclick={() ↦ 1}>"increment"</button>
      i
      <button onclick={() ↦ −1}>"decrement"</button>
    </div>)
  , update = (p ↦ a ↦ s ↦ (a + s, none))
  )
```

3.2 Client/server protocol

Once the programmer has built an interface using the API introduced in Section 3.1, it needs to be rendered and delivered to the browser output window. ProofWidgets extends the architecture discussed in Section 2.2 with an additional protocol for controlling the life-cycle of a user interface rendered in the client editor (Figure 2). When a sourcefile for the prover is opened (in Figure 2, `myfile.lean`), the server begins parsing, elaborating and verifying this sourcefile as usual. The server incrementally annotates the sourcetext as it is processed and these annotations are stored in memory. The annotations include tracing diagnostics messages as well as thunks[8] of the goal states at various points in a proof. When the user

[8] A thunk is a lazily evaluated expression.

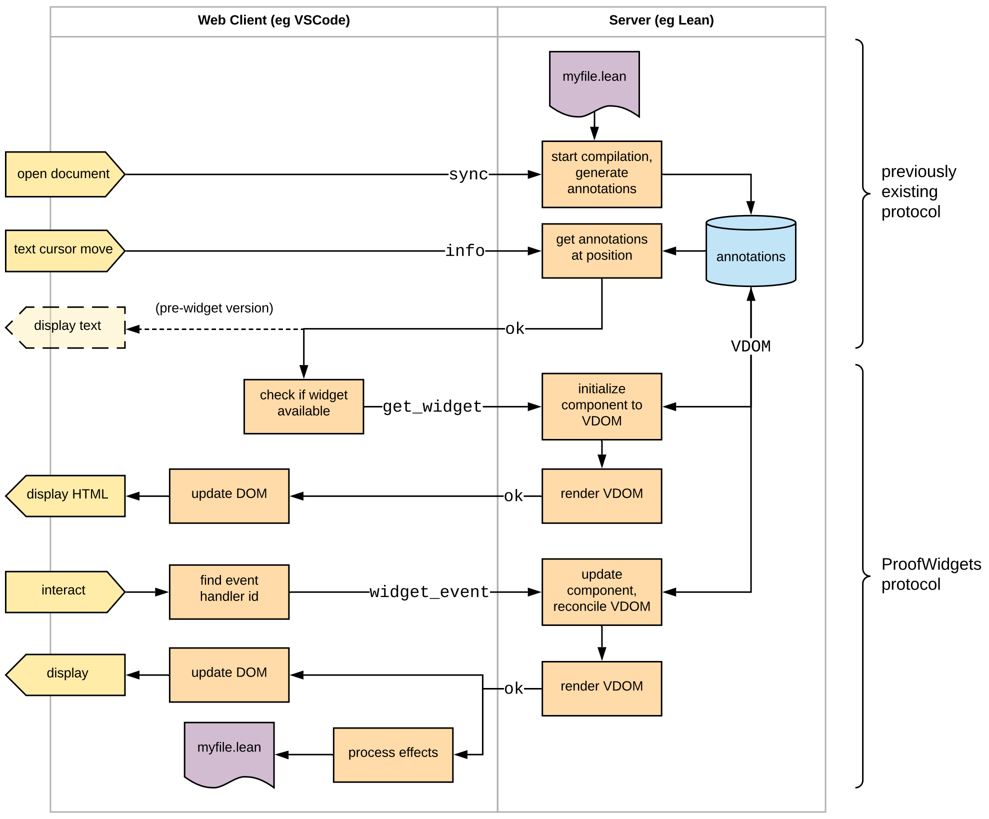


Figure 2 The architecture of the ProofWidgets client/server communication protocol. Arrows that span the dividing lines between the client and server components are API requests and responses. The contribution of this paper is present in the section marked "ProofWidgets protocol". The arrows crossing the boundary between the client and server applications are sent in the form of JSON messages. Rightward arrows are **requests** and leftward arrows are **responses**.

clicks on a particular piece of sourcecode in the editor ("text cursor move" in Figure 2), the client makes an `info` request for this position to the server, which responds with an `ok` response containing the logs at that point.

The ProofWidgets protocol extends the `info` messages to allow the prover to similarly annotate various points in the document with VDOM trees as introduced in Section 2.3. These annotating components have the type `Component TacticState Empty` where `TacticState` is the current state of the prover and `Empty` is the uninhabited type. A default component for rendering goals of proof scripts is provided, but users may override this with their own components. The VDOM trees are derived from this component, where the VDOM has the same tree structure as the `Html` datatype (i.e., a tree of elements, strings and components), but the components in the VDOM tree also contain the current state and the current child subtree of the component. This serves the purpose of storing a model of the current state of the user interface. These VDOMs can be *rendered* to HTML fragments that are sent to the client editor and presented in the editor's output window.

There are two ways to create a VDOM tree from a component: from scratch using **initialisation** or by updating an existing VDOM tree using **reconciliation**.

Initialisation is used to create a fresh VDOM tree. To initialise a component, the system first calls `init` to produce a new state s. s is fed to the `view` method to create an `Html` tree t. Any child components in t are recursively initialised.

The inputs to reconciliation are an existing VDOM tree v and a new `Html` tree t. t is created when the `view` function is called on a parent component. The goal of reconciliation is to create a new VDOM tree matching the structure of t, but with the component states from v transferred over. The tree diffing algorithm that determines whether a state should be transferred is similar to the React reconciliation algorithm[9] and so we will omit a discussion of the details here. The main point is that when a user interface changes, the states of the components are preserved to give the illusion of a mutating user interface.

For interaction, the HTML fragment returned from the server may also contain event handlers. Rather than being calls to JavaScript methods as in a normal web-app, the client editor intercepts these events and forwards them to the server using a `widget_event` request. The server then **updates** the component according to the event to produce a new `Html` tree that is reconciled with the current VDOM tree. The ProofWidgets framework then responds with the new HTML fragment derived from the new VDOM tree. In order to ensure that the correct event handler is fired, the client receives a unique identifier for each handler that is present on the VDOM and returns this identifier upon receiving a user interaction. So, in effect, the ITP server performs the role of an event handler: processing a user interaction and then updating the view rendered to the screen accordingly. In addition to updating the view, the response to a `widget_event` request may also contain **effects**. These are commands to the editor, for example revealing a certain position in the file or inserting text at the cursor position. Effects are used to implement features such as go-to definition and modifying the contents of sourcefiles in light of a suggested modification to advance the proof state. If a second user interaction event occurs while the first is being handled, the server will queue these events.

The architecture design presented above is a different approach to how existing tools handle the user interface. It offers a much smaller programming API consisting of `Component` and `Html` and a client/server protocol that supports the operation of arbitrary user interfaces controlled by the ITP server. Existing tools (Section 5) instead give fixed APIs for interaction with the ITP, or support rendering of custom HTML without or with limited interactivity.

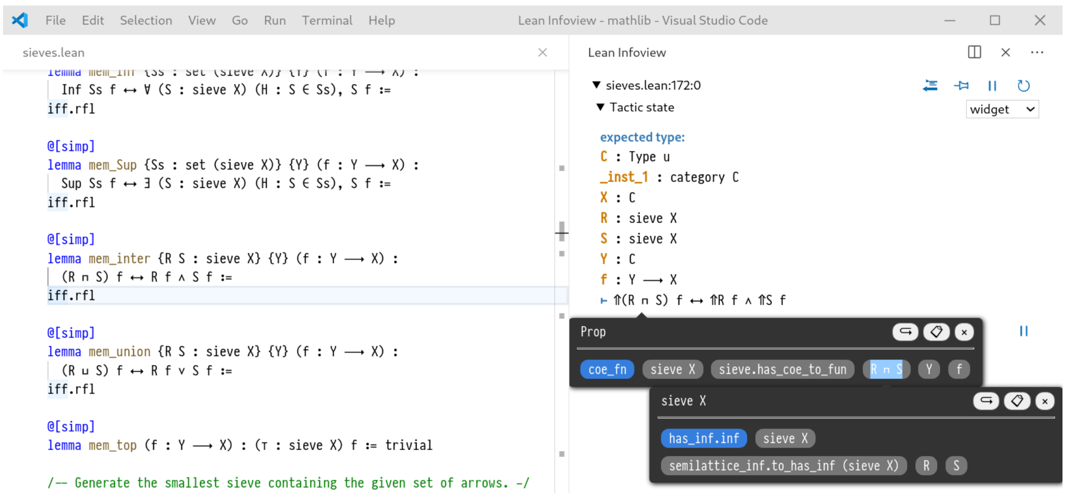
To implement ProofWidgets for an ITP system, it is necessary to implement the three subsystems that have been summarised in this section: a programming API for components; the client editor code (i.e., the VSCode extension) that receives responses from the server and inserts HTML fragments to the editors output window; and the server code to initialise, reconcile and render these components.

4 Implementation and applications

In this section, we present the Lean implementation of ProofWidgets and discuss a set of example widgets for interacting with proof objects.

The VSCode extension for Lean[10] has an output pane called the "infoview" (the right-hand pane in Figure 3) which is configured to use the ProofWidgets protocol. This infoview window runs as a sandboxed web-browser instance and uses React to manage updating the

[9] `https://reactjs.org/docs/reconciliation.html`
[10] `https://github.com/leanprover/vscode-lean`


File Edit Selection View Go Run Terminal Help
Lean Infoview - mathlib - Visual Studio Code
sieves.lean
Lean Infoview
widget

```
lemma mem_Inf {Ss : set (sieve X)} {Y} (f : Y ⟶ X) :
  Inf Ss f ↔ ∀ (S : sieve X) (H : S ∈ Ss), S f :=
iff.rfl

@[simp]
lemma mem_Sup {Ss : set (sieve X)} {Y} (f : Y ⟶ X) :
  Sup Ss f ↔ ∃ (S : sieve X) (H : S ∈ Ss), S f :=
iff.rfl

@[simp]
lemma mem_inter {R S : sieve X} {Y} (f : Y ⟶ X) :
  (R ⊓ S) f ↔ R f ∧ S f :=
iff.rfl

@[simp]
lemma mem_union {R S : sieve X} {Y} (f : Y ⟶ X) :
  (R ⊔ S) f ↔ R f ∨ S f :=
iff.rfl

@[simp]
lemma mem_top (f : Y ⟶ X) : (⊤ : sieve X) f := trivial

/-- Generate the smallest sieve containing the given set of arrows. -/
```

```
▼ sieves.lean:172:0
 ▼ Tactic state

  expected type:
  C : Type u
  _inst_1 : category C
  X : C
  R : sieve X
  S : sieve X
  Y : C
  f : Y ⟶ X
  ⊢ ⇑(R ⊓ S) f ↔ ⇑R f ∧ ⇑S f

  Prop
  coe_fn  sieve X  sieve.has_coe_to_fun  R ⊓ S  Y  f

      sieve X
      has_inf.inf  sieve X
      semilattice_inf.to_has_inf (sieve X)  R  S
```

Figure 3 Screenshot showing the interactive expression view in action within the Lean theorem prover. The left-hand pane is the Lean source document and the right-hand pane is the infoview showing the context and expected type at the editor's cursor. There are two nested tooltips; one giving information about an expression in the infoview and the other on an expression within the first tooltip. The information that the tooltip provides is customisable, currently showing the type of the expression and a list of implicit arguments for the given expression.

DOM based on the HTML fragments received from the server. Lean ProofWidgets can also work over remote proving sessions (where the client editor is running on a different machine to the ITP server, possibly in another country).

4.1 Interactive Expressions

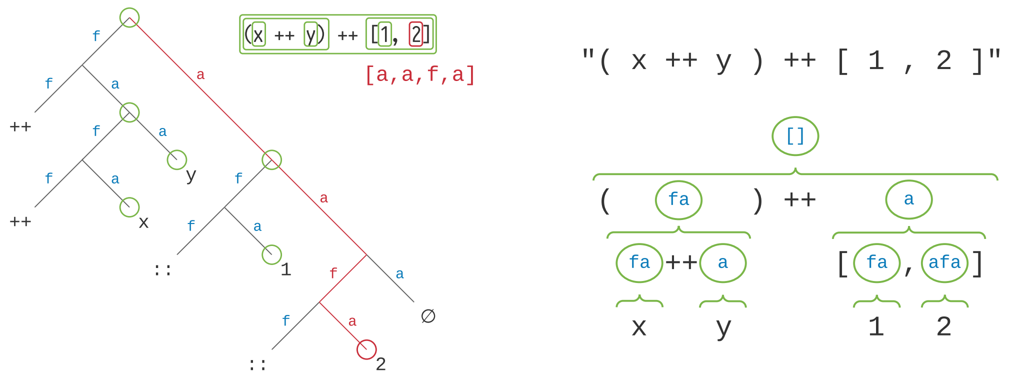
By "interactive expressions" we mean augmenting a printed expression string with a mapping structure such that the software can determine the correspondence between substrings and subexpressions. This mapping can then be used to create interactive term inspectors and tactics. As will be discussed in Section 5, pretty printing with this additional information is not a novel feature, however the way in which we have designed it makes these features available for producing proofs. Richly decorating expressions returned from the prover was first described by Bertot and Laurent as 'proof by pointing' [4].

An example of the interactive expression module in action is given in Figure 3: as one hovers over the various expressions and subexpressions in the infoview, one gets semantically correct highlighting for the expressions, and when you click on a subexpression, a tooltip appears containing type information. This tooltip is itself a component and so can also be interacted with, including opening a nested tooltip.

A number of other features are demonstrated in Figure 3:

- Hovering over subterms highlights the appropriate parts of the pretty printed string.
- The buttons in the top right of the tooltip activate effects including a "go to definition" button and a "copy type to clipboard" button.
- Expressions within the tooltip can also be explored in a nested tooltip. This is possible thanks to the state tracking system detailed in Section 3.2.

Note that the Lean editor already has features for displaying type information for the source document with the help of hover info. However, this tooltip mechanism is only textual (not interactive) and only works on expressions that have been written in the source document. Prior to ProofWidgets there was no way to inspect expressions as they appeared in the infoview.



(a) **(b)**

Figure 4 An expression tree for `(x ++ y) ++ [1,2]` is shown in Figure 4a. Each `f` or `a` above the lines is an expression coordinate. The red `[a,a,f,a]` example of an expression address, corresponding to the red line on the tree. Each green circle in the tree will pretty-print to a string independent of the expression tree above it. Figure 4b shows a `eformat` tree produced by pretty-printing the expression `(x ++ y) ++ [1,2]`. The green circles are `eformat.tag` constructors and the blue address text within is the relative address of the `tag` in relation to the `tag` above it. So that means that the full expression address for a subterm can be recovered by concatenating the `Address`es above it in the tree, for example the 2 subexpression is at `[] ++ [a] ++ [a,f,a] = [a,a,f,a]`.

All of the code which dictates the appearance and behaviour of the infoview widget is written in Lean and reloads dynamically when its code is changed. This means that users can produce their own custom tooltips and improve upon the infoview experience within their own Lean project.

In terms of performance, the generality of the ProofWidgets architecture means that it is possible for the user of the UI to execute long-running calculations or to render a large VDOM. This would result in creating a sluggish UI. However, for realistic use cases, such as producing a goal state widget as shown in Figure 3, the system should be responsive. Using the Chromium developer tools, the round trip time from a mouse pointer movement to updating the pixels on the screen was measured to be less than 80ms on an Intel i7 laptop from 2012 for a goal state widget with over 1000 VDOM nodes. This means that the system can easily handle events that need to be responsive without requiring more advanced hardware requirements than would be needed for ITP without ProofWidgets.

To support interactive expressions, we modified the Lean pretty-printer. Prior to our modifications, the pretty-printer would take an expression and a context for the expression and produce a member of the `format` type. This is implemented as a symbolic expression or "sexpr" *a la* LISP [17]. Our modification causes the pretty-printer to instead produce an instance of `eformat`. `eformat` is the same as `format` except that certain points in the sexpr tree are tagged with two pieces of information: the subexpression that produced the substring and an *expression address* indicating where the subexpression lies in the parent expression. The expression address is a list of *expression coordinates* used to reference subterms of an expression. By *expression coordinate*, we mean an enum that indexes the recursive arguments in the constructors for an expression. This is visualised in Figure 4a and Figure 4b. Listing 2 gives a simplified picture of the datatypes used to define expression coordinates and `eformat`.

The `eformat` tags act as a reversed source-map between the resulting sexpr and the original expression. This tagging also works beneath specialised syntax such as lists `[1,2,3]` and set comprehensions. This tagged string is used to create ProofWidgets that allow users

Listing 2 Simplified datatypes to demonstrate expression coordinates and `eformat`. Here the simplified `expr` datatype has four constructors for creating variables, function application, lambda abstraction and constants. Next, define coordinates `coord` on `expr`. Each constructor of `coord` corresponds to a different recursive argument in a constructor for `expr`. An `eformat` is a string with structural information present.

```
inductive expr
| var   : string → expr
| app   : expr → expr → expr
| lam   : string → expr → expr
| const : string → expr

inductive coord
| f | a | lam_body

def address := list coord

inductive eformat
| tag : expr → address → eformat → eformat
| append : eformat → eformat → eformat
| of_string : string → eformat
```

to interactively inspect various parts of the expression in the infoview. In the case of a subexpression being below a binder (e.g., in the body of a lambda expression) the pretty printer instantiates the de-Bruijn [7] variable with a dummy local variable, so the given subexpression doesn't contain free de-Bruijn variables and still typechecks without having to know the binders above the subexpression.

To render an interactive expression, we define a stateful component:[11]

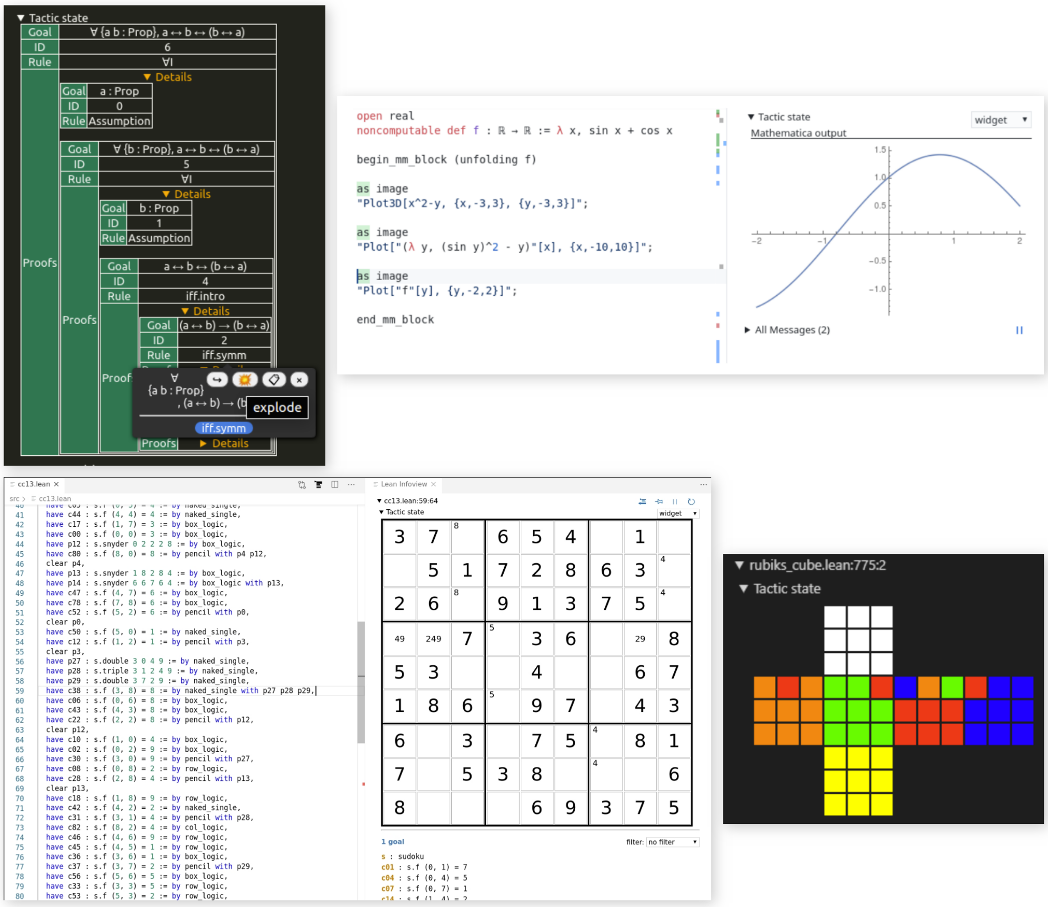
$$p : \texttt{component}\ (\texttt{tactic\_state} \times \texttt{expr})\ \texttt{empty}$$

The `tactic_state` object includes some contextual information such as metavariable context that are needed to print the expression correctly. The state of p includes an optional `address` of the expression. When the user hovers over a particular point in the printed `eformat`, the expression address corresponding to that part of the string is calculated using the `tag`s and this address is set as the state of the component. This address is then used to colour in the portion of the string that is currently hovered over by the user's mouse cursor which gives the semantic-aware highlighting effect.

4.2 Use within the Lean community

Our Lean implementation of the ProofWidgets framework has been adopted in to the leanprover-community fork of Lean 3 and mathlib, Lean's library of formalised mathematics [16]. The library is highly active and has hundreds of contributors using the widgets system[12] to render goal states.

[11] The sourcecode for this component within mathlib, the Lean mathematical library, can be found at `https://github.com/leanprover-community/mathlib/blob/master/src/tactic/interactive_expr.lean`.
[12] See `https://leanprover-community.github.io/mathlib_stats.html` for statistics on mathlib contributions.



```
open real
noncomputable def f : ℝ → ℝ := λ x, sin x + cos x

begin_mm_block (unfolding f)

as image
"Plot3D[x^2-y, {x,-3,3}, {y,-3,3}]";

as image
"Plot["(λ y, (sin y)^2 - y)"[x], {x,-10,10}]";

as image
"Plot["f"[y], {y,-2,2}]";

end_mm_block
```

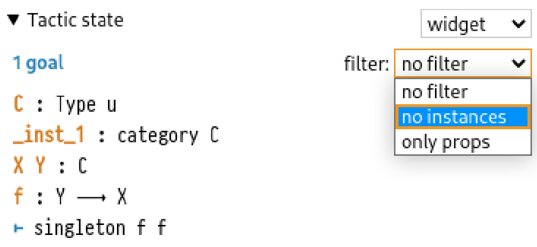
```
have c44 : s.f (4, 4) = 4 := by naked_single,
have c17 : s.f (1, 7) = 3 := by box_logic,
have c00 : s.f (0, 0) = 3 := by box_logic,
have p12 : s.snyder 0 2 2 2 8 := by box_logic,
have c80 : s.f (8, 0) = 8 := by pencil with p4 p12,
clear p4,
have p13 : s.snyder 1 8 2 8 4 := by box_logic,
have p14 : s.snyder 6 6 7 6 4 := by box_logic with p13,
have c47 : s.f (4, 7) = 6 := by box_logic,
have c78 : s.f (7, 8) = 6 := by box_logic,
have c52 : s.f (5, 2) = 6 := by pencil with p0,
clear p0,
have c50 : s.f (5, 0) = 1 := by naked_single,
have c12 : s.f (1, 2) = 1 := by pencil with p3,
clear p3,
have p27 : s.double 3 0 4 9 := by naked_single,
have p28 : s.triple 3 1 2 4 9 := by naked_single,
have p29 : s.double 3 7 2 9 := by naked_single,
have c38 : s.f (3, 8) = 8 := by naked_single with p27 p28 p29,
have c06 : s.f (0, 6) = 8 := by box_logic,
have c43 : s.f (4, 3) = 8 := by box_logic,
have c22 : s.f (2, 2) = 8 := by pencil with p12,
clear p12,
have c10 : s.f (1, 0) = 4 := by box_logic,
have c02 : s.f (0, 2) = 9 := by box_logic,
have c30 : s.f (3, 0) = 9 := by pencil with p27,
have c08 : s.f (0, 8) = 2 := by row_logic,
have c28 : s.f (2, 8) = 4 := by pencil with p13,
clear p13,
have c18 : s.f (1, 8) = 9 := by row_logic,
have c42 : s.f (4, 2) = 2 := by naked_single,
have c31 : s.f (3, 1) = 4 := by pencil with p28,
have c82 : s.f (8, 2) = 4 := by col_logic,
have c46 : s.f (4, 6) = 9 := by row_logic,
have c45 : s.f (4, 5) = 1 := by row_logic,
have c36 : s.f (3, 6) = 1 := by box_logic,
have c37 : s.f (3, 7) = 2 := by pencil with p29,
have c56 : s.f (5, 6) = 5 := by box_logic,
have c33 : s.f (3, 3) = 5 := by row_logic,
have c53 : s.f (5, 3) = 2 := by row_logic,
```

```
1 goal                                  filter: no filter
s : sudoku
c01 : s.f (0, 1) = 7
c04 : s.f (0, 4) = 5
c07 : s.f (0, 7) = 1
```

■ **Figure 5** Community-made projects using ProofWidgets. In order, these are: `explode` proof viewer for inspecting proof terms by Minchao Wu; *Mathematica bridge* for viewing plots using Mathematica by Robert Y Lewis and Minchao Wu; *Sudoku solver and visualiser* by Markus Himmel; *Rubik's cube formalisation* by Kendall Frey.

The implementation in Lean has already been picked up by the community to make a wide variety of graphical interface solutions which can be viewed in Figure 5. Of particular note is the Mathematica bridge by Lewis and Wu [13], which connects Lean to Wolfram Mathematica and uses our framework to show Lean functions plotted by Mathematica.

ProofWidgets are also in use within mathlib to rapidly prototype new user interface features as they are requested. As a concrete example: in the development of the Lean VSCode extension, it was requested that it should be possible to filter some of the variables in the goal state to declutter the output window (see Figure 6). This was achieved by reparsing the textual goal state emitted by the Lean server component and removing the filtered items using regular expressions. Using ProofWidgets, it was required to add some specific code for the VSCode client – supporting such a feature in other editors would require rewriting the filtering code. Additionally, if the Lean server changes how the goal state is formatted, the filtering code would need to be rewritten. Even if an API which allows more semantic access to the expression structure is used, such as SerAPI [10], there is still the problem that the filtering code has to be written multiple times for each supported editor. Using ProofWidgets, the filtering code can be written once *in Lean itself* and it then works in



```
▼ Tactic state                          widget

1 goal                          filter: no filter
                                        no filter
C : Type u                              no instances
_inst_1 : category C                    only props
X Y : C
f : Y ⟶ X
⊢ singleton f f
```

Figure 6 Example widget for filtering goal states, an example of being able to implement new UI features within Lean.

any editor that supports the widgets API (at the time of writing VSCode and a prototype version of the web editor). Furthermore, Lean users are free to make any custom tweaks to the UI without needing to make any changes to the editor code.

5 Related work

We now list some of the other tools and systems for creating user interfaces within the context of interactive theorem proving and how they relate to our work.

Isabelle's *Prover IDE* (PIDE) was developed by Wenzel in Scala and is based on the jEdit text editor [21]. It richly annotates Isabelle documents and proof states to provide inline documentation, interactive and customisable commands, and automatic insertion of text, among other features. Isabelle's development environment allows users to code their own UI in Scala, which performs a similar role to ProofWidgets. PIDE broadly shares the design goals of ProofWidgets, but approaches the problem differently.

An advantage of the ProofWidgets approach compared to PIDE's is that the API between the editor and the prover can be smaller since, in ProofWidgets, the appearance and behaviour is entirely dictated by the server. In contrast, the implementation of PIDE is tightly coupled to the bundled jEdit editor, which has some advantages over ProofWidgets in that it gives more control to the developer to create new UIs. The downside of PIDE's approach here is that one must maintain this editor and so supporting any other editor with feature-parity becomes difficult. ProofWidgets also makes use of modern web technology which is ubiquitously supported. In contrast, PIDE uses a Java UI library called Swing. Creating custom UIs in PIDE requires coding in both Scala and StandardML, and the result does not easily generalise to the VSCode Isabelle extension.

There have been some recent efforts to support VSCode as a client editor for Isabelle files [22]. A web-based client for Isabelle, called *Clide* [15] was developed, although ultimately it provides only a subset of the functionality of the jEdit version.

SerAPI [10] is a library for machine-machine interaction with the Coq theorem prover. This supports some web-based IDE projects such as *jsCoq* [11] and, recently, Alectyron [18]. Alectyron enables users to embed web-based representations of data. SerAPI contrasts to ProofWidgets in that it expects another program to be responsible for displaying graphical elements such as goal states and visualisations; in the ProofWidgets architecture all of the UI appearance and behaviour code is also written in Lean, and the web-app client can render general UIs emitted by the system.

There are some older GUI-centric theorem provers that should be mentioned: *LΩUI* [19], *HyperProof* [3] and *XBarnacle* [14]. These tools were all highly innovative for including graphical and multimodal representations of proofs, however the code for these has succumbed to bit rot, to the extent that they can only be viewed through the screenshots that were included with the papers. Source code for Ωmega and CLAM (which *LΩUI* and XBarnacle use respectively) can be found in the Theorem Prover Museum.[13]

Vicary's *Globular* [2] and Breitner's *Incredible Proof Machine* [5] also inspired our work. These tools are natively web-based and offer a visual representation of the proof state for users to manipulate. A lot of the motivation behind ProofWidgets was to bring some of this visual pixiedust to a more general, heavyweight proof assistants.

6 Conclusion and future work

We designed the ProofWidgets framework: a general client/server protocol, a functional UI programming API and a system for tagging pretty-printed expressions. The framework is implemented in Lean to allow for rapid development of new modalities of interaction with provers, all within the metalanguage of Lean. This enables a faster development cycle for improving the UI of Lean which has a direct impact on the users of ITP. We hope that these benefits can be brought to other interactive theorem provers in the future.

6.1 Future work

In terms of performance, in order to produce responsive interfaces that use long-running tactics (e.g., searching a library or running a solver) it will be necessary to provide a mechanism for allowing concurrency. At the moment, if a long-running operation is needed to produce output, this will block the rendering process and the UI will become unresponsive for the length of the operation. Currently, Lean has a `task` type which represents a 'promise' to the result of a long-running operation, which could be utilised to solve this problem. This could be cleanly incorporated in ProofWidgets by providing an additional hook `with_task` (see Listing 3):

Listing 3 Adding concurrency to components.

```
component.with_task
  (get_task : π → Task τ)
  : (Component ((Option τ × π) α) → (Component π α)
```

Here, `get_task` returns a long-running task object and the props for the inner component transition from none to some $t : \tau$ upon the completion of the task. Cancelling a task is implemented simply by causing a rerender.

The next implementation project is to support Lean 4. Lean 4 has a bootstrapped compiler, so the reconciling code can be written in Lean 4 itself without having to modify the core codebase as was necessary for Lean 3. Lean 4 has an overhauled, extensible parser system [20] which could be used to create an HTML-like domain-specific language directly within Lean.

[13] `https://theoremprover-museum.github.io/`

References

1 Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, 2013. `doi:10.1145/2501654.2501666`.

2 Krzysztof Bar, Aleks Kissinger, and Jamie Vicary. Globular: an online proof assistant for higher-dimensional rewriting. *Log. Methods Comput. Sci.*, 14(1), 2018. `doi:10.23638/LMCS-14(1:8)2018`.

3 Jon Barwise and John Etchemendy. Hyperproof: Logical reasoning with diagrams. In *Working Notes of the AAAI Spring Symposium on Reasoning with Diagrammatic Representations*, 1992. URL: `https://www.aaai.org/Papers/Symposia/Spring/1992/SS-92-02/SS92-02-016.pdf`.

4 Yves Bertot and Laurent Théry. A generic approach to building user interfaces for theorem provers. *J. Symb. Comput.*, 25(2):161–194, 1998. `doi:10.1006/jsco.1997.0171`.

5 Joachim Breitner. Visual theorem proving with the incredible proof machine. In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, volume 9807 of *Lecture Notes in Computer Science*, pages 123–139. Springer, 2016. `doi:10.1007/978-3-319-43144-4_8`.

6 Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for guis. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 411–422. ACM, 2013. `doi:10.1145/2491956.2462161`.

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. North-Holland, 1972. URL: `http://alexandria.tue.nl/repository/freearticles/597619.pdf`.

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

9 Conal Elliott and Paul Hudak. Functional reactive animation. In Simon L. Peyton Jones, Mads Tofte, and A. Michael Berman, editors, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997*, pages 263–273. ACM, 1997. `doi:10.1145/258948.258973`.

10 Emilio Jesús Gallego Arias. Serapi: Machine-friendly, data-centric serialization for coq. Technical report, MINES ParisTech, 2016. URL: `https://core.ac.uk/download/pdf/51221893.pdf`.

11 Emilio Jesús Gallego Arias, Benoît Pin, and Pierre Jouvelot. jscoq: Towards hybrid theorem proving interfaces. In Serge Autexier and Pedro Quaresma, editors, *Proceedings of the 12th Workshop on User Interfaces for Theorem Provers*, volume 239 of *Electronic Proceedings in Theoretical Computer Science*, pages 15–27. Open Publishing Association, 2017. `doi:10.4204/EPTCS.239.2`.

12 Mike Gordon. From LCF to HOL: a short history. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 169–186. The MIT Press, 2000.

13 Robert Y. Lewis. An extensible ad hoc interface between lean and mathematica. In Catherine Dubois and Bruno Woltzenlogel Paleo, editors, *Proceedings of the Fifth Workshop on Proof eXchange for Theorem Proving, PxTP 2017, Brasília, Brazil, 23-24 September 2017*, volume 262 of *EPTCS*, pages 23–37, 2017. `doi:10.4204/EPTCS.262.4`.

14 Helen Lowe and David Duncan. Xbarnacle: Making theorem provers more accessible. In William McCune, editor, *Automated Deduction - CADE-14, 14th International Conference on Automated Deduction, Townsville, North Queensland, Australia, July 13-17, 1997, Proceedings*, volume 1249 of *Lecture Notes in Computer Science*, pages 404–407. Springer, 1997. `doi:10.1007/3-540-63104-6_39`.

15 Christoph Lüth and Martin Ring. A web interface for isabelle: The next generation. In Jacques Carette, David Aspinall, Christoph Lange, Petr Sojka, and Wolfgang Windsteiger, editors, *Intelligent Computer Mathematics - MKM, Calculemus, DML, and Systems and Projects 2013, Held as Part of CICM 2013, Bath, UK, July 8-12, 2013. Proceedings*, volume 7961 of *Lecture Notes in Computer Science*, pages 326–329. Springer, 2013. `doi:10.1007/978-3-642-39320-4_22`.

16 The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 367–381, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3372885.3373824`.

17 John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, 1960. `doi:10.1145/367177.367199`.

18 Clément Pit-Claudel. Untangling mechanized proofs. In Ralf Lämmel, Laurence Tratt, and Juan de Lara, editors, *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020*, pages 155–174. ACM, 2020. `doi:10.1145/3426425.3426940`.

19 Jörg H. Siekmann, Stephan M. Hess, Christoph Benzmüller, Lassaad Cheikhrouhou, Armin Fiedler, Helmut Horacek, Michael Kohlhase, Karsten Konrad, Andreas Meier, Erica Melis, Martin Pollet, and Volker Sorge. LOUI: Lovely omega user interface. *Formal Aspects Comput.*, 11(3):326–342, 1999. `doi:10.1007/s001650050053`.

20 Sebastian Ullrich and Leonardo de Moura. Beyond notations: Hygienic macro expansion for theorem proving languages. 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 167–182. Springer, 2020. `doi:10.1007/978-3-030-51054-1_10`.

21 Makarius Wenzel. Isabelle/jedit - A prover IDE within the PIDE framework. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors, *Intelligent Computer Mathematics - 11th International Conference, AISC 2012, 19th Symposium, Calculemus 2012, 5th International Workshop, DML 2012, 11th International Conference, MKM 2012, Systems and Projects, Held as Part of CICM 2012, Bremen, Germany, July 8-13, 2012. Proceedings*, volume 7362 of *Lecture Notes in Computer Science*, pages 468–471. Springer, 2012. `doi:10.1007/978-3-642-31374-5_38`.

22 Makarius Wenzel. Isabelle/pide after 10 years of development. In *UITP workshop: User Interfaces for Theorem Provers.*, 2018. URL: `https://sketis.net/wp-content/uploads/2018/08/isabellepide-uitp2018.pdf`.

A Formalization of Dedekind Domains and Class Groups of Global Fields

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

Sander R. Dahmen
Department of Mathematics, Vrije Universiteit Amsterdam, The Netherlands

Ashvni Narayanan
London School of Geometry and Number Theory, UK

Filippo A. E. Nuccio Mortarino Majno di Capriglio
Univ Lyon, Université Jean Monnet Saint-Étienne, CNRS UMR 5208, Institut Camille Jordan, F-42023 Saint-Étienne, France

Abstract

Dedekind domains and their class groups are notions in commutative algebra that are essential in algebraic number theory. We formalized these structures and several fundamental properties, including number theoretic finiteness results for class groups, in the Lean prover as part of the mathlib mathematical library. This paper describes the formalization process, noting the idioms we found useful in our development and mathlib's decentralized collaboration processes involved in this project.

2012 ACM Subject Classification Mathematics of computing → Mathematical software; Security and privacy → Logic and verification

Keywords and phrases formal math, algebraic number theory, commutative algebra, Lean, mathlib

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.5

Supplementary Material Full source code of the formalization is part of mathlib. Copies of the source files relevant to this paper are available in a separate repository.
Software: `https://github.com/lean-forward/class-number`
archived at `swh:1:dir:3e46fd736dbb76b6e7be7f31ebc973f1a1c0237d`

Funding *Anne Baanen*: NWO Vidi grant No. 016.Vidi.189.037, Lean Forward
Sander R. Dahmen: NWO Vidi grant No. 639.032.613, New Diophantine Directions
Ashvni Narayanan: EPSRC Grant EP/S021590/1 (UK)

Acknowledgements We would like to thank Jasmin Blanchette and the anonymous reviewers for useful comments on previous versions of the manuscript, which found their way into this paper.
A. N. would like to thank Prof. Kevin Buzzard for his constant support and encouragement, and for introducing her to the other co-authors.
A. N. and F. N. wish to express their deepest gratitude to Anne Baanen for the generosity shown along all stages of the project. Without Anne's never-ending patience, it would have been impossible for them to contribute to this project, and to overcome several difficulties.
Finally, we would like to thank the whole mathlib community for invaluable advice all along the project.

1 Introduction

In its basic form, number theory studies properties of the integers $\mathbb{Z}$ and its fraction field, the rational numbers $\mathbb{Q}$. Both for the sake of generalization, as well as for providing powerful techniques to answer questions about the original objects $\mathbb{Z}$ and $\mathbb{Q}$, it is worthwhile to study finite extensions of $\mathbb{Q}$, called *number fields*, as well as their *rings of integers* (Section 2), whose relations mirror the way $\mathbb{Q}$ contains $\mathbb{Z}$ as a subring. In this paper, we describe our project aiming at formalizing these notions and some of their important properties. Our goal, however, is not to get to the definitions and properties as quickly as possible, but rather to lay the foundations for future work, as part of a natural and more general theory as we shall explain below.

In particular, our project resulted in formalized definitions and elementary properties of number fields and their rings of integers (Section 3.3), Dedekind domains (Section 4), and the ideal class group and class number (Section 7). Apart form the very basics concerning number fields, these concepts were not formalized before as far as we are aware of. We note that our formal definition of the class number is an essential requirement for the use of theorem provers in modern number theory research. The main proofs that we formalized show that two definitions of Dedekind domains are equivalent (Section 4.3), that the ring of integers is a Dedekind domain (Section 6) and that the class group of a number field is finite (Section 7). In fact, most of our results for number fields are also obtained in the more general setting of *global fields*.

Our work is developed as part of the mathematical library mathlib [25] for the Lean 3 theorem prover [9]. The formal system of Lean is a dependent type theory based on the calculus of inductive constructions, with a proof-irrelevant impredicative universe `Prop` at the bottom of a noncumulative hierarchy of universes `Prop : Type : Type 1 : Type 2 : ...` ; "an arbitrary `Type u`" is abbreviated as `Type*`. Other important characteristics of Lean as used in mathlib are the use of quotient types, ubiquitous classical reasoning and the use of typeclasses to define the hierarchy of algebraic structures.

Organizationally, mathlib is characterized by a distributed and decentralized community of contributors, a willingness to refactor its basic definitions, and a preference for small yet complete contributions over larger projects added all at once. In this project, as part of the development of mathlib, we followed this philosophy by contributing pieces of our work as they were finished. We, in turn, used results contributed by others after the start of the project. At several points, we had just merged a formalization into mathlib that another contributor needed, immediately before they contributed a result that we needed. Due to the decentralized organization and fluid nature of contributions to mathlib, its contents are built up of many different contributions from over 100 different authors. Attributing each formalization to a single set of main authors would not do justice to all others whose additions and tweaks are essential to its current use. Therefore, we will make clear whether a contribution is part of our project or not, but we will not stress whom we consider to be the main authors.

The source files of the formalization are currently in the process of being merged into mathlib. The up-to-date development branch is publically available.[1] We also maintain a repository[2] containing the source code referred to in this paper.

2 Mathematical background

Let us now introduce some of the main objects we study, described informally. We assume some familiarity with basic ring and field theory.

[1] `https://github.com/leanprover-community/mathlib/tree/dedekind-domain-dev`
[2] `https://github.com/lean-forward/class-number`

A *number field* K is a finite extension of the field $\mathbb{Q}$, and as such has the structure of a finite dimensional vector space over $\mathbb{Q}$; its dimension is called the *degree* of K. The easiest example is $\mathbb{Q}$ itself, and the two-dimensional cases are given by the quadratic number fields $\mathbb{Q}(\sqrt{d}) = \{a + b\sqrt{d} : a, b \in \mathbb{Q}\}$ where $d \in \mathbb{Z}$ is not a square. For an interesting cubic example, let α_0 be the unique real number satisfying $\alpha_0^3 + \alpha_0^2 - 2\alpha_0 + 8 = 0$. It gives rise to the number field $\mathbb{Q}(\alpha_0) = \{a + b\alpha_0 + c\alpha_0^2 : a, b, c \in \mathbb{Q}\}$. In general, taking any root α of an irreducible polynomial of degree n over $\mathbb{Q}$ yields a number field of degree n: $\mathbb{Q}(\alpha) = \{c_0 + c_1\alpha + \ldots + c_{n-1}\alpha^{n-1} : c_0, c_1, \ldots, c_{n-1} \in \mathbb{Q}\}$, and, up to isomorphism, these are all the number fields of degree n.

The *ring of integers* $\mathcal{O}_K$ of a number field K is defined as the integral closure of $\mathbb{Z}$ in K, namely

$$\mathcal{O}_K := \big\{x \in K : f(x) = 0 \text{ for some } \textit{monic} \text{ polynomial } f \text{ with integer coefficients}\big\},$$

where we recall that a polynomial is called *monic* if its leading coefficient equals 1. While it might not be immediately obvious that $\mathcal{O}_K$ is a ring, this follows from general algebraic properties of integral closures. Some examples of $\mathcal{O}_K$ are the following. Taking $K = \mathbb{Q}$, we get $\mathcal{O}_K = \mathbb{Z}$ back. For $K = \mathbb{Q}(i) = \mathbb{Q}(\sqrt{-1})$ we get that $\mathcal{O}_K$ is the ring of Gaussian integers $\mathbb{Z}[i] = \{a + bi : a, b \in \mathbb{Z}\}$. But for $K = \mathbb{Q}(\sqrt{5})$ we do *not* simply get $\mathbb{Z}[\sqrt{5}] = \{a + b\sqrt{5} : a, b \in \mathbb{Z}\}$ as $\mathcal{O}_K$, since the golden ratio $\varphi := (1 + \sqrt{5})/2 \notin \mathbb{Z}[\sqrt{5}]$ satisfies the monic polynomial equation $\varphi^2 - \varphi - 1 = 0$; hence by definition, $\varphi \in \mathcal{O}_K$. It turns out that $\mathcal{O}_K = \mathbb{Z}[\varphi] = \{a + b\varphi : a, b \in \mathbb{Z}\}$. Finally, if $K = \mathbb{Q}(\alpha_0)$ with α_0 as before, then $\mathcal{O}_K = \{a + b\alpha_0 + c(\alpha_0 + \alpha_0^2)/2 : a, b, c \in \mathbb{Z}\}$, illustrating that explicitly writing down $\mathcal{O}_K$ can quickly become complicated. Further well-known rings of integers are the Eisenstein integers $\mathbb{Z}[(1 + \sqrt{-3})/2]$ and the ring $\mathbb{Z}[\sqrt{2}]$.

Thinking of $\mathcal{O}_K$ as a generalization of $\mathbb{Z}$, it is natural to ask which of its properties still hold in $\mathcal{O}_K$ and, when this fails, if a reasonable weakening does.

An important property of $\mathbb{Z}$ is that it is a principal ideal domain (PID), meaning that every ideal is generated by one element. This implies that every nonzero nonunit element can be written as a finite product of prime elements, which is unique up to reordering and multiplying by ± 1: a ring where this holds is called a unique factorization domain, or UFD. For example, 6 can be factored in primes in 4 equivalent ways, namely $6 = 2 \cdot 3 = 3 \cdot 2 = (-2) \cdot (-3) = (-3) \cdot (-2)$. In fact, the previously mentioned examples of rings of integers are UFDs, but this is certainly not true for all rings of integers. For example, unique factorization *does not* hold in $\mathbb{Z}[\sqrt{-5}]$: it is easy to prove that $6 = 2 \cdot 3$ and $6 = (1 + \sqrt{-5})(1 - \sqrt{-5})$ provide two essentially different ways to factor 6 into prime elements of $\mathbb{Z}[\sqrt{-5}]$.

As it turns out, there is a way to remedy this. Namely, by considering factorization of *ideals* instead of elements: given a number field K, with ring of integers $\mathcal{O}_K$, a beautiful and classical result by Dedekind shows that every nonzero ideal of $\mathcal{O}_K$ can be factored as a product of prime ideals in a unique way, up to reordering.

Although unique factorization in terms of ideals is of great importance, it is still interesting, and sometimes necessary, to also consider factorization properties in terms of elements. We mentioned that unique factorization in $\mathbb{Z}$ follows from the fact that every ideal is generated by a single element. Now, it is convenient to extend the notion of ideals of $\mathbb{Z}$ to that of *fractional ideals*. These are additive subgroups of $\mathbb{Q}$ of the form $\frac{1}{d}I$ with I an ideal of $\mathbb{Z}$ and d a nonzero integer. When the distinction is important, we refer to an ideal $I \subseteq \mathbb{Z}$ as an *integral ideal*. The nonzero fractional ideals of $\mathbb{Z}$ naturally form a multiplicative group (whereas there is no integral ideal $I \subseteq \mathbb{Z}$ such that $I * (2\mathbb{Z}) = (1)$). The statement that every ideal is generated by a single element translates to the fact that the quotient group of nonzero fractional ideals modulo $\mathbb{Q}^\times$ is trivial (where $\frac{a}{b} \in \mathbb{Q}^\times$ corresponds to $\frac{1}{b}a\mathbb{Z}$, and as usual, the multiplicative group of invertible elements of a ring R is denoted by $R^\times$).

It turns out that this quotient group can be defined for every ring of integers $\mathcal{O}_K$. The fundamental theoretical notion beneath this construction is that of Dedekind domain: these are integral domains D which are Noetherian (every ideal of D is finitely generated), integrally closed (if an element x in $\operatorname{Frac} D$ (the fraction field of D) is a root of a monic polynomial with coefficients in D, then actually $x \in D$), and of Krull dimension at most 1 (every nonzero prime ideal of D is maximal). It can be proved that the nonzero fractional ideals of D again form a group, and the quotient of this group by the image of the natural embedding of $(\operatorname{Frac} D)^\times$ is called the (*ideal*) *class group* $\mathcal{C}l_D$.

What is arithmetically crucial is the theorem ensuring that the ring of integers $\mathcal{O}_K$ of every number field K is a Dedekind domain, and that in this case the class group $\mathcal{C}l_{\mathcal{O}_K}$ is actually *finite*. In particular, $\mathcal{C}l_{\mathcal{O}_K}$ can be seen as "measuring" how far ideals of $\mathcal{O}_K$ are from being generated by a single element and, consequently, as a measure of the failure of unique factorization. The order of $\mathcal{C}l_{\mathcal{O}_K}$ is called *the class number* of K. Intuitively, then, the smaller the class number, the fewer factorizations are possible. In particular, the class number of K is equal to 1 if and only if $\mathcal{O}_K$ is a UFD.

The statements in the previous paragraph also hold for *function fields*, namely fields which are finite extensions of $\mathbb{F}_q(t) \simeq \operatorname{Frac} \mathbb{F}_q[t]$, where $\mathbb{F}_q[t]$ stands for the ring of univariate polynomials (in a free variable t) with coefficients in a finite field with q elements $\mathbb{F}_q$. Recall that when q is a prime number, $\mathbb{F}_q$ is simply the field $\mathbb{Z}/q\mathbb{Z}$. A field which is either a number field or a function field is called a *global field*.

In the next sections we will describe the formalization of the above concepts.

3 Number fields, global fields and rings of integers

We refer the reader to Section 2 for the mathematical background needed in this section.

We formalized number fields as the following typeclass:

```
class is_number_field (K : Type*) [field K] : Prop :=
[cz : char_zero K] [fd : finite_dimensional ℚ K]
```

The *class* keyword declares a structure type (in other words, a type of records) and enables typeclass inference for terms of this type. Round brackets mark parameters explicitly supplied by the user, such as `(K : Type*)`, square brackets mark instance parameters inferred by the typeclass system, such as `[field K]`. The condition `[cz : char_zero K]` states that K has characteristic zero, so the canonical ring homomorphism $\mathbb{Z} \to K$ is an embedding. This implies that there is a $\mathbb{Q}$-algebra structure on K (found by typeclass instance search), endowing K with the $\mathbb{Q}$-vector space structure used in the `[fd : finite_dimensional ℚ K]` hypothesis.

Typeclasses were originally introduced in Haskell as a mechanism for operator overloading [28], and are used throughout Lean's core library and mathlib to endow types with mathematical structures consisting of both operators and their properties [25]. The typeclass system will automatically infer values for instance parameters, by searching for values of the appropriate type among the local parameters or declarations marked as an *instance* [2, §10].

We defined the function field K over a finite field $\mathbb{F}_q$ using the following typeclass:

```
class is_function_field_over {𝔽q F : Type*} [field 𝔽q] [fintype 𝔽q]
  [field F] (f : fraction_map (polynomial 𝔽q) F) (K : Type*) [field K]
  [algebra f.codomain K] : Prop :=
[fd : finite_dimensional f.codomain K]
```

Curly brackets mark implicit parameters inferred through unification, such as `{`$\mathbb{F}_q$ `F : Type*}`. The map `f` witnesses that F is a fraction field of the polynomial ring $\mathbb{F}_q[t]$, the notation `f.codomain` endows F with the $\mathbb{F}_q[t]$-algebra structure of $\mathbb{F}_q(t)$. We present a more detailed analysis of `fraction_map` in Section 3.5.

3.1 Field extensions

The definition of `is_number_field` illustrates our treatment of field extensions. A field L containing a subfield K is said to be a field extension L/K. Often we encounter towers of field extensions: we might have that $\mathbb{Q}$ is contained in K, K is contained in L, L is contained in an algebraic closure $\overline{K}$ of K, and $\overline{K}$ is contained in $\mathbb{C}$. We might formalize this situation by viewing $\mathbb{Q}$, K, L and $\overline{K}$ as sets of complex numbers $\mathbb{C}$ and defining field extensions as subset relations between these subfields. This way, no coercions need to be inserted in order to map elements of one field into a larger field. Unfortunately, we can only avoid coercions as far as we are able to stay within one largest field. For example, the definition of complex numbers depends on many results for rational numbers, which would need to be proved again, or transported, for the subfield of $\mathbb{C}$ isomorphic to $\mathbb{Q}$.

Instead, we formalized results about field extensions through parametrization. The fields K and L can be arbitrary types and the hypothesis "L is a field extension of K" is represented by an instance parameter `[algebra K L]` denoting a K-algebra structure on L. There are multiple possible K-algebra structures for a field L and Lean does not enforce uniqueness of typeclass instances, but the mathlib maintainers try to ensure all instances that can be inferred are definitionally equal. The `algebra` structure provides us with a canonical ring homomorphism `algebra_map K L` : $K \to L$; this map is injective because K and L are fields. In other words, field extensions are given by their canonical embeddings.

3.2 Scalar towers

The main drawback of using arbitrary embeddings to represent field extensions is that we need to prove that these maps commute. For example, we might start with a field extension $L/\mathbb{Q}$, then define a subfield K of L, resulting in a tower of extensions $L/K/\mathbb{Q}$. In such a tower, the map $\mathbb{Q} \to L$ should be equal to the composition $\mathbb{Q} \to K$ followed by $K \to L$. Such an equality cannot always be achieved by defining the map $\mathbb{Q} \to L$ to be this composition: in the example, the map $\mathbb{Q} \to K$ depends on the map $\mathbb{Q} \to L$.

The solution in mathlib is to parametrize over all three maps, as long a there is also a proof of coherence: a hypothesis of the form "$L/K/F$ is a tower of field extensions" is translated into three instance parameters `[algebra F K]`, `[algebra K L]` and `[algebra F L]`, along with a parameter `[is_scalar_tower F K L]` expressing that the maps commute.

The `is_scalar_tower` typeclass derives its name from its applicability to any three types between which exist scalar multiplication operations:

```
class is_scalar_tower (M N α : Type*)
  [has_scalar M N] [has_scalar N α] [has_scalar M α] : Prop :=
(smul_assoc : ∀ (x : M) (y : N) (z : α), (x · y) · z = x · (y · z))
```

For example, if R is a ring, A is an R-algebra and M an A-module, we can state that M is also an R-module by adding a `[is_scalar_tower R A M]` parameter. Since `x · y` for an R-algebra A is defined as `algebra_map R A x * y`, applying `smul_assoc` for each $x : K$ with $y = (1 : L)$ and $z = (1 : F)$ shows that the `algebra_maps` indeed commute.

Common is_scalar_tower instances are declared in mathlib, such as for the maps $R \to S \to A$ when S is a R-subalgebra of A. The effect is that almost all coherence proof obligations are automated through typeclass instance search. Only when defining a new algebra structure were we required to supply the is_scalar_tower instances ourselves. Our reliance on typeclasses did not cause any noticeable slowness in proof checking: there was no instance that should be found but could not due to timeouts.

3.3 Rings of integers

When K is a number field (defined as a field satifying is_number_field), the ring $\mathcal{O}_K$ of integers in K is defined as the integral closure of $\mathbb{Z}$ in K. This is the subring containing those $x : K$ that are the root of a monic polynomial with coefficients in $\mathbb{Z}$:

```
def number_field.ring_of_integers (K : Type*) [field K]
  [is_number_field K] : subalgebra ℤ K :=
integral_closure ℤ K
```

where integral_closure was previously defined in mathlib.

When K is a function field over the finite field $\mathbb{F}_q$, we defined $\mathcal{O}_K$ analogously as integral_closure (polynomial $\mathbb{F}_q$) K. To treat both definitions of ring of integers on an equal footing, we will work with the integral closure of any principal ideal domain when possible.

3.4 Subobjects

The ring of integers is one example of a subobject, such as a subfield, subring or subalgebra, defined through a characteristic predicate. In mathlib, subobjects are "bundled", in the form of a structure comprising the carrier set and proofs showing the carrier set is closed under the relevant operations. Bundled subobjects provide similar benefits as bundled morphisms; the choice for the latter is explained in the mathlib overview paper [25].

Two new subobjects that we defined in our development were subfield as well as intermediate_field. We defined a subfield of a field K as a subset of K that contains 0 and 1 and is closed under addition, negation, multiplication and taking inverses. If L is a field extension of K, we defined an intermediate field as a subfield that is also a K-subalgebra: a subfield that contains the image of algebra_map K L. Other examples of subobjects available in mathlib are submonoids, subgroups and submodules (with ideals as a special case of submodules).

The new definitions found immediate use: soon after we contributed our definition of intermediate_field to mathlib, the Berkeley Galois theory group used it in a formalization of the primitive element theorem. Soon after the primitive element theorem was merged into mathlib, we used it in our development of the trace form. This anecdote illustrates the decentralized development style of mathlib, with different groups and people building on each other's results in a collaborative process.

By providing a coercion from subobjects to types, sending a subobject S to the subtype of all elements of S, and putting typeclass instances on this subtype, we could reason about inductively defined rings such as $\mathbb{Z}$ and subrings such as integral_closure ℤ K uniformly. If S : subfield K, there is a canonical ring embedding, the map that sends $x : S$ to K by "forgetting" that $x \in S$, and we registered this map as an algebra S K instance, also allowing us to treat field extensions of the form $\mathbb{Q} \to \mathbb{C}$ and subfields uniformly. Similarly, for F : intermediate_field K L, we defined the corresponding algebra K F, algebra F L and is_scalar_tower K F L instances.

3.5 Fields of fractions

The fraction field $\operatorname{Frac} R$ of an integral domain R can be defined explicitly as a quotient type as follows: starting from the set of pairs (a, b) with $a, b \in R$ such that $b \neq 0$, one quotients by the equivalence relation generated by $(\alpha a, \alpha b) \sim (a, b)$ for all $\alpha \neq 0 : R$, writing the equivalence class of (a, b) as $\frac{a}{b}$. It can easily be proved that the ring structure on R extends uniquely to a field structure on $\operatorname{Frac} R$; in mathlib this construction is called `fraction_ring R`. When $R = \mathbb{Z}$, this yields the traditional description of $\mathbb{Q}$ as the set of equivalence classes of fractions, where $\frac{2}{3} = \frac{-4}{-6}$, etc. The drawback of this construction is that there are many other fields that can serve as the field of fractions for the same ring. Consider the field $\{z \in \mathbb{C} : \Re z \in \mathbb{Q}, \Im z \in \mathbb{Q}\}$, which is isomorphic to $\operatorname{Frac}(\mathbb{Z}[i])$ but not definitionally equal to it.

The strategy used in mathlib is to rather allow for many different *fraction fields* of our given integral domain R, as fields K along with an injective *fraction map* $f \colon R \to K$ which witnesses that all elements of K are "fractions" of elements of R, and to parametrize every result over the choice of f. In the definition used by mathlib, a fraction map is a special case of a *localization map*. Different localizations restrict the denominators to different multiplicative submonoids of $R \setminus \{0\}$.

The conditions on f imply that K is the smallest field containing R, expressed by the following unique mapping property. If $g \colon R \to A$ is an injective map to a ring A such that $g(x)$ has a multiplicative inverse for all $x \neq 0 : R$, then it can be extended uniquely to a map $K \to A$ compatible with f and g. In particular, if $f_1 \colon R \to K_1$ and $f_2 \colon R \to K_2$ are fraction maps, they induce an isomorphism $K_1 \simeq K_2$. The construction of $\operatorname{Frac} R$ then results in *a* field of fractions (with fraction map `fraction_ring.of R`) rather than *the* field of fractions.

This comes at a price: informally, at any given stage of one's reasoning, the field K is fixed and the map $f \colon R \to K$ is applied implicitly, just viewing every $x : R$ as $x : K$. It is now impossible to view $f(R) \leq K$ as an inclusion of subalgebras, because the map f is needed explicitly to give the R-algebra structure on K. As a solution, we use a type synonym `f.codomain := K` and instantiate the R-algebra structure given by f on this synonym.

3.6 Representing monogenic field extensions

In Section 2 we have informally said that every number field K can be written as $K = \mathbb{Q}(\alpha)$ for a root α of an irreducible polynomial $P \in \mathbb{Q}[X]$. This can be made precise in several ways. For instance, one can consider a large field L (of characteristic 0) where P splits completely, then choose a root $\alpha \in L$ and let $K = \mathbb{Q}(\alpha)$ be the smallest subfield of L containing α. Or, one can consider the quotient ring $\mathbb{Q}[X]/P$ and observe that this is a field where the class $X \pmod{P}$ is a root of P. The assignment $\alpha \mapsto X \pmod{P}$ yields an isomorphism of the two fields, but any other choice of a root $\alpha' \in L$ leads to another isomorphism $\mathbb{Q}(\alpha') \cong \mathbb{Q}[X]/P$. Although mathematically we often tacitly identify the constructions, there is no canonical representation of the *monogenic* extensions of $\mathbb{Q}$, those which can be obtained by adjoining a single root of one polynomial.

The same continues to hold if we replace the base field $\mathbb{Q}$ with another field F, thus considering extensions of the form $F(\alpha)$, now requiring that α be a root of some $P \in F[X]$. Various constructions of $F(\alpha)$ have already been formalized in mathlib. The ability to switch between these representations is important: sometimes K and F are fixed and we want an arbitrary α; sometimes α is fixed and we want an arbitrary type representing $F(\alpha)$.

To find a uniform way to reason about all these definitions, we chose to formalize the notion of *power basis* to represent monogenic field extensions: this is a basis of the form $1, x, x^2, \dots, x^{n-1} : K$ (viewing K as a F-vector space). We defined a structure type bundling the information of a power basis. Omitting some generalizations not needed in this paper, the definition reads:

```
structure power_basis (F K : Type*) [field F] [field K] [algebra F K] :=
(gen : S) (dim : ℕ)
(is_basis : is_basis F (λ (i : fin dim), gen ^ (i : ℕ)))
```

We formalized that the previously defined notions of monogenic field extensions are equivalent to the existence of a power basis.

With the `power_basis` structure, we gained the ability to parametrize our results, being able to choose the F and K in a monogenic field extension K/F, or being able to choose the α generating $F(\alpha)$ (by setting `power_basis.gen pb` equal to α). To specialize a result from an arbitrary K with a power basis over F to a specific construction of $K = F(\alpha)$, one can apply the result to the power basis generated by α and rewrite `power_basis.gen F(`α`)` $= \alpha$.

4 Dedekind domains

The right setting to study algebraic properties of number fields are *Dedekind domains*. We formalized fundamental results on Dedekind domains, including the equivalence of two definitions of Dedekind domain.

4.1 Definitions

There are various equivalent conditions, used at various times, for an integral domain D to be a Dedekind domain. The following three have been formalized in mathlib:

- `is_dedekind_domain D`: D is a Noetherian integral domain, integrally closed in its fraction field and has Krull dimension at most 1;
- `is_dedekind_domain_inv D`: D is an integral domain and nonzero fractional ideals of D have a multiplicative inverse (we discuss the notion and formalization of fractional ideals in Section 4.2);
- `is_dedekind_domain_dvr D`: D is a Noetherian integral domain and the localization of D at each nonzero prime ideal is a discrete valuation ring.

Note that fields are Dedekind domains according to these conventions.

The mathlib community chose `is_dedekind_domain` as the main definition, since this condition is usually the one checked in practice [22]. The other two equivalent definitions were added to mathlib, but before formalizing the proof that they are indeed equivalent. Having multiple definitions allowed us to do our work in parallel without depending on unformalized results. For example, the proof of unique ideal factorization in a Dedekind domain initially assumed `is_dedekind_domain_inv D`, and the proof that the ring of integers $\mathcal{O}_K$ is a Dedekind domain concluded `is_dedekind_domain (ring_of_integers K)`. After the equivalence between `is_dedekind_domain D` and `is_dedekind_domain_inv D` was formalized, we could easily replace usages of `is_dedekind_domain_inv` with `is_dedekind_domain`.

The conditions `is_dedekind_domain` and `is_dedekind_domain_inv` require a fraction field K, although the truth value of the predicates does not depend on the choice of K. For ease of use, we let the type of `is_dedekind_domain` depend only on the domain D by instantiating K in the definition as `fraction_ring D`. From now on, we fix a fraction map $f \colon D \to K$.

```
class is_dedekind_domain (D : Type*) [integral_domain D] : Prop :=
(to_is_noetherian_ring : is_noetherian_ring D)
(dimension_le_one : dimension_le_one D)
(is_integrally_closed : integral_closure D (fraction_ring D) = ⊥)
```

The notation $\bot$ is used in mathlib for the bottom element of a lattice. For example, here $\bot$ denotes the smallest D-subalgebra of `fraction_ring D`, i.e. D itself, and `⊥ : ideal D` denotes the zero ideal.

Applications of `is_dedekind_domain` can choose a specific fraction field through the following lemma exposing the alternate definition:

```
lemma is_dedekind_domain_iff (f : fraction_map D K) :
  is_dedekind_domain D ↔
    is_noetherian_ring D ∧ dimension_le_one D ∧
    integral_closure D f.codomain = ⊥
```

We marked `is_dedekind_domain` as a typeclass by using the keyword `class` rather than `structure`, allowing the typeclass system to automatically infer the Dedekind domain structure when an appropriate instance is declared, such as for PIDs or rings of integers.

4.2 Fractional ideals

The notion which is pivotal to the definition of the ideal class group of a Dedekind domain is that of *fractional ideals*: given any integral domain R with a field of fractions F, we define `is_fractional` as a predicate on R-submodules J of F, informally as "there is an $x : R$ with $xJ \subseteq R$". For a Dedekind domain, nonzero fractional ideals form a group under multiplication. As seen in Section 3.5, this notion depends on the field K as well as on the fraction map $f\colon R \to K$. A more precise way of stating the above condition is then $f(x)J \subseteq f(R)$. We formalized the definition of fractional ideals relative to a map $f\colon R \to K$ as a type `fractional_ideal f`, whose elements consist of the R-submodule of F along with a proof of `is_fractional`. The structure of fractional ideals does not depend on the choice of a fraction map, which we formalized as an isomorphism `fractional_ideal.canonical_equiv` between the fractional ideals relative to fraction maps $f_1\colon R \to K_1$ and $f_2\colon R \to K_2$.

We defined the addition, multiplication and intersection operations on fractional ideals, by showing that the corresponding operations on submodules map fractional ideals to fractional ideals. We also formalized that these operations give a commutative semiring structure on the type of fractional ideals. For example, multiplication of fractional ideals is defined as

```
lemma fractional_mul (I J : fractional_ideal f) :
  is_fractional f (I * J : submodule R f.codomain) := _ -- proof omitted

instance : has_mul (fractional_ideal f) :=
⟨λ I J, ⟨I * J : submodule R f.codomain, fractional_mul I J⟩⟩
```

Defining the quotient of two fractional ideals requires slightly more work. Consider any R-algebra A and an injection $R \hookrightarrow A$. Given ideals $I, J \leq R$, the submodule $I/J \leq A$ is defined by the property

```
lemma submodule.mem_div_iff_forall_mul_mem {x : A} {I J : submodule R A} :
  x ∈ I / J ↔ ∀ y ∈ J, x * y ∈ I
```

Beware that the notation $1/I$ might be misleading here: indeed, for general integral domains, the equality $I * 1/I = 1$ might not hold. As an example, one can consider the ideal (X, Y) in $\mathbb{C}[X, Y]$. On the other hand, we formalized that this equality holds for Dedekind domains (Section 4.3) as the following lemma:

```
lemma fractional_ideal.is_unit {hD : is_dedekind_domain D}
  (I : fractional_ideal f) (hne : I ≠ ⊥) : is_unit I
```

This justifies the notation $I^{-1} = 1/I$. In fact, we define this notation even for the ideal 0, by declaring that $0^{-1} = 0$. This reflects the existence of the typeclass `group_with_zero` in `mathlib`, consisting of groups endowed with an extra element `0` whose inverse is again `0`.

Moreover, `mathlib` used to define $a/b := a * b^{-1}$, but our definition of $I^{-1} = 1/I$ would cause circularity. This led us to a major refactor of this core definition. In particular, we had to weaken the definitional equality to a proposition; this involved many small changes throughout `mathlib`.[3]

4.3 Equivalence of the definitions

We now describe how we proved and formalized that the two definitions `is_dedekind_domain` and `is_dedekind_domain_inv` of being a Dedekind domain are equivalent. Let D be a Dedekind domain, and $f\colon D \to K$ a fraction map to a field of fractions K of D.

To show that `is_dedekind_domain_inv` implies `is_dedekind_domain`, we follow the proof given by Fröhlich in [14, Chapter 1, § 2, Proposition 1.2.1]. A constant challenge that was faced while coding this proof was already mentioned in Section 3.5, namely the fact that elements of the ring must be traced along the fraction map. The proofs for being integrally closed and of dimension being less than or equal to 1 are fairly straightforward.

Formalizing the Noetherian condition was the most challenging. Fröhlich considers elements $a_1, \ldots, a_n \in I$ and $b_1, \ldots, b_n \in I^{-1}$ for any nonempty fractional ideal I, satisfying $\sum_i a_i b_i = 1$. However, it is quite challenging to prove that an element of the product of two D-submodules A and B must be of the form $\sum_{i=1}^{m} a_i * b_i$, for $a_i \in A$ and $b_i \in B$ for all $1 \leq i \leq m$. Instead, we show that, for every element $x \in A * B$, there are finite sets $T \subseteq A$, $T' \subseteq B$ such that `x ∈ span (T * T’)`, formalized as `submodule.mem_span_mul_finite_of_mem_mul`. Now considering a nonzero integral ideal I of the ring D, by definition of invertibility we can write `1 ∈ (1 : fractional_ideal f) = I * 1 / I`. Hence, we obtain finite sets $T \subset I$ and $T' \subset 1/I$ such that 1 is contained in the D-span of $T * T'$. We used the `norm_cast` tactic [19] to resolve most coercions, however, this tactic did not solve coercions coming from the fraction map. With coercions, the actual statement of the latter expression in Lean is `↑T’ ⊆ ↑↑(1 / ↑I)`, which reads

```
(T’ : set (fraction_ring.of D).codomain) ⊆
  (((1 / (I : fractional_ideal (fraction_ring.of D)))
    : submodule D (fraction_ring.of D).codomain)
    : set (fraction_ring.of D).codomain
```

The lemma `fg_of_one_mem_span_mul` then shows that I is finitely generated, concluding the proof.

[3] The pull requests are available as `https://github.com/leanprover-community/mathlib/pull/5302` and `https://github.com/leanprover-community/mathlib/pull/5303`.

The theorem `fractional_ideal.mul_inv_cancel` proves the converse, namely that `is_dedekind_domain` implies `is_dedekind_domain_inv`. The classical proof consists of three steps: first, every maximal ideal $M \subseteq D$, seen as a fractional ideal, is invertible; secondly, every nonzero ideal is invertible, using that it is contained in a maximal ideal; thirdly, the fact that every fractional ideal J satisfies $xJ \leq I$ for a suitable $x \in D$ and an ideal $I \subseteq D$ implies that every fractional ideal is invertible, concluding the proof that nonzero fractional ideals form a group. The third step was easy, building upon the material developed for the general theory of `fractional_ideal f`. Concerning the first two, we found that passing from the case where M is maximal to the general case required more code than directly showing invertibility of arbitrary nonzero ideals. The formal statement reads

```
lemma coe_ideal_mul_one_div [hD : is_dedekind_domain D]
  (I : ideal D) (hne : I ≠ ⊥) :
  ↑I * ((1 : fractional_ideal f) / ↑I) = (1 : fractional_ideal f)
```

from where it becomes apparent that we had to repeatedly distinguish between `I : ideal D`, and its coercion `↑I : fractional_ideal f` although these objects, from a mathematical point of view, are identical.

The formal proof of this result relies on the lemma `exists_not_mem_one_of_ne_bot`, which says that for every non-trivial ideal $0 \subsetneq I \subsetneq D$, there exists an element in the field K which is not integral (so, not in $f(D)$) but lies in $1/I$. The proof begins by invoking that every nonzero ideal in the Noetherian ring D contains a product of nonzero prime ideals. This result was not previously available in mathlib. The dimension condition shows its full force when applying this lemma: each prime ideal in the product, being nonzero, will be maximal because the Krull dimension of D is at most 1; from this, `exists_not_mem_one_of_ne_bot` follows easily. Having the above lemma at our disposal, we were able to prove that every ideal $I \neq 0$ is invertible by arguing by contradiction: if $I * 1/I \lneq D$, we can find an element $x \in K \setminus f(R)$ which is in $1/(1 * 1/I)$ thanks to `exists_not_mem_one_of_ne_bot` and some easy algebraic manipulation will imply that x is actually integral over D. Since D is integrally closed, it must lie in $f(D)$, contradicting the construction of x. Combining these results gives the equivalence between the two conditions for being a Dedekind domain.

5 Principal ideal domains are Dedekind

As an example of our definitions, we discuss in some detail our formalization of the fact that a principal ideal domain is a Dedekind domain. There is no explicit definition of PIDs in mathlib, rather it is split up into two hypotheses. One uses `[integral_domain R]` `[is_principal_ideal_ring R]` to denote a PID R, where `is_principal_ideal_ring` is a typeclass defined for all commutative rings:

```
class is_principal_ideal_ring (R : Type*) [comm_ring R] : Prop :=
(principal : ∀ (I : ideal R), is_principal I)
```

Our proof that the hypotheses `[integral_domain R] [is_principal_ideal_ring R]` imply `is_dedekind_domain R` was relatively short:

```
instance principal_ideal_ring.to_dedekind_domain (R : Type*)
  [integral_domain R] [is_principal_ideal_ring R] :
  is_dedekind_domain R :=
⟨principal_ideal_ring.is_noetherian_ring,
 dimension_le_one.principal_ideal_ring _,
 unique_factorization_monoid.integrally_closed (fraction_ring.of R)⟩
```

Recall from Section 3 that the `instance` keyword marks the declaration for inference by the typeclass system.

The Noetherian property of a Dedekind domain followed easily by the previously defined lemma `principal_ideal_ring.is_noetherian_ring`, since, by definition, each ideal in a principal ideal ring is finitely generated (by a single element).

We proved the lemma `dimension_le_one.principal_ideal_ring`, which is an instantiation of the existing result `is_prime.to_maximal_ideal`, showing a nonzero prime ideal in a PID is maximal. The latter lemma uses the characterization that I is a maximal ideal if and only if any strictly larger ideal $J \supsetneq I$ is the full ring R. If I is a nonzero prime ideal and $J \supsetneq I$ in the PID R, we have that the generator j of J is a divisor of the generator i of I. Since I is prime, this implies that either $j \in I$, contradicting the assumption that $J \supsetneq I$, $i = 0$, contradicting that I is nonzero, or that j is a unit, implying $J = R$ as desired.

The final condition of a PID being integrally closed was the most challenging. We used the previously defined instance `principal_ideal_ring.to_unique_factorization_monoid` to deduce that a PID is a unique factorisation monoid (UFM), to instantiate our proof that every UFM is integrally closed. In the same way that principal ideal domains are generalized to principal ideal rings, `mathlib` generalizes unique factorization domains to unique factorization monoids. A commutative monoid R with an absorbing element 0 and injectivity of multiplication is defined to be a UFM, if the relation "x properly divides y" is well-founded (implying each element can be factored as a product of irreducibles) and an element of R is prime if and only if it is irreducible (implying the factorization is unique). The first condition is satisfied for a PID since the Noetherian property implies that the division relation is well-founded. The second condition followed from `principal_ideal_ring.irreducible_iff_prime`. To prove that an irreducible element p is prime, the proof uses that prime elements generate prime ideals and irreducible elements of a PID generate maximal ideals. Since all maximal ideals are prime ideals, the ideal generated by p is maximal, hence prime, thus p is prime. We proved the lemma `irreducible_of_prime`, which shows the converse holds in any commutative monoid with zero.

To show that a UFM is integrally closed, we first formalized the Rational Root Theorem, named `denom_dvd_of_is_root`, which states that for a polynomial $p : R[X]$ and an element of the fraction field $x : \operatorname{Frac} R$ such that $p(x) = 0$, the denominator of x divides the leading coefficient of p. If x is integral with minimal polynomial p, the leading coefficient is 1, therefore the denominator is a unit and x is an element of R. This gave us the required lemma `unique_factorization_monoid.integrally_closed`, which states that the integral closure of R in its fraction field is R itself.

6 Rings of integers are Dedekind domains

An important classical result in algebraic number theory is that the ring of integers of a number field K, defined as the integral closure of $\mathbb{Z}$ in K, is a Dedekind domain. We formalized a stronger result: given a Dedekind domain D and a field of fractions F, if K is a finite separable extension of F, then the integral closure of D in K is a Dedekind domain with fraction field K. Our approach was adapted from Neukirch [22, Theorem 3.1]. Throughout this section, let D be a Dedekind domain with a field of fractions F (given by the map $f\colon D \to F$), K a finite, separable field extension of F and let S denote the integral closure of D in K.

The first step was to show that K is a field of fractions for the integral closure, namely, there is a map `fraction_map_of_finite_extension f K : fraction_map S K`. The main content of `fraction_map_of_finite_extension` consisted of showing that all elements $x : K$ can be written as y/z for elements $y \in S$, $z \in D \subseteq S$; the standard proof of this fact (see [10, Theorem 15.29]) formalized readily.

We could then show that the integral closure of D in K is a Dedekind domain, by proving it is integrally closed in K, has Krull dimension at most 1 and is Noetherian. The fact that the integral closure is integrally closed was immediate.

To show the Krull dimension is at most 1, we needed to develop basic going-up theory for ideals. In particular, we showed that an ideal I in an integral extension is maximal if it lies over a maximal ideal, and used a result already available in mathlib that a prime ideal I in an integral extension lies over a prime ideal.

```
lemma is_maximal_of_is_integral_of_is_maximal_comap
  (I : ideal S) [is_prime I]
  (hI : is_maximal (comap f I)) : is_maximal I
theorem is_prime.comap (I : ideal S) [hI : is_prime I] :
  is_prime (comap f I)
```

The final condition, that the integral closure S of D in L is a Noetherian ring, required the most work. We started by following the first half of Dummit and Foote [10, Theorem 15.29], so that it sufficed to find a nondegenerate bilinear form B such that all integral $x, y : K$ satisfy $B(x, y) \in$ `integral_closure` D K. We then formalized the results in Neukirch [22, §§ 2.5–2.8] to show that the *trace form* is a bilinear form satisfying these requirements.

6.1 The trace form

In the notation from the previous section, consider the bilinear map `lmul := λ x y : K, x * y`. The trace of the linear map `lmul x` is called the *algebra trace* $\mathrm{Tr}_{K/F}(x)$ of x. We defined the algebra trace as a linear map, in this case from K to F:

```
noncomputable def trace : K →ₗ[F] F :=
linear_map.comp (linear_map.trace F K) (to_linear_map (lmul F K))
```

This definition was marked noncomputable since `linear_map.trace` makes a case distinction on the existence of a finite basis, choosing an arbitrary finite basis if one exists and returning 0 otherwise. This latter case did not occur in our development.

We defined the *trace form* to be an F-bilinear form on K, mapping $x, y : K$ to $\mathrm{Tr}_{K/F}(xy)$.

```
noncomputable def trace_form : bilin_form F K :=
{ bilin := λ x y, trace F K (x * y), .. /- proofs omitted -/ }
```

In the following, let $L/K/F$ be a tower of finite extensions of fields, namely we assumed `[algebra F K] [algebra K L] [algebra F L] [is_scalar_tower F K L]`, as described in Section 3.2.

The value of the trace depends on the choice of F and K; we formalized this as lemmas `trace_algebra_map x : trace F K (algebra_map F K x) = findim F K • x` as well as `trace_comp K x : trace F L x = trace F K (trace K L x)`. These results followed by direct computation.

To compute $\mathrm{Tr}_{K/F}(x)$, it therefore suffices to consider the trace of x in the smallest field containing x and F, which is the monogenic extension $F(x)$ discussed in Section 3.6. There is a nice formula for the trace in $F(x)$, although the terms in this formula are elements in a larger field L (such as the *splitting field* of the minimal polynomial of x). In formalizing this formula, we first mapped the trace to L using the canonical embedding `algebra_map F L`, which gave the following lemma statement:

```
lemma power_basis.trace_gen_eq_sum_roots (pb : power_basis F K)
  (h : polynomial.splits (algebra_map F L) pb.minpoly_gen) :
  algebra_map F L (trace F K pb.gen) =
    sum (roots (map (algebra_map F L) pb.minpoly_gen))
```

We formulated the lemma in terms of the power basis, since we needed to use it for $F(x)$ here and for an arbitrary finite separable extension L/K later in the proof.

The elements of `(pb.minpoly_gen.map (algebra_map F L)).roots` are called *conjugates* of x in L. Each conjugate of x is integral since it is a root of (the same) monic polynomial, and integer multiples and sums of integral elements are integral. Combining `trace_gen_eq_sum_roots` and `trace_algebra_map` showed that the trace of x is an integer multiple (namely `findim F(x) L`) of a sum of conjugate roots, hence we concluded that the trace (and trace form) of an integral element is also integral.

Finally, we showed that the trace form is nondegenerate, following Neukirch [22, Proposition 2.8]. Since K/F is a finite, separable field extension, it has a power basis `pb` generated by x. Letting x_k denote the k-th conjugate of x in an algebraically closed field $L/K/F$, the main difficulty was in checking the equality $\sum_k x_k^{i+j} = \mathrm{Tr}_{K/F}(x^{i+j})$. Directly applying `trace_gen_eq_sum_roots` was tempting, since we had a sum over conjugates of powers on both sides. However, the two expressions did not precisely match: the left hand side is a sum of conjugates of x, where each conjugate is raised to the power $i + j$, while the conclusion of `trace_gen_eq_sum_roots` resulted in a sum over conjugates of x^{i+j}.

Instead, the paper proof switched here to an equivalent definition of conjugate: the conjugates of x in L are the images (counted with multiplicity) of x under each embedding $\sigma\colon F(x) \to L$ that fixes F. This equivalence between the two notions of conjugate was contributed to `mathlib` by the Berkeley group in the week before we realized we needed it. Mapping `trace_gen_eq_sum_roots` through the equivalence gave $\mathrm{Tr}_{K/F}(x) = \sum_\sigma \sigma x$. Since each σ is a ring homomorphism, $\sigma\ x^{i+j} = (\sigma\ x)^{i+j}$, so the conjugates of x^{i+j} are the $(i + j)$-th powers of conjugates of x, which concluded the proof.

7 Class group and class number

Given a Dedekind domain with fraction map $f\colon D \to K$, we formalized the notion of class group in Lean by defining a map `to_principal_ideal f : units f.codomain → units (fractional_ideal f)`, and defined the class group as

```
def class_group := quotient_group.quotient (to_principal_ideal (range f))
```

In general, Dedekind domains can have infinite class groups. However, as discussed in Section 2, the rings of integers of global fields have finite class groups.

We let K be a number field and K' be a function field, with ring of integers $\mathcal{O}_K$ and $\mathcal{O}_{K'}$ (w.r.t. a fixed $\mathbb{F}_q[t]$), respectively. Most proofs of the finiteness of $\mathcal{Cl}_{\mathcal{O}_K}$ one finds in a modern textbook (see [22, Theorems 4.4, 5.3, 6.3]) depend on Minkowski's lattice point theorem, a result from the geometry of numbers (which has been formalized in Isabelle/HOL [11]). Extending this proof to show the finiteness of $\mathcal{Cl}_{\mathcal{O}_{K'}}$ is quite involved and does not result in a uniform proof for $\mathcal{Cl}_{\mathcal{O}_K}$ and $\mathcal{Cl}_{\mathcal{O}_{K'}}$. Our formalization instead adapted and generalized a classical approach to the finiteness of $\mathcal{Cl}_{\mathcal{O}_K}$, where the use of Minkowski's theorem is replaced by the pigeonhole principle. For an informal writeup of the proof, used in the formalization efforts, see `https://github.com/lean-forward/class-number/blob/itp-2021-final/FiniteClassGroup.pdf`. The classical approach seems to go back to Kronecker and can be found, for instance, in [17]. We note that some other "uniform" approaches can be found in [1] and [24].

Let D be an Euclidean domain: in particular, it will be a PID and hence a Dedekind domain. Given a fraction map $f\colon D \to F$, let K be a finite separable field extension of F. We formalized, in the theorem `class_group.finite_of_admissible`, that the integral closure of D in K has a finite class group if D has an "admissible" absolute value `abs`. This notion originated in our project from the adaptation and generalization of the classical finiteness proof in interaction with the formalization efforts. Very informally, the admissibility conditions require that the remainder operator `%` produces values that are not too far apart. Formally, we defined the type of admissible absolute values on D as follows, where `to_fun` is local notation for an application of the absolute value operator:

```
structure admissible_absolute_value (D : Type*) [euclidean_domain D]
  extends euclidean_absolute_value D ℤ :=
(card : ℝ → ℕ) (exists_partition :
  ∀ (n : ℕ) (ε > (0 : ℝ)) (b ≠ (0 : D)) (A : fin n → D),
  ∃ (t : fin n → fin (card ε)), ∀ i₀ i₁, t i₀ = t i₁ →
  (to_fun (A i₁ % b - A i₀ % b) : ℝ) < to_fun b · ε)
```

The above condition formalizes and generalizes an intermediate result in paper finiteness proofs; the different proofs for number fields and function fields (still assuming K/F separable) become the same after this point. We used division with remainder to replace the *fractional part* operator on F in the classical proof, which was essential to incorporate function fields, and at the same time allowed our proof to stay entirely within D to avoid coercions.

The absolute value extends to a norm `abs_norm f abs : integral_closure D K → ℤ`. We used the admissibility of `abs` to find a finite set `finset_approx L f abs` of elements of D, such that the following generalization of [17, Theorem 12.2.1] holds.

```
theorem exists_mem_finset_approx' (a b : integral_closure D L) :=
  ∃ (q : integral_closure D L) (r ∈ finset_approx L f abs),
  abs_norm f abs (r · a - q * b) < abs_norm f abs b
```

After this, the classical approach mentioned above formalized smoothly.

It remained to define an admissible absolute value for $\mathbb{Z}$ and $\mathbb{F}_q[t]$. On $\mathbb{Z}$, it was straightforward to formalize that the usual Archimedean absolute value fulfils the requirements. For $\mathbb{F}_q[t]$, we showed that $|f|_{\deg} := q^{\deg f}$ for $f \in \mathbb{F}_q[t]$ is the required admissible absolute value; we note that this was somewhat more involved to formalize. We concluded that when K is a global field, restricting to *separable* extensions of $\mathbb{F}_q(t)$ in the function field case (but see the remark below), the class group is finite:

```
noncomputable instance : fintype
  (class_group (number_field.ring_of_integers.fraction_map K)) :=
class_group.finite_of_admissible K int.fraction_map int.admissible_abs
```

```
noncomputable instance [is_separable f.codomain K] : fintype
  (class_group (function_field.ring_of_integers.fraction_map f K)) :=
class_group.finite_of_admissible F f polynomial.admissible_card_pow_degree
```

Finally, we defined `number_field.class_number` and `function_field.class_number` as the cardinality of the respective class groups.

We remark that it is possible to get rid of the `[is_separable f.codomain K]` assumption above. For instance, using that any function field K, given as finite extension of $\mathbb{F}_q(t)$, contains an $s \in K$ such that $K/\mathbb{F}_q(s)$ is a finite *and separable* extension; see for example [18, Corollary 4.4 in Chapter VIII] (noting that $\mathbb{F}_q$ is perfect and K has transcendence degree 1 over $\mathbb{F}_q$). One then also needs to show that finiteness of the class group of the integral closure of $\mathbb{F}_q[s]$ in K is preserved upon replacing $\mathbb{F}_q[s]$ by $\mathbb{F}_q[t]$. A trivial way to get rid of the assumption in the statement above is to simply move it to our definition of function field. While this would be mathematically consistent by the result just cited, we did not opt to do this (for instance showing a finite extension of a function field is a function field would become nontrivial).

We rounded off our development by determining the class number in the simplest possible case: the rational numbers $\mathbb{Q}$. First, we formalized the theorem `class_number_eq_one_iff`, stating that the class number of K is 1 if and only if $\mathcal{O}_K$ is a principal ideal domain. After defining the isomorphism `rat.ring_of_integers_equiv` showing $\mathcal{O}_\mathbb{Q}$ is $\mathbb{Z}$, we could use the fact that $\mathbb{Z}$ is a PID to conclude that the class number of $\mathbb{Q}$ is equal to 1:

```
theorem rat.class_number : number_field.class_number ℚ = 1 :=
class_number_eq_one_iff.mpr (is_principal_ideal_ring.of_surjective _
  rat.ring_of_integers_equiv.symm.surjective)
```

8 Discussion

8.1 Related work

Broadly speaking, one could see the formalization work as part of number theory. There are several formalization results in this direction. Most notably, Eberl formalized a substantial part of analytic number theory in Isabelle/HOL [12]. Narrowing somewhat to a more algebraic setting, Cano, Cohen, Dénès, Mörtberg and Siles formalized constructive definitions in ring theory, most notable for our discussion being the Krull dimension [5]. We are not aware of any other formal developments of fractional ideals, Dedekind domains or class groups of global fields.

There are many libraries formalizing basic notions of commutative algebra such as field extensions and ideals, including the Mathematical Components library in Coq [20], the algebraic library for Isabelle/HOL [3], the `set.mm` database for MetaMath [21] and the Mizar Mathematical Library [16]. The field of algebraic numbers, or more generally algebraic closures of arbitrary fields, are also available in many provers. For example, Blot [4] formalized algebraic numbers in Coq, Cohen [8] constructed the subfield of real algebraic numbers in Coq, Thiemann, Yamada and Joosten [27] formalized algebraic numbers in Isabelle/HOL, Carneiro [6] in MetaMath, and Watase [29] in Mizar. To our knowledge, the Coq Mathematical Components library is the only formal development beside ours specifically dealing with number fields [20, `field/algnum.v`].

Apart from the general theory of algebraic numbers, there are formalizations of specific rings of integers. For instance, the Gaussian integers $\mathbb{Z}[i]$ have been formalized in Isabelle/HOL by Eberl [13], in MetaMath by Carneiro [7] and in Mizar by Futa, Mizushima, and Okazaki [15]. Eberl's Isabelle/HOL formalization deserves special mention in this context since it introduces techniques from algebraic number theory, defining the integer-valued norm on $\mathbb{Z}[i]$ and classifying the prime elements of $\mathbb{Z}[i]$.

8.2 Future directions

Having formalized various basic results of algebraic number theory, there are several natural directions for future work, including formalizing some of the following results.

- The group of units of the ring of integers in a number field is finitely generated, or slightly stronger, Dirichlet's unit theorem [22, Theorem 7.4] (and the function field analogue).
- Other finiteness results in algebraic number theory, most notably Hermite's theorem about the existence of finitely many number fields, up to isomorphism, with bounded discriminant [22, Theorem 2.16] (and the function field analogue).
- Class number computations, say of quadratic number fields. This could be part of verifying correctness of number theoretic software, such as KASH/KANT [23] and PARI/GP [26].
- Applications of algebraic number theory to solving Diophantine equations, such as determining all pairs of integers (x, y) such that $y^2 = x^3 + D$ for given nonzero $D \in \mathbb{Z}$.

8.3 Conclusion

In this project, we confirmed the rule that the hardest part of formalization is to get the definitions right. Once this is accomplished, the paper proof (sometimes first adapted with formalization in mind) almost always translates into a formal proof without too much effort. In particular, we regularly had to invent abstractions to treat instances of the "same" situation uniformly. Instead of fixing a canonical representation, be it $F \subseteq K \subseteq L$ as subfields or the field of fractions $\operatorname{Frac} R$, or the monogenic $K(\alpha)$, we found that making the essence of the situation an explicit parameter, as in `is_scalar_tower`, `fraction_map` or `power_basis`, allows to treat equivalent viewpoints uniformly without the need for transferring results.

The formalization efforts described in this paper cannot be cleanly separated from the development of mathlib as a whole. The decentralized organization and highly integrated design of mathlib meant that we could contribute our formalizations as we completed them, resulting in a quick integration into the rest of the library. Other contributors building on these results often extended them to meet our requirements, before we could identify that we needed them, as the anecdote in Section 3.4 illustrates. In other words, the low barriers for contributions ensured mutually beneficial collaboration.

Quantifying the ratio between the length of our formal proofs and their paper counterparts in an accurrate and meanifngful way will be very difficult as background assumptions and levels of detail varied significantly. We actually did not always literally follow some written text, but deviated from the paper mathematics (often discussed orally, on blackboards, through Zulip, etc.) on many occasions. An important aspect we had to take into account was to consistently combine different descriptions of mathematical objects from different sources. The formalization project described in this paper resulted in the contribution of thousands of lines of Lean code involving hundreds of declarations. A rough estimate concerning the former would be that about five thousand lines of project specific code were added, and about half of that number of lines of more generic background code. We validated existing design choices used in mathlib, refactored those that did not scale well and contributed our own set of designs. The real achievement was not to complete each proof, but to build a better foundation for formal mathematics.

References

1 E. Artin and G. Whaples. Axiomatic characterization of fields by the product formula for valuations. *Bull. Amer. Math. Soc.*, 51(7):469–492, July 1945. URL: `https://projecteuclid.org:443/euclid.bams/1183507128`.

2 Jeremy Avigad, Leonardo de Moura, and Soonho Kong. *Theorem Proving in Lean.* Carnegie Mellon University, 2021. Release 3.23.0, `https://leanprover.github.io/theorem_proving_in_lean/`.

3 C. Ballarin (editor), J. Aransay, M. Baillon, P. E. de Vilhena, S. Hohe, F. Kammüller, and L. C. Paulson. The Isabelle/HOL algebra library. URL: `http://isabelle.in.tum.de/dist/library/HOL/HOL-Algebra/index.html`.

4 Valentin Blot. Basics for algebraic numbers and a proof of Liouville's theorem in C-CoRN, 2009. MSc internship report.

5 Guillaume Cano, Cyril Cohen, Maxime Dénès, Anders Mörtberg, and Vincent Siles. Formalized linear algebra over elementary divisor rings in Coq. *Logical Methods in Computer Science*, 12(2), 2016. `doi:10.2168/LMCS-12(2:7)2016`.

6 M. Carneiro. Definition `df-aa`. URL: `http://us.metamath.org/mpeuni/df-aa.html`.

7 M. Carneiro. Definition `df-gz`. URL: `http://us.metamath.org/mpeuni/df-gz.html`.

8 C. Cohen. Construction of real algebraic numbers in Coq. In L. Beringer and A. P. Felty, editors, *ITP 2012*, volume 7406 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2012. `doi:10.1007/978-3-642-32347-8_6`.

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 D. S. Dummit and R. M. Foote. *Abstract algebra.* John Wiley & Sons, Inc., Hoboken, NJ, third edition, 2004.

11 M. Eberl. Minkowski's theorem. *Archive of Formal Proofs*, 2017. , Formal proof development. URL: `https://isa-afp.org/entries/Minkowskis_Theorem.html`.

12 M. Eberl. Nine chapters of analytic number theory in Isabelle/HOL. In J. Harrison, J. O'Leary, and A. Tolmach, editors, *ITP 2019*, volume 141 of *LIPIcs*, pages 16:1–16:19. Schloss Dagstuhl, Leibniz-Zentrum fuer Informatik, 2019. `doi:10.4230/LIPIcs.ITP.2019.16`.

13 M. Eberl. Gaussian integers. *Archive of Formal Proofs*, 2020. , Formal proof development. URL: `https://isa-afp.org/entries/Gaussian_Integers.html`.

14 A. Fröhlich. Local fields. In *Algebraic Number Theory (Proc. Instructional Conf., Brighton, 1965)*, pages 1–41. Thompson, Washington, D.C., 1967.

15 Y. Futa, D. Mizushima, and H. Okazaki. Formalization of Gaussian integers, Gaussian rational numbers, and their algebraic structures with Mizar. In *2012 International Symposium on Information Theory and its Applications*, pages 591–595, 2012.

16 A. Grabowski, A. Kornilowicz, and C. Schwarzweller. On algebraic hierarchies in mathematical repository of Mizar. In M. Ganzha, L. Maciaszek, and M. Paprzycki, editors, *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems*, volume 8 of *ACSIS*, pages 363–371, 2016.

17 K. Ireland and M. Roosen. *A Classical Introduction to Modern Number Theory.* Springer-Verlag New York, second edition, 1990.

18 Serge Lang. *Algebra*, volume 211 of *Graduate Texts in Mathematics.* Springer-Verlag, New York, third edition, 2002. `doi:10.1007/978-1-4613-0041-0`.

19 Robert Y. Lewis and Paul-Nicolas Madelaine. Simplifying casts and coercions (extended abstract). In Pascal Fontaine, Konstantin Korovin, Ilias S. Kotsireas, Philipp Rümmer, and Sophie Tourret, editors, *Practical Aspects of Automated Reasoning*, volume 2752 of *CEUR Workshop Proceedings*, pages 53–62. CEUR-WS.org, 2020. URL: `http://ceur-ws.org/Vol-2752/paper4.pdf`.

20 A. Mahboubi and E. Tassi. *The Mathematical Components Libraries*. `https://math-comp.github.io/mcb/`, 2017.

21 N. D. Megill and D. A. Wheeler. *Metamath: A Computer Language for Mathematical Proofs*. Lulu Press, Morrisville, North Carolina, 2019. `http://us.metamath.org/downloads/metamath.pdf`.

22 J. Neukirch. *Algebraic number theory*, volume 322 of *Fundamental Principles of Mathematical Sciences*. Springer-Verlag, Berlin, 1999. Translated from the 1992 German original and with a note by Norbert Schappacher, With a foreword by G. Harder. `doi:10.1007/978-3-662-03983-0`.

23 M. E. et al Pohst. The computer algebra system KASH/KANT. `http://www.math.tu-berlin.de/~kant`.

24 A. Stasinski. A uniform proof of the finiteness of the class group of a global field. *to appear in Amer. Math. Monthly*, 2020. URL: `https://arxiv.org/abs/1909.07121`.

25 The mathlib Community. The Lean mathematical library. In J. Blanchette and C. Hrițcu, editors, *CPP 2020*, page 367–381. ACM, 2020. `doi:10.1145/3372885.3373824`.

26 The PARI Group, Univ. Bordeaux. *PARI/GP version 2.11.2*, 2019. available from `http://pari.math.u-bordeaux.fr/`.

27 R. Thiemann, A. Yamada, and S. Joosten. Algebraic numbers in Isabelle/HOL. *Archive of Formal Proofs*, 2015. , Formal proof development. URL: `https://isa-afp.org/entries/Algebraic_Numbers.html`.

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

29 Y. Watase. Algebraic numbers. *Formalized Mathematics*, 24(**4**):291–299, 2016. `doi:10.1515/forma-2016-0025`.

A Formally Verified Checker for First-Order Proofs

Seulkee Baek
Department of Philosophy, Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

The Verified TESC Verifier (VTV) is a formally verified checker for the new Theory-Extensible Sequent Calculus (TESC) proof format for first-order ATPs. VTV accepts a TPTP problem and a TESC proof as input, and uses the latter to verify the unsatisfiability of the former. VTV is written in Agda, and the soundness of its proof-checking kernel is verified in respect to a first-order semantics formalized in Agda. VTV shows robust performance in a comprehensive test using all eligible problems from the TPTP problem library, successfully verifying all but the largest 5 of 12296 proofs, with >97% of the proofs verified in less than 1 second.

2012 ACM Subject Classification Computing methodologies → Theorem proving algorithms

Keywords and phrases TESC, TPTP, TSTP, ATP

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.6

Supplementary Material *Software (Source Code)*: `https://github.com/skbaek/tesc/tree/itp`
archived at `swh:1:dir:e587c9dc69d404233c43ccdd1e411f1f734d0afd`

Funding This work has been partially supported by AFOSR grant FA9550-18-1-0120.

Acknowledgements I'd like to thank Jeremy Avigad and the anonymous referees for their careful proofreading of drafts and advice for improvements.

1 Introduction

Modern automated reasoning tools are highly complex pieces of software, and it is generally no simple matter to establish their correctness. Bugs have been discovered in them in the past [18, 11], and more are presumably hidden in systems used today. One popular strategy for coping with the possibility of errors in automated reasoning is the *De Bruijn* criterion [2], which states that automated reasoning software should produce "proof objects" which can be independently verified by simple checkers that users can easily understand and trust. In addition to reducing the trust base of theorem provers, the De Bruijn criterion comes with the additional benefit that the small trusted core is often simple enough to be formally verified itself. Such thoroughgoing verification is far from universal, but there has been notable progress toward this goal in various subfields of automated reasoning, including interactive theorem provers, SAT solvers, and SMT solvers.

One area in which similar developments have been conspicuously absent is first-order automated theorem provers (ATPs), where the lack of a machine-checkable proof format [17] precluded the use of simple independent verifiers. The Theory-Extensible Sequent Calculus (TESC) [1] is a new proof format for first-order ATPs designed to fill this gap. In particular, the format's small set of fined-grained inference rules makes it relatively easy to implement and verify its proof checker.

This paper presents the Verified TESC Verifier (VTV), a proof checker for the TESC format written and verified in Agda [3]. The aim of the paper is twofold. Its immediate

[1] TESC is a new proof format developed by the author, and there is a separate paper awaiting submission to other venues that provides details of its specification and comprehensive test results using the TPTP problem library. For more details, please refer to `https://github.com/skbaek/tesc/tree/itp` for a draft of the format paper, as well as the source code of the TESC toolchain.

purpose is to demonstrate the reliability of TESC proofs by showing precisely what is established by their successful verification using VTV. Many of the implementation issues we shall discuss, however, are relevant to any formalization of first-order logic and not limited to either VTV or Agda. Therefore, the techniques VTV uses to solve them may be useful for other projects that reason about and perform computations with first-order logic data structures.

Speaking of applicability to other systems, one may also ask: why Agda? This choice was motivated more by subjective preference and convenience (especially the ease of code extraction) than any meticulous comparison of the strengths of its alternatives. No serious effort was made to replace it with other systems (e.g. Coq or Isabelle/HOL), but it is entirely possible that they would have been just as suitable.

The rest of the paper is organized as follows: Section 2 gives a brief survey of similar works and how VTV relates to them. Section 3 describes the syntax and inference rules of the TESC proof calculus. Section 4 presents the main TESC verifier kernel, and Section 5 gives a detailed specification of the verifier's soundness property. Sections 3, 4, and 5 also include code excerpts and discuss how their respective contents are formalized in Agda. Section 6 shows the results of empirical tests measuring VTV's performance. Section 7 gives a summary and touches on potential future work.

2 Related Works

SAT solving is arguably the most developed subfield of automated reasoning in terms of verified proof checkers. A non-exhaustive list of SAT proof formats with verified checkers include RAT [13], RUP and IORUP [14], LRAT [6], and GRIT [7]. In the related field of SMT solving, the SMTCoq project [1] also uses a proof checker implemented and verified in the Coq proof assistant.

Despite the limitations imposed by Gödel's second incompleteness theorem [10], there has been interesting work toward verification of interactive theorem provers. All of HOL Light except the axiom of infinity has been proven consistent in HOL Light itself [11], which was further extended later to include definitional principles for new types and constants [15]. There are also recent projects that go beyond the operational semantics of programming languages and verifies interactive theorem provers closer to the hardware, such as the Milawa theorem prover which runs on a Lisp runtime verified against x86 machine code [8], and the Metamath Zero [4] theorem prover which targets x84-64 instruction set architecture.

VTV is designed to serve a role similar to these verified checkers for first-order ATPs and the TESC format. There has been several different approaches [22, 5] to verifying the output of first-order ATPs, but the only example the author is aware of which uses independent proof objects with a verified checker is the Ivy system [16]. The main difference between Ivy and VTV is that Ivy's proof objects only record resolution and paramodulation steps, so all input problems have to be normalized by a separate preprocessor written in ACL2. In contrast, the TESC format supports preprocessing steps like Skolemization and CNF normalization, and allows ATPs to work directly on input problems with their optimized preprocessing strategies.

3 Proof Calculus

The syntax of the TESC calculus is as follows:

$$
\begin{aligned}
f &::= \sigma \mid \#_k \\
t &::= x_k \mid f(\vec{t}) \\
\vec{t} &::= \cdot \mid \vec{t}, t \\
\phi &::= \top \mid \bot \mid f(\vec{t}) \mid \neg\phi \mid \phi \lor \chi \mid \phi \land \chi \mid \phi \to \chi \mid \phi \leftrightarrow \chi \mid \forall\phi \mid \exists\phi \\
\Gamma &::= \cdot \mid \Gamma, \phi
\end{aligned}
$$

We let f range over *functors*, which are usually called "non-logical symbols" in other presentations of first-order logic. The TESC calculus makes no distinction between function and relation symbols, and relies on the context to determine whether a symbol applied to arguments is a term or an atomic formula. For brevity, we borrow the umbrella term "functor" from the TPTP syntax and use it to refer to any non-logical symbol.

There are two types of functors: σ ranges over *plain functors*, which you can think of as the usual relation or function symbols. We assume that there is a suitable set of symbols Σ, and let $\sigma \in \Sigma$. Symbols of the form $\#_k$ are *indexed functors*, and the number k is called the *functor index* of $\#_k$. Indexed functors are used to reduce the cost of introducing fresh functors: if you keep track of the largest functor index k that occurs in the environment, you may safely use $\#_{k+1}$ as a fresh functor without costly searches over a large number of terms and formulas. [2] This is similar to the indexing tricks used in other systems (e.g. HOL theorem provers) which use the maximum variable indices of theorems to make sure that two theorems have disjoint sets of variables.

We use t to range over terms, $\vec{t}$ over lists of terms, ϕ over formulas, and Γ over sequents. Quantified formulas are written without variables thanks to the use of De Bruijn indices [9]; the number k in variable x_k is its De Bruijn index. As usual, parentheses may be inserted for scope disambiguation, and the empty list operator "$\cdot$" may be omitted when it appears as part of a complex expression. E.g., the sequent $\cdot, \phi, \psi$ and term $f(\cdot)$ may be abbreviated to ϕ, ψ and f.

Formalization of TESC syntax in Agda is mostly straightforward, but with one small caveat. Consider the following definition of the type of terms:

```
data Term : Set where
  var : Nat → Term
  fun : Functor → List Term → Term
```

The constructor fun builds a complex term out of a functor and a list of arguments. Since these arguments are behind a List, they are not immediate subterms of the complex term, and Agda cannot automatically prove termination for recursive calls that use them. For instance, term-vars<? : Nat → Term → Bool is a function such that term-vars<? k t evaluates to true iff all variables in t have indices smaller than k. It would be natural to define this function as

```
term-vars<? : Nat → Term → Bool
term-vars<? k (var m) = m <ᵇ k
term-vars<? k (fun _ ts) = all (term-vars<? k) ts
```

[2] Thanks to Marijn Heule for suggesting this idea.

Table 1 TESC inference rules. size(Γ) is the number of formulas in Γ, and $\Gamma[i]$ denotes the (0-based) ith formula of sequent Γ, where $\Gamma[i] = \top$ if the index i is out-of-bounds. $\mathrm{lfi}(x)$ is the largest functor index (lfi) occurring in x. If x incudes no functor indices, $\mathrm{lfi}(x) = -1$. $\mathrm{adm}(k, \phi)$ asserts that ϕ is an admissable formula in respect to the target theory and sequent size k.

Rule	Conditions	Example
$\dfrac{\Gamma, A(b, \Gamma[i])}{\Gamma}\,A$		$\dfrac{\phi \leftrightarrow \psi, \phi \to \psi}{\phi \leftrightarrow \psi}\,A$
$\dfrac{\Gamma, B(0, \Gamma[i]) \qquad \Gamma, B(1, \Gamma[i])}{\Gamma}\,B$		$\dfrac{\phi \vee \psi, \phi \qquad \phi \vee \psi, \psi}{\phi \vee \psi}\,B$
$\dfrac{\Gamma, C(t, \Gamma[i])}{\Gamma}\,C$	$\mathrm{lfi}(t) \leq \mathrm{size}(\Gamma)$	$\dfrac{\neg\exists f(x_0), \neg f(g)}{\neg\exists f(x_0)}\,C$
$\dfrac{\Gamma,\ D(\mathrm{size}(\Gamma), \Gamma[i])}{\Gamma}\,D$		$\dfrac{\exists f(x_0),\ f(\#_1)}{\exists f(x_0)}\,D$
$\dfrac{\Gamma, N(\Gamma[i])}{\Gamma}\,N$		$\dfrac{\neg\neg\phi, \phi}{\neg\neg\phi}\,N$
$\dfrac{\Gamma, \neg\phi \qquad \Gamma, \phi}{\Gamma}\,S$	$\mathrm{lfi}(\phi) \leq \mathrm{size}(\Gamma)$	$\dfrac{\neg f(\#_0) \qquad f(\#_0)}{\cdot}\,S$
$\dfrac{\Gamma, \phi}{\Gamma}\,T$	$\mathrm{lfi}(\phi) \leq \mathrm{size}(\Gamma)$, $\mathrm{adm}(\mathrm{size}(\Gamma), \phi)$	$\dfrac{= (f, f)}{\cdot}\,T$
$\dfrac{}{\Gamma}\,X$	For some i and j, $\Gamma[i] = \neg\Gamma[j]$	$\dfrac{}{\neg\phi, \phi}\,X$

where $m <^b k$ evaluates to true iff m is less than k. But Agda rejects this definition because it cannot prove that the recursive calls terminate. We can get around this problem by a mutual recursion between a pair of functions, one for terms and one for lists of terms:

```
term-vars<? : Nat → Term → Bool
terms-vars<? : Nat → List Term → Bool
term-vars<? k (var m) = m <^b k
term-vars<? k (fun _ ts) = terms-vars<? k ts
terms-vars<? _ [] = true
terms-vars<? k (t :: ts) = term-vars<? k t ∧ terms-vars<? k ts
```

All other functions that recurse on terms are also defined using similar mutual recursion.

The inference rules of the TESC calculus are shown in Table 1. The TESC calculus is a one-sided first-order sequent calculus, so having a valid TESC proof of a sequent Γ shows that Γ is collectively unsatisfiable. The A,B,C,D, and N rules are the *analytic* rules. Analytic rules are similar to the usual one-sided sequent calculus rules, except that each analytic rule is overloaded to handle several connectives at once. For example, consider the formulas $\phi \wedge \psi$, $\neg(\phi \vee \psi)$, $\neg(\phi \to \psi)$, and $\phi \leftrightarrow \psi$. In usual sequent calculi, you would need a different rule for each of the connectives $\wedge$, $\vee$, $\to$, and $\leftrightarrow$ to break down these formulas. But all four formulas are "essentially conjunctive" in the sense that the latter three are equivalent to $\neg\phi \wedge \neg\psi$, $\phi \wedge \neg\psi$, and $(\phi \to \psi) \wedge (\psi \to \phi)$. So it is more convenient to handle all four of them with a single rule that analyzes a formula into its left and right conjuncts, which is the analytic A rule. Similarly, the B, C, D rules are used to analyze essentially disjunctive, universal, existential formulas, and the N rule performs double-negation elimination. For a complete list of formula analysis functions that show how each analytic rule breaks down formulas, see Appendix A. The analytic rules are a slightly modified adaptation of Smullyan's *uniform notation* for analytic tableaux [21], which is where they get their names from.

All rules are designed to preserve the invariant $\text{lfi}(\Gamma) < \text{size}(\Gamma)$ for every sequent Γ. We say that a sequent Γ is *good* if it satisfies this invariant. It is important that all sequents are good, because this is what ensures that the indexed functor $\#_{\text{size}(\Gamma)}$ is fresh in respect to a sequent Γ.

Of the three remaining rules, S is the usual cut rule, and X is the axiom or init rule. The T rule may be used to add *admissable* formulas. A formula ϕ is admissable in respect to a target theory T and sequent size k if it satisfies the following condition:

- For any good sequent Γ that is satisfiable modulo T and $\text{size}(\Gamma) = k$, the sequent Γ, ϕ is also satisfiable modulo T.

More intuitively, the T rule allows you to add formulas that preserve satisfiability. Notice that the definition of admissability, and hence the definition of well-formed TESC proofs, depends on the implicit target theory. This is the "theory-extensible" part of TESC. The current version of VTV verifies basic TESC proofs that target the theory of equality, so it allows T rules to introduce equality axioms, fresh relation symbol definitions, and choice axioms. But VTV can be easily modified in a modular way to target other theories as well.

TESC proofs are formalized in Agda as an inductive type Proof, which has a separate constructor for each TESC inference rule:

```
data Proof : Set where
  rule-a : Nat → Bool → Proof → Proof
  rule-b : Nat → Proof → Proof → Proof
  rule-c : Nat → Term → Proof → Proof
  rule-d : Nat → Proof → Proof
  rule-n : Nat → Proof → Proof
  rule-s : Formula → Proof → Proof → Proof
  rule-t : Formula → Proof → Proof
  rule-x : Nat → Nat → Proof
```

For example, the constructor rule-a takes Nat and Bool arguments because this is the information necessary to specify an application of the A rule. I.e., rule-a i b p is a proof whose last inference rule adds the formula $A(b, \Gamma[i])$. Notice that sequents are completely absent from the definition of Proof. This is a design choice made in favor of efficient space usage. Since TESC proofs are uniquely determined by their root sequents together with complete information of the inference rules used, TESC files conserve space by omitting any components that can be constructed on the fly during verification, which includes all intermediate sequents and formulas introduced by analytic rules. Proof is designed to only record information included in TESC files, since terms of the type Proof are constructed by parsing input TESC files.

4 The Verifier

The checker function verify for Proof performs two tasks: (1) it constructs the omitted intermediate sequents as it recurses down a Proof, and (2) it checks that the conditions are satisfied for each inference rule used. For instance, consider the definition of verify for the C rule case, together with its type signature:

```
verify : Sequent → Proof → Bool
verify Γ (rule-c i t p) =
  term-lfi<? (suc (size Γ)) t ∧
  verify (add Γ (analyze-c t (nth i Γ))) p
```

The terms (nth i Γ), (analyze-c t (nth i Γ)), and (add Γ (analyze-c t (nth i Γ))) each corresponds to $\Gamma[i]$, $C(t, \Gamma[i])$, and $\Gamma, C(t, \Gamma[i])$. The conjunct term-lfi<? (suc (size Γ)) t ensures the side condition $\text{lfi}(t) \leq \text{size}(\Gamma)$ is satisfied, and the recursive call to verify checks that the subproof p is a valid proof of the sequent $\Gamma, C(t, \Gamma[i])$. The cases for other rules are also defined similarly.

The argument type Sequent for verify presents some interesting design choices. What kind of data structures should be used to encode sequents? The first version of VTV used lists, but lists immediately become a bottleneck with practically-sized problems due to their poor random access speeds. The default TESC verifier included in the TPTP-TSTP-TESC Processor (T3P, the main tool for generating and verifying TESC files) uses arrays, but arrays are hard to come by and even more difficult to reason about in dependently typed languages like Agda. Self-balancing trees like AVL or red-black trees come somewhere between lists and arrays in terms of convenience and performance, but it can still be tedious to prove basic facts about them if those proofs are not available in your language of choice, as is the case for Agda's standard library.

For VTV, we cut corners by taking advantage of the fact that (1) formulas are never deleted from sequents, and (2) new formulas are always added to the right end of sequents. (1) and (2) allow us to use a simple balancing algorithm. In order to use the algorithm, we first define the type of trees in a way that allows each tree to efficiently keep track of its own size:

```
data Tree (A : Set) : Set where
  empty : Tree A
  fork : Nat → A → Tree A → Tree A → Tree A
```

For any non-leaf tree fork k a t s, the number k is the size of fork k a t s. This property is not guaranteed to hold by construction, but it is easy to ensure that it always holds in practice. Since sizes of trees are stored as Nat, the function size : Tree → Nat can always report the size of trees in constant time. New elements can be added to trees in a balanced way as follows: when adding an element to a non-leaf tree, compare its subtree sizes. If the right subtree is smaller, make a recursive call and add to the right subtree. If the sizes are equal, make a new tree that contains the current tree as its left subtree, an empty tree as its right subtree, and stores the new element in the root node. The definition of the add function for trees implements this algorithm precisely:

```
add : {A : Set} → Tree A → A → Tree A
add empty a = fork 1 a empty empty
add (fork k b t s) a =
  if (size s <ᵇ size t)
  then (fork (k + 1) b t (add s a))
  else (fork (k + 1) a (fork k b t s) empty)
```

This addition algorithm does not keep the tree maximally balanced, but it provides reasonable performance and does away with complex ordering and balancing mechanisms, which makes reasoning about trees considerably easier. Given the type Tree, the type Sequent can be simply defined as Tree Formula.

5 Soundness

In order to prove the soundness of verify, we first need to formalize a first-order semantics that it can be sound in respect to. Most of the formalization is routine, but it also includes some oddities particular to VTV.

One awkward issue that recurs in formalization of first-order semantics is the handling of arities. Given that each functor has a unique arity, what do you do with ill-formed terms and atomic formulas with the wrong number of arguments? You must either tweak the syntax definition to preclude such possibilities, or deal with ill-formed terms and formulas as edge cases, both of which can lead to bloated definitions and proofs.

For VTV, we avoid this issue by assuming that every functor has infinite arities. Or rather, for each functor f with arity k, there are an infinite number of other functors that share the name f and have arities $0, 1, ..., k-1, k+1, k+2, ...$ ad infinitum. With this assumption, the denotation of functors can be defined as

```
Rels : Set
Rels = List D → Bool

Funs : Set
Funs = List D → D
```

where D is the type of the domain of discourse that the soundness proof is parametrized over. A Rels (resp. Funs) can be thought of as a collection of an infinite number of relations (resp. functions), one for each arity. An interpretation is a pair of a relation assignment and a function assignment, which assign Rels and Funs to functors.

```
RA : Set
RA = Functor → Rels

FA : Set
FA = Functor → Funs
```

Variable assignments assign denotations to Nat, since variables are identified by their Bruijn indices.

```
VA : Set
VA = Nat → D
```

For reasons we've discussed in Section 3, term valuation requires a pair of mutually recursive functions in order to recurse on terms:

```
term-val : FA → VA → Term → D
terms-val : FA → VA → List Term → List D
term-val _ V (var k) = V k
term-val F V (fun f ts) = F f (terms-val F V ts)
terms-val F V [] = []
terms-val F V (t :: ts) = (term-val F V t) :: (terms-val F V ts)
```

Formula valuation is mostly straightforward, but some care needs to be taken in the handling of variable assignments and quantified formulas. The function qtf : Bool → Formula → Formula is a constructor of Formula for quantified formulas, where qtf false and qtf true encode the universal and existential quantifiers, respectively. Given a relation assignment R, function assignment F, and variable assignment V, the values of formulas qtf false ϕ and qtf true ϕ under R, F, and V are defined as

```
R , F , V ⊨ (qtf false ϕ) = ∀ x → (R , F , (V / 0 ↦ x) ⊨ ϕ)
R , F , V ⊨ (qtf true ϕ) = ∃ λ x → (R , F , (V / 0 ↦ x) ⊨ ϕ)
```

The notations $\forall\ x \to A$ and $\exists\ \lambda\ x \to A$ may seem strange, but they are just Agda's way of writing $\forall\ x\ A$ and $\exists\ x\ A$. For any V, k, and d, $(V\ /\ k \mapsto d)$ is an updated variable assignment obtained from V by assigning d to the variable x_k, and pushing the assignments of all variables larger than x_k by one. E.g., $(V\ /\ 0 \mapsto x)\ 0 = x$ and $(V\ /\ 0 \mapsto x)\ (\mathsf{suc}\ m) = V\ m$.

Now we can define (un)satisfiability of sequents in terms of formula valuations:

```
satisfies : RA → FA → VA → Sequent → Set
satisfies R F V Γ = ∀ ϕ → ϕ ∈ Γ → R , F , V ⊨ ϕ

sat : Sequent → Set
sat Γ = ∃ λ R → ∃ λ F → ∃ λ V → (respects-eq R × satisfies R F V Γ)

unsat : Sequent → Set
unsat Γ = ¬ (sat Γ)
```

Note that × is the constructor for product types, which serves the same role as a logical conjunction here. The respects-eq R clause asserts that the relation assignment R respects equality. This condition is necessary because we are targeting first-order logic with equality, and we are only interested in interpretations that satisfy all equality axioms.

VTV's formalization of first-order semantics is atypical in that (1) every functor doubles as both relation and function symbols with infinite arities, and (2) the definition of satisfiability involves variable assignments, thereby applying to open as well as closed formulas. These idiosyncracies, however, are harmless for our purposes: whenever a traditional interpretation (with unique arities for each functor and no variable assignment) M satisfies a set of sentences Γ, M can be easily extended to an interpretation in the above sense that still satisfies Γ, since the truths of sentences in Γ are unaffected by functors or variables that do not occur in them. Conversely, any satisfying interpretation in the above sense can be pruned into a traditinal one without changing truth values of sentences. Therefore, both definitions of satisfiability agree completely on input sequents that consist of sentences. Furthermore, we may also assume that all formulas in an input sequent are sentences, because the TPTP parser will reject any open formula as it cannot assign a De Brujin index to an unbound variable. There remains, however, one small catch: the impossibility of parsing an open formula is the property of an unverified preprocessor written in Rust, so you must trust that it has been correctly implemented.

Now we finally come to the soundness statement for verify:

```
verify-sound : ∀ (S : Sequent) (p : Proof) → good S → T (verify S p) → unsat S
```

T : Bool → Set maps Boolean values true and false to the trivial and empty sets ⊤ and ⊥, respectively. The condition good S is necessary, because the soundness of TESC proofs is dependent on the invariant that all sequents are good (in the sense we defined in Section 3). But we can do better than merely assuming that the input sequent is good, because the parser which converts the input character list into the initial (i.e., root) sequent is designed to fail if the parsed sequent is not good. Therefore, we can prove this fact and use the success of the parser as confirmation that the parsed sequent is good. parse-verify is the outer function which accepts two character lists as argument, parses them into a Sequent and a Proof, and calls verify on them. The soundness statement for parse-verify is as follows:

$$
\begin{array}{l}
\mathsf{parse\text{-}verify\text{-}sound} : \forall\ (\mathit{seq\text{-}chars}\ \mathit{prf\text{-}chars} : \mathsf{List}\ \mathsf{Char}) \to \\
\quad \mathsf{T}\ (\mathsf{parse\text{-}verify}\ \mathit{seq\text{-}chars}\ \mathit{prf\text{-}chars}) \to \\
\quad \exists\ \lambda\ (S : \mathsf{Sequent}) \to \mathsf{returns}\ \mathsf{parse\text{-}sequent}\ \mathit{seq\text{-}chars}\ S \times \mathsf{unsat}\ S
\end{array}
$$

In other words, if parse-verify succeeds on two input character lists, then the sequent parser successfully parses the first character list into some unsatisfiable sequent S. parse-verify-sound is an improvement over verify-sound, but it also shows the limitation of the current setup. It only asserts that there is *some* unsatisfiable sequent parsed from the input characters, but provides no guarantees that this sequent is actually equivalent to the original TPTP problem. This means that the formal verification of VTV is limited to the soundness of its proof-checking kernel, and the correctness its TPTP parsing phase has to be taken on faith.

6 Test Results

The performance of VTV was tested by running it on all eligible problems in the TPTP [23] problem library. A TPTP problem is eligible if it satisfies all of the following conditions (parenthesized numbers indicate the total number of problems that satisfy all of the preceding conditions).

- It is in the CNF or FOF language (17053).
- Its status is "theorem" or "unsatisfiable" (13636).
- It conforms to the official TPTP syntax. More precisely, it does not have any occurrences of the character "%" in the `sq_char` syntactic class, as required by the TPTP syntax. This is important because T3P assumes that the input TPTP problem is syntactically correct and uses "%" as an endmarker (13389).
- All of its formulas have unique names. T3P requires this in order to unambiguously identify formulas by their names during proof compilation (13119).
- It can be solved by Vampire [19] or E [20] in one minute using default settings (Vampire = 7885, E = 4853 [3]).
- The TSTP solution produced by Vampire or E can be compiled to a TESC proof by T3P (Vampire = 7792, E = 4504).

The resulting 7792 + 4504 = 12296 proofs were used for testing VTV. All tests were performed on Amazon EC2 `r5a.large` instances, running Ubuntu Server 20.04 LTS on 2.5 GHz AMD EPYC 7000 processors with 16 GB RAM. [4]

Out of the 12296 proofs, there were 5 proofs that VTV failed to verify due to exhausting the 16 GB available memory. A cactus plot of verification times for the remaining 12291 proofs are shown in Fig. 1. As a reference point, we also show the plot for the default TESC

[3] It should be noted that the default setting is actually not optimal for E. When used with the `-auto` or `-auto-schedule` options, E's success rates are comparable to that of Vampire. But the TSTP solutions produced by E under these higher-performance modes are much more difficult to compile down to a TESC proof, so they could not be used for these tests.

[4] The aforementioned TESC repository (`https://github.com/skbaek/tesc/tree/itp`) also includes detailed information on the testing setup and a complete CSV of test results.

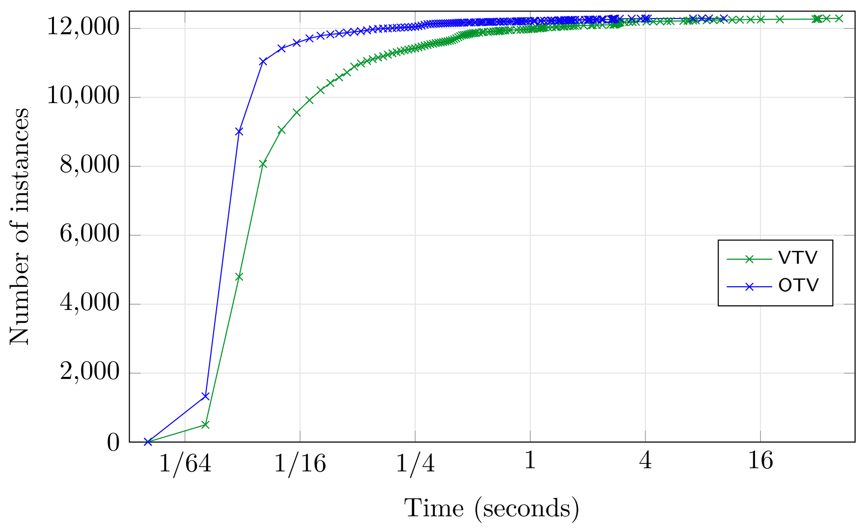


Figure 1 Verification times of VTV and OTV. The datapoints show the number of TESC proofs that each verifier could check within the given time limit. The plots look more "jumpy" toward the lower end due to the limited time measurement resolution (10 ms) of the unix `time` command.

verifier included in T3P running on the same proofs. The default TESC verifier is written in Rust, and is optimized for performance with no regard to verification. For convenience, we refer to it as the Optimized TESC Verifier (OTV).

VTV is slower than OTV as expected, but the difference is unlikely to be noticed in actual use since the absolute times for most proofs are very short, and the total times are dominated by a few outliers. VTV verified >97.4% of proofs under 1 second, and >99.3% under 5 seconds. Also, the median time for VTV is 40 ms, a mere 10 ms behind the OTV's 30 ms. But OTV's mean time (54.54 ms) is still much shorter than that of VTV (218.93 ms), so users may prefer to use it for verifying one of the hard outliers or processing a large batch of proofs at once.

The main drawback of VTV is its high memory consumption. Fig. 2 shows the peak memory usages of the two verifiers. For a large majority of proofs, memory usages of both verifiers are stable and stay within the 14-20 MB range, but VTV's memory usage spikes earlier and higher than OTV. Due to the limit of the system used, memory usage could only be measured up to 16 GB, but the actual peak for VTV would be higher if we included the 5 failed verifications. A separate test running VTV on an EC2 instance with 64 GB ram (`r5a.2xlarge`) still failed for 3 of the 5 problematic proofs, so the memory requirement for verifying all 12296 proofs with VTV is at least >64 GB. In contrast, OTV could verify all 12296 proofs with less than 3.2 GB of memory.

It's not completely clear where the memory usage difference comes from, but it is most likely caused by the different ways in which OTV and VTV keep track of subgoal sequents. OTV uses a single main array to store the current goal sequent, and destructively updates this array whenever the goal sequent is changed. Consider, for instance, the application of the S rule to a sequent Γ, which creates subgoal sequents $\Gamma, \neg\phi$ and Γ, ϕ. Before the rule application, OTV's main array holds Γ. Then it proceeds by:

1. Storing ϕ on a secondary array for later use
2. Pushing $\neg\phi$ onto the main array to change the goal sequent to $\Gamma, \neg\phi$
3. Verifying that the left subproof is a valid proof of the sequent $\Gamma, \neg\phi$

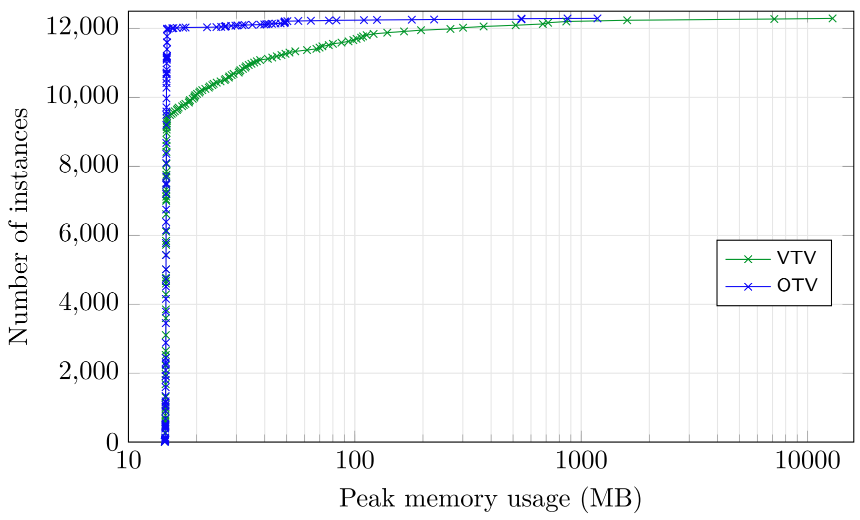


Figure 2 Peak memory usages of VTV and OTV. The datapoints show the number of TESC proofs that each verifier could check within the given peak memory usage.

4. Truncating the main array back to Γ
5. Pushing ϕ onto the main array to change the goal sequent to Γ, ϕ
6. Verifying that the right subproof is a valid proof of the sequent Γ, ϕ

This means that OTV never allocates any memory to subgoal sequents that are already closed. On the other hand, VTV is an Agda program that gets compiled to Haskell, so it is almost certain that new memory will be allocated for both $\Gamma, \neg\phi$ and Γ, ϕ for this step. Furthermore, $\Gamma, \neg\phi$ will stay in memory even after it has been closed until the garbage collector deallocates it. Therefore, it is expected that VTV will use more memory than OTV, especially when verifying proofs that branch many times and create large numbers of subgoals.

7 Conclusion

The robust test results show that VTV can serve as a fallback option when extra rigour is required in verification, thereby increasing our confidence in the correctness of TESC proofs. It can also help the design of other TESC verifiers by providing a reference implementation that is guaranteed to be correct. On a more general note, VTV is an example of a first-order logic formalization that strikes a practical balance between ease of proofs and efficient computation while avoiding some common pitfalls like non-termination and complex arity checking. Therefore, it can provide a useful starting point for other verified programming projects that use first-order logic.

There are two main ways in which VTV could be further improved. Curbing its memory usage would be the most important prerequisite for making it the default verifier in T3P. This may require porting VTV to a verified programming language with more efficient data structures (especially arrays) and finer-grained control over memory usage. The upcoming Lean 4 seems a promising candidate for this, considering its heavy emphasis on practical programming.

VTV could also benefit from a more reliable TPTP parser. A formally verified parser would be ideal, but the complexity of TPTP's syntax makes it difficult to even *specify* the correctness of a parser, let alone prove it. A more realistic approach would be imitating the technique used by verified LRAT checkers [12], making VTV print the parsed problem and textually comparing its output with the original problem. Due to TPTP's flexible syntax, however, the mapping from input files to printed outputs will necessarily be non-injective (e.g., there is no way to infer the positions of line breaks from a parsed formula). This means that the original problem will have to be normalized before comparison, and we must either verify or trust an additional TPTP normalizer.

References

1 Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of sat/smt solvers to coq through proof witnesses. In *International Conference on Certified Programs and Proofs*, pages 135–150. Springer, 2011.

2 Henk Barendregt and Freek Wiedijk. The challenge of computer mathematics. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 363(1835):2351–2375, 2005.

3 Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

4 Mario Carneiro. Metamath zero: The cartesian theorem prover. *arXiv preprint arXiv:1910.10703*, 2019.

5 Zakaria Chihani, Tomer Libal, and Giselle Reis. The proof certifier checkers. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 201–210. Springer, 2015.

6 Luís Cruz-Filipe, Marijn JH Heule, Warren A Hunt, Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified rat verification. In *International Conference on Automated Deduction*, pages 220–236. Springer, 2017.

7 Luís Cruz-Filipe, Joao Marques-Silva, and Peter Schneider-Kamp. Efficient certified resolution proof checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 118–135. Springer, 2017.

8 Jared Davis and Magnus O Myreen. The reflective milawa theorem prover is sound (down to the machine code that runs it). *Journal of Automated Reasoning*, 55(2):117–183, 2015.

9 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. North-Holland, 1972.

10 Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für mathematik und physik*, 38(1):173–198, 1931.

11 John Harrison. Towards self-verification of hol light. In *International Joint Conference on Automated Reasoning*, pages 177–191. Springer, 2006.

12 Marijn Heule, Warren Hunt, Matt Kaufmann, and Nathan Wetzler. Efficient, verified checking of propositional proofs. In *International Conference on Interactive Theorem Proving*, pages 269–284. Springer, 2017.

13 Marijn JH Heule, Warren A Hunt, and Nathan Wetzler. Verifying refutations with extended resolution. In *International Conference on Automated Deduction*, pages 345–359. Springer, 2013.

14 Marijn JH Heule, Warren A Hunt Jr, and Nathan Wetzler. Bridging the gap between easy generation and efficient verification of unsatisfiability proofs. *Software Testing, Verification and Reliability*, 24(8):593–607, 2014.

15 Ramana Kumar, Rob Arthan, Magnus O Myreen, and Scott Owens. Hol with definitions: Semantics, soundness, and a verified implementation. In *International Conference on Interactive Theorem Proving*, pages 308–324. Springer, 2014.

16 William McCune and Olga Shumsky. Ivy: A preprocessor and proof checker for first-order logic. In *Computer-Aided reasoning*, pages 265–281. Springer, 2000.

17 Giles Reger and Martin Suda. Checkable proofs for first-order theorem proving. In *ARCADE@ CADE*, pages 55–63, 2017.

18 Giles Reger, Martin Suda, and Andrei Voronkov. Testing a saturation-based theorem prover: Experiences and challenges. In *International Conference on Tests and Proofs*, pages 152–161. Springer, 2017.

19 Alexandre Riazanov and Andrei Voronkov. The design and implementation of vampire. *AI communications*, 15(2, 3):91–110, 2002.

20 Stephan Schulz. E–a brainiac theorem prover. *Ai Communications*, 15(2, 3):111–126, 2002.

21 Raymond M Smullyan. *First-order logic*. Courier Corporation, 1995.

22 Geoff Sutcliffe. Semantic derivation verification: Techniques and implementation. *International Journal on Artificial Intelligence Tools*, 15(06):1053–1070, 2006.

23 Geoff Sutcliffe. The TPTP problem library and associated infrastructure. *Journal of Automated Reasoning*, 43(4):337, 2009.

A Formula analysis functions

A	$A(0, \neg(\phi \lor \psi)) = \neg\phi$	$A(1, \neg(\phi \lor \psi)) = \neg\psi$
	$A(0, \phi \land \psi) = \phi$	$A(1, \phi \land \psi) = \psi$
	$A(0, \neg(\phi \to \psi)) = \phi$	$A(1, \neg(\phi \to \psi)) = \neg\psi$
	$A(0, \phi \leftrightarrow \psi) = \phi \to \psi$	$A(1, \phi \leftrightarrow \psi) = \psi \to \phi$
B	$B(0, \phi \lor \psi) = \phi$	$B(1, \phi \lor \psi) = \psi$
	$B(0, \neg(\phi \land \psi)) = \neg\phi$	$B(1, \neg(\phi \land \psi)) = \neg\psi$
	$B(0, \phi \to \psi) = \neg\phi$	$B(1, \phi \to \psi) = \psi$
	$B(0, \neg(\phi \leftrightarrow \psi)) = \neg\phi \to \psi$	$B(1, \neg(\phi \leftrightarrow \psi)) = \neg\psi \to \phi$
C	$C(t, \forall\phi) = \phi[0 \mapsto t]$	$C(t, \neg\exists\phi) = \neg\phi[0 \mapsto t]$
D	$D(k, \exists\phi) = \phi[0 \mapsto \#_k])$	$D(k, \neg\forall\phi) = \neg\phi[0 \mapsto \#_k]$
N	$N(\neg\neg\phi) = \phi$	

The Formula analysis functions used by analytic rules. For any argument not explicitly defined above, the functions all return $\top$. E.g., $A(0, \phi \lor \psi) = \top$. $\phi[k \mapsto t]$ denotes the result of replacing all variables in ϕ bound to the kth quantifier with term t.

Value-Oriented Legal Argumentation in Isabelle/HOL

Christoph Benzmüller
Freie Universität Berlin, Germany

David Fuenmayor
University of Luxembourg, Luxembourg
Freie Universität Berlin, Germany

Abstract

Literature in AI & Law contemplates argumentation in legal cases as an instance of theory construction. The task of a lawyer in a legal case is to construct a theory containing: (a) relevant generic facts about the world, (b) relevant legal rules such as precedents and statutes, and (c) contingent facts describing or interpreting the situation at hand. Lawyers then elaborate convincing arguments starting from these facts and rules, deriving into a positive decision in favour of their client, often employing sophisticated argumentation techniques involving such notions as burden of proof, *stare decisis*, legal balancing, etc. In this paper we exemplarily show how to harness Isabelle/HOL to model lawyer's argumentation using value-oriented legal balancing, while drawing upon shallow embeddings of combinations of expressive modal logics in HOL. We highlight the essential role of model finders (*Nitpick*) and "hammers" (*Sledgehammer*) in assisting the task of legal theory construction and share some thoughts on the practicability of extending the catalogue of ITP applications towards legal informatics.

2012 ACM Subject Classification Computing methodologies → Knowledge representation and reasoning

Keywords and phrases Higher order logic, preference logic, shallow embedding, legal reasoning

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.7

Supplementary Material *Isabelle/HOL* formalisation sources are available online at GitHub [1].
Software (Isabelle/HOL formalisation sources): `https://github.com/cbenzmueller/LogiKEy` [1]
archived at `swh:1:dir:88ab2e30ba1f4807b3dad6abaf1fc1738e809706`

Acknowledgements We thank Bertram Lomfeld for encouraging us to take on this challenge; we also thank the anonymous reviewers for their valuable feedback.

1 Introduction

In this paper we explore (value-oriented) legal reasoning as a new application area for higher-order proof assistants. More specifically, we employ *Isabelle/HOL* [33] to formalise, verify, and enhance legal arguments as presented in the context of a legal case between two parties: a *plaintiff* and a *defendant*. In the spirit of previous work in the AI & Law tradition, we tackle the formal reconstruction of legal cases as a task of theory construction, namely, "building, evaluating and using theories" [5]. Thus, *"the task for a lawyer or a judge in a "hard case" is to construct a theory of the disputed rules that produces the desired legal result, and then to persuade the relevant audience that this theory is preferable to any theories offered by an opponent"* [32].

We utilise the framework of shallow semantical embeddings (SSE; cf. [7, 15]) of (combinations of) non-classical logics in classical higher-order logic (HOL). HOL, which is instantiated here as *Isabelle/HOL*, thereby serves as a *meta-logic*, rich enough to support the encoding of combinations of *object logics* (modal, conditional, deontic, etc. [6, 8, 9, 10]) allowing for the modelling of adaptable value systems. For this sake, we also integrate some basic notions from *formal concept analysis* (FCA) [22] to exemplarily illustrate the encoding of a theory of legal values as proposed by Lomfeld [30].

This paper improves an unpublished workshop paper [11]; *Paper structure:* In §2 we outline our object logic of choice, a modal logic of preferences [37], and we then present a SSE of this logic in the *Isabelle/HOL* proof assistant. Subsequently we depict in §3 the encoding of a logic of legal values by drawing upon FCA notions and Lomfeld's value theory. In §4 we demonstrate how the formalisation of relevant legal and world knowledge can be used for formally reconstructing value-oriented arguments for an exemplary property law case. We conclude in §5 with some comments on related work and further reflections and ideas for the prospective application of ITP in the legal domain.

2 Shallow Embedding of the Object Logic

2.1 Modal Preference Logic $\mathcal{PL}$

As will become evident later on, our object logic needs to provide the means for representing (conditional) preferences between propositions. For this sake we have chosen the modal logic of *ceteris paribus* preferences as introduced by van Benthem et al. [37], which we abbreviate by $\mathcal{PL}$ in the remainder. For the purpose of this present paper we will focus our discussion on $\mathcal{PL}$'s basic preference language, disregarding the mechanism of *ceteris paribus* clauses. Nevertheless, we have provided a complete encoding and assessment of $\mathcal{PL}$ in the associated *Isabelle/HOL* sources [1]. We will briefly outline below some relevant syntactic and semantic notions of $\mathcal{PL}$ and refer the reader to [37] for a complete exposition.

$\mathcal{PL}$ is composed of normal *S4* and *K4* modal operators, together with a *global* existential modality $\mathbf{E}$. Combinations of these modalities enable us to capture a wide variety of propositional preference statements of the form $A \prec B$ (for different, indexed $\prec$-relations as shown below). The formulas of $\mathcal{PL}$ are inductively defined as follows (where $\mathtt{p}$ ranges over a set $\mathtt{Prop}$ of propositional constant symbols):

$$\varphi, \psi ::= \ \mathtt{p} \mid \varphi \wedge \psi \mid \neg\varphi \mid \Diamond^{\preceq}\varphi \mid \Diamond^{\prec}\varphi \mid \mathbf{E}\varphi$$

$\Diamond^{\preceq}\varphi$ is to be read as "φ is true in a state that is considered to be at least as good as the current state", $\Diamond^{\prec}\varphi$ as "φ is true in a state that is considered to be strictly better than the current state", and $\mathbf{E}\varphi$ as "there is a state where φ is true". $\Box^{\preceq}\varphi$, $\Box^{\prec}\varphi$ and $\mathbf{A}\varphi$ can be introduced to abbreviate $\neg\Diamond^{\preceq}\neg\varphi$, $\neg\Diamond^{\prec}\neg\varphi$ and $\neg\mathbf{E}\neg\varphi$, respectively. Further, standard logical connectives such as $\vee$, $\rightarrow$ and $\leftrightarrow$ can be defined as usual. We use **boldface** fonts to distinguish standard logical connectives of $\mathcal{PL}$ from their counterparts in HOL.

A preference model $\mathcal{M}$ is a triple $\mathcal{M} = \langle W, \preceq, V\rangle$ where: (i) W is a set of states; (ii) $\preceq$ is a so-called "betterness relation" that is reflexive and transitive (i.e. a preorder), where its strict subrelation $\prec$ is defined as: $w \prec v$ iff $w \preceq v \wedge v \not\preceq w$ for all v and w (totality of $\preceq$, i.e. $v \preceq w$ or $w \preceq v$, is generally not assumed); (iii) V is a standard modal valuation. Below we show the truth conditions for $\mathcal{PL}$'s modal connectives (the rest are standard):

$$\begin{aligned}
&\mathcal{M}, w \models \Diamond^{\preceq}\varphi \text{ iff } \exists v \in W \text{ such that } w \preceq v \text{ and } \mathcal{M}, v \models \varphi\\
&\mathcal{M}, w \models \Diamond^{\prec}\varphi \text{ iff } \exists v \in W \text{ such that } w \prec v \text{ and } \mathcal{M}, v \models \varphi\\
&\mathcal{M}, w \models \mathbf{E}\varphi \quad\ \text{ iff } \exists v \in W \text{ such that } \mathcal{M}, v \models \varphi
\end{aligned}$$

A formula φ is *true at* world $w \in W$ in model $\mathcal{M}$ if $\mathcal{M}, w \models \varphi$. φ is *globally true in* $\mathcal{M}$, denoted $\mathcal{M} \models \varphi$, if φ is *true at* every $w \in W$. Moreover, φ is *valid* (in a class of models $\mathcal{K}$) if *globally true in* every $\mathcal{M}$ ($\in \mathcal{K}$), denoted $\models_{\mathcal{PL}} \varphi$ ($\models_{\mathcal{K}} \varphi$).

Quite relevant to our purposes is the fact that $\mathcal{PL}$ introduces eight semantical definitions for binary preference operations on propositions ($\preceq_{EE}, \preceq_{AE}, \preceq_{EA}, \preceq_{AA}$, and their strict variants). They correspond, roughly speaking, to the four different ways of combining a pair of universal and existential quantifiers when "lifting" an ordering on worlds to an ordering on sets of worlds (i.e. propositions). In this respect $\mathcal{PL}$ can be seen as a family of preference logics encompassing, in particular, the proposals by von Wright [38] and Halpern [24]. $\mathcal{PL}$ appears well suited for effective automation using the SSE approach, which has been an important selection criterion. This judgment is based on good prior experience with the SSE of related (monadic) modal logics [14, 15] whose semantics employs Kripke-style relational semantics.

2.2 Encoding $\mathcal{PL}$ in Meta-logic HOL

We employ the *shallow semantical embeddings* (SSE) technique [7, 15] to encode (a semantical characterisation of) the logical connectives of an *object logic* as λ-expressions in HOL. This essentially shows that the object logic can be unraveled as a fragment of HOL and hence automated as such. For (multi-)modal normal logics, like $\mathcal{PL}$, the relevant semantical structures are Kripke-style relational frames. $\mathcal{PL}$ formulas can thus be encoded as predicates in HOL taking worlds as arguments.[1]

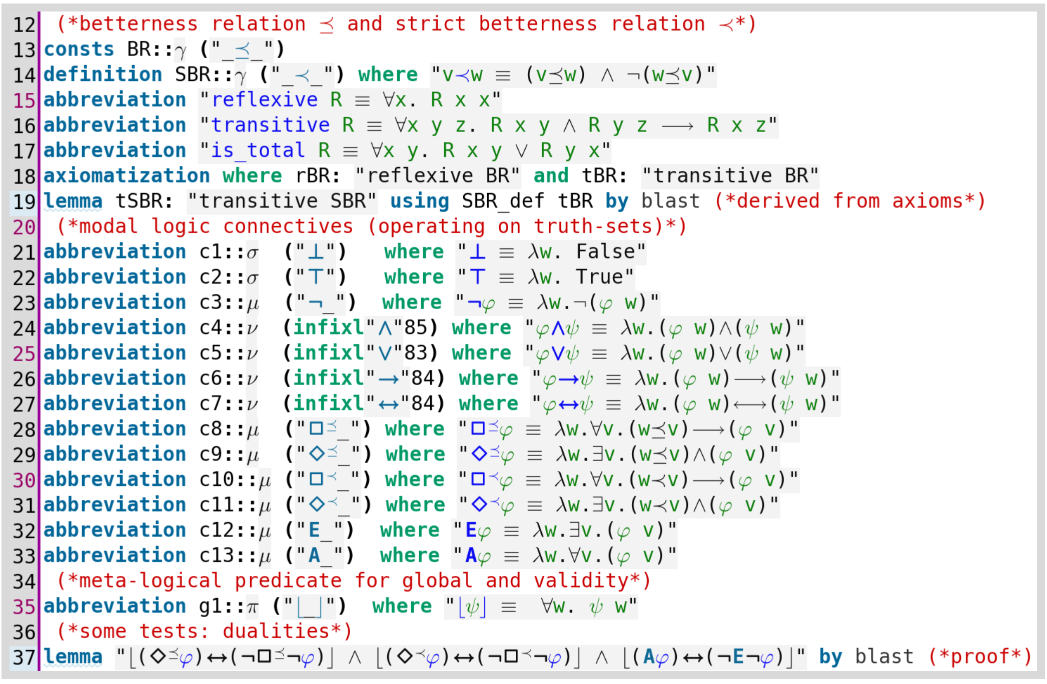
As a result, we obtain a combined, interactive and automated, theorem prover and model finder for (an extended variant of) $\mathcal{PL}$ realised within *Isabelle/HOL*. This is a new contribution, since we are not aware of any other existing implementation and automation of such a logic. Moreover, the SSE technique supports the automated assessment of meta-logical properties of the embedded logic at a semantical level, which in turn provides practical evidence for the correctness of our encoding.

We now give a succinct overview of the SSE of $\mathcal{PL}$ [1]. The embedding starts with declaring the HOL base type ι, corresponding to the domain of possible worlds/states in our formalisation. $\mathcal{PL}$ propositions are modelled as predicates on objects of type ι (i.e. as *truth-sets* of worlds) and, hence, they are given the type $(\iota \to o)$, which is abbreviated as σ in the remainder. The "betterness relation" $\preceq$ of $\mathcal{PL}$ is introduced as an uninterpreted constant symbol $\preceq_{(\iota\to\iota\to o)}$ in HOL, and its strict variant $\prec$ is introduced as an abbreviation $\prec_{(\iota\to\iota\to o)}$ standing for the HOL term $\lambda v \lambda w (v \preceq w \wedge \neg(w \preceq v))$; see Fig. 1. $\preceq$-accessible worlds are interpreted as those that are *at least as good* as the present one, and we hence postulate that $\preceq$ is a preorder, i.e. reflexive and transitive. In a next step the σ-type lifted logical connectives of $\mathcal{PL}$ are introduced as abbreviations for λ-terms in the meta-logic HOL. The conjunction operator $\wedge$ of $\mathcal{PL}$, for example, is introduced as an abbreviation $\wedge_{\sigma\to\sigma\to\sigma}$ which stands for the HOL term $\lambda\varphi_\sigma\lambda\psi_\sigma\lambda w_\iota(\varphi\ w\ \wedge\ \psi\ w)$, so that $\varphi_\sigma \wedge \psi_\sigma$ reduces to $\lambda w_\iota(\varphi\ w\ \wedge\ \psi\ w)$, denoting the set[2] of all worlds w in which both φ and ψ hold. Analogously, for negation, we introduce an abbreviation $\neg_{\sigma\to\sigma}$, which stands for $\lambda\varphi_\sigma\lambda w_\iota\neg(\varphi\ w)$.

The operators $\Diamond^{\preceq}$ and $\Diamond^{\prec}$ use $\preceq$ and $\prec$ as guards in their definitions. These operators are introduced as shorthands $\Diamond^{\preceq}_{\sigma\to\sigma}$ and $\Diamond^{\prec}_{\sigma\to\sigma}$ abbreviating the HOL terms $\lambda\varphi_\sigma\lambda w_\iota\exists v_\iota(w \preceq v \wedge \varphi\ v)$ and $\lambda\varphi_\sigma\lambda w_\iota\exists v_\iota(w \prec v \wedge \varphi\ v)$, respectively. $\Diamond^{\preceq}_{\sigma\to\sigma}\varphi_\sigma$ thus reduces to $\lambda w_\iota\exists v_\iota(w \preceq v \wedge \varphi\ v)$, denoting the set of all worlds w so that φ holds in some world v that is at least as good as w; analogous for $\Diamond^{\prec}_{\sigma\to\sigma}$. Additionally, the *global existential* modality $\mathbf{E}_{\sigma\to\sigma}$ is introduced as shorthand for the HOL term $\lambda\varphi_\sigma\lambda w_\iota\exists v_\iota(\varphi\ v)$. The duals $\Box^{\preceq}_{\sigma\to\sigma}\varphi_\sigma$, $\Box^{\prec}_{\sigma\to\sigma}\varphi_\sigma$ and $\mathbf{A}_{\sigma\to\sigma}\varphi$ can easily be added so that they are equivalent to $\neg\Diamond^{\preceq}_{\sigma\to\sigma}\neg\varphi_\sigma$, $\neg\Diamond^{\prec}_{\sigma\to\sigma}\neg\varphi_\sigma$ and $\neg\mathbf{E}_{\sigma\to\sigma}\neg\varphi$ respectively. A special predicate $\lfloor\varphi_\sigma\rfloor$ (read φ_σ is *valid*) for σ-type lifted $\mathcal{PL}$ formulas in HOL is defined as an abbreviation for $\forall w_\iota(\varphi_\sigma\, w)$.

1 This corresponds to the well-known standard translation to first-order logic. Observe, however, that the additional expressivity of HOL allows us to also encode and flexibly combine non-normal modal logics (conditional, deontic, etc.) and to encode also different kinds of quantifiers; see e.g. [6, 8, 9, 10].

2 In HOL (with Henkin semantics) sets are associated with their characteristic functions.



```
 (*betterness relation ≼ and strict betterness relation ≺*)
consts BR::γ ("_≼_")
definition SBR::γ ("_≺_") where "v≺w ≡ (v≼w) ∧ ¬(w≼v)"
abbreviation "reflexive R ≡ ∀x. R x x"
abbreviation "transitive R ≡ ∀x y z. R x y ∧ R y z ⟶ R x z"
abbreviation "is_total R ≡ ∀x y. R x y ∨ R y x"
axiomatization where rBR: "reflexive BR" and tBR: "transitive BR"
lemma tSBR: "transitive SBR" using SBR_def tBR by blast (*derived from axioms*)
 (*modal logic connectives (operating on truth-sets)*)
abbreviation c1::σ  ("⊥")   where "⊥ ≡ λw. False"
abbreviation c2::σ  ("⊤")   where "⊤ ≡ λw. True"
abbreviation c3::μ  ("¬_")  where "¬φ ≡ λw.¬(φ w)"
abbreviation c4::ν  (infixl"∧"85) where "φ∧ψ ≡ λw.(φ w)∧(ψ w)"
abbreviation c5::ν  (infixl"∨"83) where "φ∨ψ ≡ λw.(φ w)∨(ψ w)"
abbreviation c6::ν  (infixl"→"84) where "φ→ψ ≡ λw.(φ w)⟶(ψ w)"
abbreviation c7::ν  (infixl"↔"84) where "φ↔ψ ≡ λw.(φ w)⟷(ψ w)"
abbreviation c8::μ  ("□≼_") where "□≼φ ≡ λw.∀v.(w≼v)⟶(φ v)"
abbreviation c9::μ  ("◇≼_") where "◇≼φ ≡ λw.∃v.(w≼v)∧(φ v)"
abbreviation c10::μ ("□≺_") where "□≺φ ≡ λw.∀v.(w≺v)⟶(φ v)"
abbreviation c11::μ ("◇≺_") where "◇≺φ ≡ λw.∃v.(w≺v)∧(φ v)"
abbreviation c12::μ ("E_")  where "Eφ ≡ λw.∃v.(φ v)"
abbreviation c13::μ ("A_")  where "Aφ ≡ λw.∀v.(φ v)"
 (*meta-logical predicate for global and validity*)
abbreviation g1::π ("⌊_⌋")  where "⌊ψ⌋ ≡  ∀w. ψ w"
 (*some tests: dualities*)
lemma "⌊(◇≼φ)↔(¬□≼¬φ)⌋ ∧ ⌊(◇≺φ)↔(¬□≺¬φ)⌋ ∧ ⌊(Aφ)↔(¬E¬φ)⌋" by blast (*proof*)
```

Figure 1 SSE of basic $\mathcal{PL}$ in *Isabelle/HOL* (extract).

$\preceq$ is now "lifted" to a preference relation between $\mathcal{PL}$ propositions (sets of worlds).[3]

$$
\begin{array}{llll}
(\varphi_\sigma \preceq_{EE} \psi_\sigma)\ u_\iota & \text{iff} & \exists s_\iota\ \varphi_\sigma s \wedge (\exists t_\iota\ \psi_\sigma t \wedge s \preceq t) & (u_\iota \text{ arbitrary}) \\
(\varphi_\sigma \preceq_{EA} \psi_\sigma)\ u_\iota & \text{iff} & \exists t_\iota\ \psi_\sigma t \wedge (\forall s_\iota\ \varphi_\sigma s \rightarrow s \preceq t) & (u_\iota \text{ arbitrary}) \\
(\varphi_\sigma \preceq_{AE} \psi_\sigma)\ u_\iota & \text{iff} & \forall s_\iota\ \varphi_\sigma s \rightarrow (\exists t_\iota\ \psi_\sigma t \wedge s \preceq t) & (u_\iota \text{ arbitrary}) \\
(\varphi_\sigma \preceq_{AA} \psi_\sigma)\ u_\iota & \text{iff} & \forall s_\iota\ \varphi_\sigma s \rightarrow (\forall t_\iota\ \psi_\sigma t \rightarrow s \preceq t) & (u_\iota \text{ arbitrary})
\end{array}
$$

As an illustration, we can read $\varphi \prec_{AA} \psi$ as "every ψ-state being *better* than every φ-state", and read $\varphi \prec_{AE} \psi$ as "every φ-state having a *better* ψ-state" (similarly for others). Each of these non-trivial variants can be argued for [37, 27]. However, as we will reveal in §3, only the EA- and AE-variants satisfy the minimal conditions required for a logic of value aggregation. Moreover, they are the only ones that satisfy transitivity.

As shown in [37], the binary preference operators above are complemented by "syntactic" counterparts defined as derived operators using the language of $\mathcal{PL}$. In fact, both sets of definitions ("semantic" and "syntactic") coincide in general only for the EE- and AE-variants (other variants coincide only if $\preceq$ is a total/linear ordering). The "syntactic" variants are encoded below in HOL employing the σ-type lifted logic $\mathcal{PL}$ (using **boldface** to differentiate them).

$$
\begin{array}{lll}
(\varphi_\sigma \preceq_{EE} \psi_\sigma) := \boldsymbol{E}(\varphi_\sigma \wedge \Diamond^{\preceq}\psi_\sigma) & \text{and} & (\varphi_\sigma \prec_{EE} \psi_\sigma) := \boldsymbol{E}(\varphi_\sigma \wedge \Diamond^{\prec}\psi_\sigma) \\
(\varphi_\sigma \preceq_{EA} \psi_\sigma) := \boldsymbol{E}(\psi_\sigma \wedge \Box^{\prec}\neg\varphi_\sigma) & \text{and} & (\varphi_\sigma \prec_{EA} \psi_\sigma) := \boldsymbol{E}(\psi_\sigma \wedge \Box^{\preceq}\neg\varphi_\sigma) \\
(\varphi_\sigma \preceq_{AE} \psi_\sigma) := \boldsymbol{A}(\varphi_\sigma \rightarrow \Diamond^{\preceq}\psi_\sigma) & \text{and} & (\varphi_\sigma \prec_{AE} \psi_\sigma) := \boldsymbol{A}(\varphi_\sigma \rightarrow \Diamond^{\prec}\psi_\sigma) \\
(\varphi_\sigma \preceq_{AA} \psi_\sigma) := \boldsymbol{A}(\psi_\sigma \rightarrow \Box^{\prec}\neg\varphi_\sigma) & \text{and} & (\varphi_\sigma \prec_{AA} \psi_\sigma) := \boldsymbol{A}(\psi_\sigma \rightarrow \Box^{\preceq}\neg\varphi_\sigma)
\end{array}
$$

[3] The variant $\preceq_{EA}$ as originally presented in [37] was in fact wrongly formulated. This mistake has been uncovered during the (iterative) formalisation process thanks to *Isabelle/HOL*.

We further extend the lifted logic $\mathcal{PL}$ by adding quantifiers. This can be done by identifying $\forall x_\alpha s_\sigma$ with the HOL term $\lambda w_\iota \forall x_\alpha (s_\sigma w)$ and $\exists x_\alpha s_\sigma$ with $\lambda w_\iota \exists x_\alpha (s_\sigma w)$. This way quantified expressions can be seamlessly employed, e.g., for the representation of legal and world knowledge in §4.

A note on the heavy use of abbreviations as opposed to definitions is in order. One motivation is to show by the simplest possible means that the logic $\mathcal{PL}$ (and also our subsequent encodings in this paper) can be understood as a genuine fragment of HOL, and the introduction of the connectives of $\mathcal{PL}$ as syntactic sugar (abbreviations) for λ-terms in HOL does just that. No specific concepts, in particular no *Isabelle/HOL* specific ones, are needed to achieve our goals, making our work easily transferrable to other higher-order proof assistant systems. Another motivation is to show that proof automation with *Sledgehammer* already works very well using only abbreviations. Of course, by using definitions we could support, e.g., selective expansions of definitions, which could be a useful option for further proof optimisation. However, this was not yet necessary for the proof automation results obtained in this work. This, and related issues, are worth considering in further work.

2.3 Faithfulness of the SSE

The faithfulness (soundness & completeness) of the present SSE of $\mathcal{PL}$ in HOL follows from previous results for SSEs of propositional multi-modal logics [14] and their quantified extensions [15]. Soundness of the SSE states that our modelling does not give any "false positives", i.e., if $\models^{\mathrm{HOL}(\Gamma)} \lfloor \varphi_\sigma \rfloor$ then $\models_{\mathcal{PL}} \varphi$, and therefore $\vdash_{\mathcal{PL}} \varphi$ in the (complete) calculus axiomatised by [37]; here HOL(Γ) corresponds to HOL extended with the relevant types and constants plus a set Γ of axioms encoding $\mathcal{PL}$ semantic conditions, i.e., reflexivity and transitivity of $\preceq_{(\iota\to\iota\to o)}$. Completeness of the SSE means that our modelling does not give "false negatives", i.e., if $\models_{\mathcal{PL}} \varphi$ then $\models^{\mathrm{HOL}(\Gamma)} \lfloor \varphi_\sigma \rfloor$. Moreover, SSE completeness can be mechanically verified by deriving the σ-type lifted $\mathcal{PL}$ axioms and inference rules in HOL(Γ).[4]

3 A Logic for Value-oriented Legal Reasoning

On top of object logic $\mathcal{PL}$ we define a domain-specific logic for reasoning with values in the context of legal cases. We subsequently encode this logic of legal values in *Isabelle/HOL* and put it to the test.

Setting the Stage: Plaintiff vs. Defendant

In a preliminary step, the contending parties in a legal case, the "plaintiff" (p) and the "defendant" (d), are introduced as an (extensible) two-valued datatype *c* (for "contender") together with a function $(\cdot)^{-1}$ used to obtain for a given party the *other* one; i.e. $\mathtt{p}^{-1} = \mathtt{d}$ and $\mathtt{d}^{-1} = \mathtt{p}$. Moreover, we add a predicate For to model the ruling *for* a party and postulate: For $x \leftrightarrow \neg$For x^{-1}.

```
 (*new datatype for parties/contenders (there could be more in principle)*)
datatype c = p | d (*plaintiff & defendant*)
fun other::"c⇒c" ("_⁻¹") where "p⁻¹ = d" | "d⁻¹ = p"
 (*new constant symbol: finding/ruling for party*)
consts For::"c⇒σ"
axiomatization where ForAx: "⌊For x ↔ (¬For x⁻¹)⌋"
```

[4] See the corresponding sources in [1], where we conducted numerous experiments mechanically verifying meta-theoretical results on $\mathcal{PL}$.

Abstract Values and Value Principles

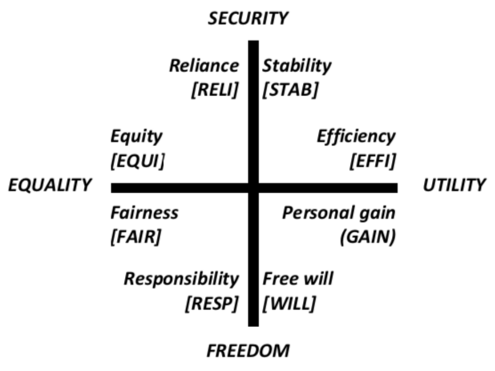


Figure 2 Value theory of Lomfeld [30].

Our approach to value-oriented legal reasoning draws upon recent work in legal theory by Lomfeld [30, 29] who considers a four-quadrant value space generated by two axes featuring antagonistic *abstract values* (FREEDOM vs. SECURITY & UTILITY vs. EQUALITY) at the extremes (Fig. 2).

A set of eight *value principles* are allocated to the four quadrants (two for each quadrant) as shown in Fig. 2. Additionally, Lomfeld's theory contemplates the encoding of legal rules as conditional preferences between conflicting value principles of the form: $R : (E_1 \wedge \cdots \wedge E_n) \Rightarrow A \prec B$. Hence, application of rule R involves *balancing* value principles A and B in context (i.e. under the conditions $E_1 \ldots E_n$).

To provide a concrete modelling of this theory in *Isabelle/HOL*, we have chosen to model value principles as sets of *abstract values*.[5] For the latter we have introduced a four-valued datatype ('t VAL). Observe that this datatype is parameterised with a type variable 't. In the remainder we take 't as being c. In doing this, we allow for the encoding of value principles w.r.t. a particular (favoured) legal party. In the remainder value principles are thus encoded as functions taking objects of type c (`p` or `d`) to sets of abstract values:

```
 (*new parameterized datatype for abstract values (wrt. a given party)*)
datatype 't VAL = FREEDOM 't | UTILITY 't | SECURITY 't | EQUALITY 't
type_synonym v = "(c)VAL⇒bool" (*principles: sets of (abstract) values*)
type_synonym cv = "c⇒v" (*principles are specified wrt. a given party*)
```

We have also introduced some convenient type-aliases; v for the type of sets of abstract values, and cv for its corresponding functional version (taking a legal party as parameter).

Instances of value principles (w.r.t. a legal party) are next introduced as sets of abstract values (w.r.t. a legal party), i.e., as objects of type cv. For this we introduce set-constructor operators for values (depicted as $\{\!|\ldots|\!\}$).

Recalling Fig. 2, we have, e.g., that the principle of STAB*ility* favouring the plaintiff (STAB^p) is encoded as a two-element set of abstract values (favouring the plaintiff), i.e., $\{\!|$SECURITY p, UTILITY $p|\!\}$. We do analogously for the other value principles.

[5] Here we suitably simplify Lomfeld's value theory to the effect that, e.g., STAB*ility* becomes identified with EFFI*ciency*. This is enough for our modelling work in §4. A more granular encoding of value principles is possible by adding a third axis to the value space in Fig. 2.

```
 (*notation for sets*)
abbreviation vset1 ("⦃_⦄") where "⦃φ⦄ ≡ λx::(c)VAL. x=φ"
abbreviation vset2 ("⦃_,_⦄")   where "⦃α,β⦄ ≡ λx::(c)VAL. x=α ∨ x=β"
 (*value principles*)
abbreviation stab::cv ("STAB_") where "STABˣ ≡ ⦃SECURITY x, UTILITY  x⦄"
abbreviation effi::cv ("EFFI_") where "EFFIˣ ≡ ⦃UTILITY  x, SECURITY x⦄"
abbreviation gain::cv ("GAIN_") where "GAINˣ ≡ ⦃UTILITY  x, FREEDOM  x⦄"
abbreviation will::cv ("WILL_") where "WILLˣ ≡ ⦃FREEDOM  x, UTILITY  x⦄"
abbreviation resp::cv ("RESP_") where "RESPˣ ≡ ⦃FREEDOM  x, EQUALITY x⦄"
abbreviation fair::cv ("FAIR_") where "FAIRˣ ≡ ⦃EQUALITY x, FREEDOM  x⦄"
abbreviation equi::cv ("EQUI_") where "EQUIˣ ≡ ⦃EQUALITY x, SECURITY x⦄"
abbreviation reli::cv ("RELI_") where "RELIˣ ≡ ⦃SECURITY x, EQUALITY x⦄"
```

From a modal logic point of view it is, alternatively, convenient to conceive value principles as truth-bearers, i.e., propositions (as sets of worlds or situations). To overcome this apparent dichotomy in the modelling of value principles (sets of abstract values vs. sets of worlds) we make use of the mathematical notion of a *Galois connection* as exemplified by the notion of *derivation operators* from the theory of *formal concept analysis* (FCA), a mathematical theory of concepts and concept hierarchies as formal ontologies. Below we succinctly discuss a couple of FCA notions relevant to our work. We refer the interested reader to [22] for an actual introduction to FCA.

Some FCA Notions

A *formal context* is a triple $K = \langle G, M, I \rangle$ where G is a set of *objects*, M is a set of *attributes*, and I is a relation between G and M (so-called *incidence relation*), i.e., $I \subseteq G \times M$. We read $\langle g, m \rangle \in I$ as "the object g has the attribute m". We define two so-called *derivation operators* $\uparrow$ and $\downarrow$ as follows:

$$\begin{aligned} A\uparrow &:= \{m \in M \mid \langle g, m\rangle \in I \text{ for all } g \in A\} && \text{for } A \subseteq G \\ B\downarrow &:= \{g \in G \mid \langle g, m\rangle \in I \text{ for all } m \in B\} && \text{for } B \subseteq M \end{aligned}$$

$A\uparrow$ is the set of all attributes shared by all objects from A, called the *intent* of A. Dually, $B\downarrow$ is the set of all objects sharing all attributes from B, called the *extent* of B. This pair of derivation operators thus forms an antitone *Galois connection* between (the powersets of) G and M, i.e. we always have that $B \subseteq A\uparrow$ iff $A \subseteq B\downarrow$.

A *formal concept* (in a context K) is defined as a pair $\langle A, B \rangle$ such that $A \subseteq G$, $B \subseteq M$, $A\uparrow = B$, and $B\downarrow = A$. We call A and B the *extent* and the *intent* of the concept $\langle A, B \rangle$, respectively. Indeed $\langle A\uparrow\downarrow, A\uparrow \rangle$ and $\langle B\downarrow, B\downarrow\uparrow \rangle$ are always concepts.

The set of concepts in a formal context is partially ordered by set inclusion of their extents, or, dually, by the (reversing) inclusion of their intents. In fact, for a given formal context this ordering forms a complete lattice: its *concept lattice*. Conversely, it can be shown that every complete lattice is isomorphic to the concept lattice of some formal context. We can thus define lattice-theoretical meet and join operations on FCA concepts in order to obtain an algebra of concepts:[6]

$$\begin{aligned} \langle A_1, B_1\rangle \wedge \langle A_2, B_2\rangle &:= \langle (A_1 \cap A_2)\ , (B_1 \cup B_2)\downarrow\uparrow\rangle \\ \langle A_1, B_1\rangle \vee \langle A_2, B_2\rangle &:= \langle (A_1 \cup A_2)\uparrow\downarrow\ , (B_1 \cap B_2)\rangle \end{aligned}$$

[6] This result can be seamlessly stated for infinite meets and joins (infima and suprema) in the usual way. It corresponds to the first part of the so-called *basic theorem on concept lattices* [22].

Value Principles

We now extend the encoding (SSE) of our object logic $\mathcal{PL}$, exploiting the high expressivity of our meta-logic HOL. We define two FCA derivation operators $\uparrow$ and $\downarrow$ employing the corresponding definitions from above. For this we take G as the domain set of worlds corresponding to the type ι and M as a domain set of abstract values, corresponding in the current modelling approach to the type *VAL*. In doing this, each value principle (set of abstract values) becomes associated with a proposition (set of worlds) by means of the operator $\downarrow$ (conversely for $\uparrow$). We encode this by defining a binary *incidence* relation $\mathcal{I}$ between worlds/states (type ι) and abstract values (type *VAL*). We define $\downarrow$ so that $V\downarrow$ denotes the set of all worlds that are $\mathcal{I}$-related to every value in V (analogously for $V\uparrow$).

We introduce an alternative notation: $[V] := V\downarrow$ which may enhance readability in some cases.

```
(**Value Theory*)
consts Irel::"ι⇒v" ("𝓘") (*incidence relation worlds-values*)
 (*derivation operators (cf. theory of "formal concept analysis") *)
abbreviation intent::"σ⇒v" ("_↑") where "W↑ ≡ λv. ∀x. W x ⟶ 𝓘 x v"
abbreviation extent::"v⇒σ" ("_↓") where "V↓ ≡ λw. ∀x. V x ⟶ 𝓘 w x"
abbreviation extent_brkt ("[_]") where "[V] ≡ V↓" (*alternative notation*)
```

Recalling the semantics of the object logic $\mathcal{PL}$ from our discussion in §2.1, we can give an intuitive reading for truth at a world in a preference model to terms of the form $P\downarrow$; namely, we can read $\mathcal{M}, w \vDash P\downarrow$ as "principle P provides a reason for (state of affairs) w to obtain". In the same vein, we can read $\mathcal{M} \vDash A \rightarrow P\downarrow$ as "principle P provides a reason for proposition A being the case".

Transferring these insights to our current modelling in *Isabelle/HOL*, we can intuitively read, e.g., the formula $\text{STAB}^d\!\downarrow w$ (of type *bool*) as: "the legal principle of stability is *justifiably* promoted in favour of the defendant (in situation w)". In a similar vein, we can read $\lfloor \text{For } d \rightarrow \text{STAB}^d\!\downarrow \rfloor$ as "promoting (legal) stability in favour of the defendant justifies deciding for him/her (in any situation)".

Value Aggregation and Preference

As discussed above, our logic of legal values must provide means for expressing conditional preferences between principles of the form: $(E_1 \wedge \cdots \wedge E_n) \Rightarrow A \prec B$. The conditional $\Rightarrow$ is modelled in this work using $\mathcal{PL}$'s material conditional $\rightarrow$, while noting that a defeasible conditional operator can indeed be defined and added by employing $\mathcal{PL}$'s modal operators [20, 28]. We can also define a binary preference connective $\prec$ for propositions by reusing any of the eight preference "lifting" variants in $\mathcal{PL}$ as discussed in §2. However, this choice cannot be arbitrary, since it needs to interact with value aggregation in an appropriate way.

Lomfeld's theory also contemplates a mechanism for expressing aggregation of value principles (as reasons). We thus define a binary value aggregation connective $\oplus$, observing that it should satisfy particular logical constraints in interaction with a (suitably selected) value preference relation $\prec$:

$$(A \prec B) \rightarrow (A \prec B \oplus C) \text{ but not } (A \prec B \oplus C) \rightarrow (A \prec B) \qquad \text{aggregation on the right}$$
$$(A \oplus C \prec B) \rightarrow (A \prec B) \text{ but not } (A \prec B) \rightarrow (A \oplus C \prec B) \qquad \text{aggregation on the left}$$
$$(B \prec A) \wedge (C \prec A) \rightarrow (B \oplus C \prec A) \qquad \text{union property (optional)}$$

The aggregation connectives are most conveniently defined using join (resp. set union) operators, which gives us commutativity. As it happens, only the $\prec_{AE}/\preceq_{AE}$ and $\prec_{EA}/\preceq_{EA}$ variants from §2 satisfy the first two conditions. They are also the only variants satisfying transitivity. Moreover, if we choose to enforce the third aggregation principle (union property), then we are left with only one variant to consider, namely $\prec_{AE}/\preceq_{AE}$. This variant also offers several benefits for our current modelling purposes: it can be faithfully encoded in the language of $\mathcal{PL}$ [37] and its behaviour is well documented in the literature [24] [27, Ch. 4].

After extensive computer-supported experiments in *Isabelle/HOL* (see [1]) we have identified the following candidate definitions satisfying all desiderata. First, for value aggregation $\oplus$:[7]

$$A \oplus_{(1)} B := \ (A \cap B)\downarrow \quad \text{and} \quad A \oplus_{(2)} B := \ (A\downarrow \vee B\downarrow)$$

Then, for a binary preference connective $\prec$ between propositions we have:

$$\varphi \prec_{(1)} \psi := \varphi \preceq_{AE} \psi \quad \text{and} \quad \varphi \prec_{(2)} \psi := \varphi \prec_{AE} \psi$$

For the rest of this work we will illustratively employ the second set of definitions indexed by (2).

Promoting Values

We still need to consider the mechanism by which we can link legal decisions, together with other legally relevant facts, to legal values. We conceive of such a mechanism as a sentence schema, which reads intuitively as: "Taking decision D in the presence of facts F promotes/advances legal (value) principle V". The formalisation of this schema corresponds to a new predicate *Promotes(F,D,V)*, where F is a conjunction of facts relevant to the case (a proposition), D is the legal decision, and V is the value principle thereby promoted.[8]

$$Promotes(F, D, V) := \quad F \to \Box^{\prec}(D \leftrightarrow \Diamond^{\prec} V\downarrow)$$

Promotes(F,D,V) can be given an intuitive reading: "in every F-situation we have that, in all better states, the admissibility of promoting value V both entails and justifies (as a reason) taking decision D".

```
 (*connective for aggregating value principles*)
abbreviation aggr ("[_⊕_]") where "[V₁⊕V₂] ≡ (V₁↓) ∨ (V₂↓)"
 (*chosen variant for preference relation (cf. Halpern (1997)*)
abbreviation pref::"σ⇒σ⇒σ"  ("_≺_") where "φ ≺ ψ ≡ φ ≺AE ψ"
 (*schema for value principle promotion*)
abbreviation "Promotes F D V ≡ ⌊F → □≺(D ↔ ◇≺(V↓))⌋"
```

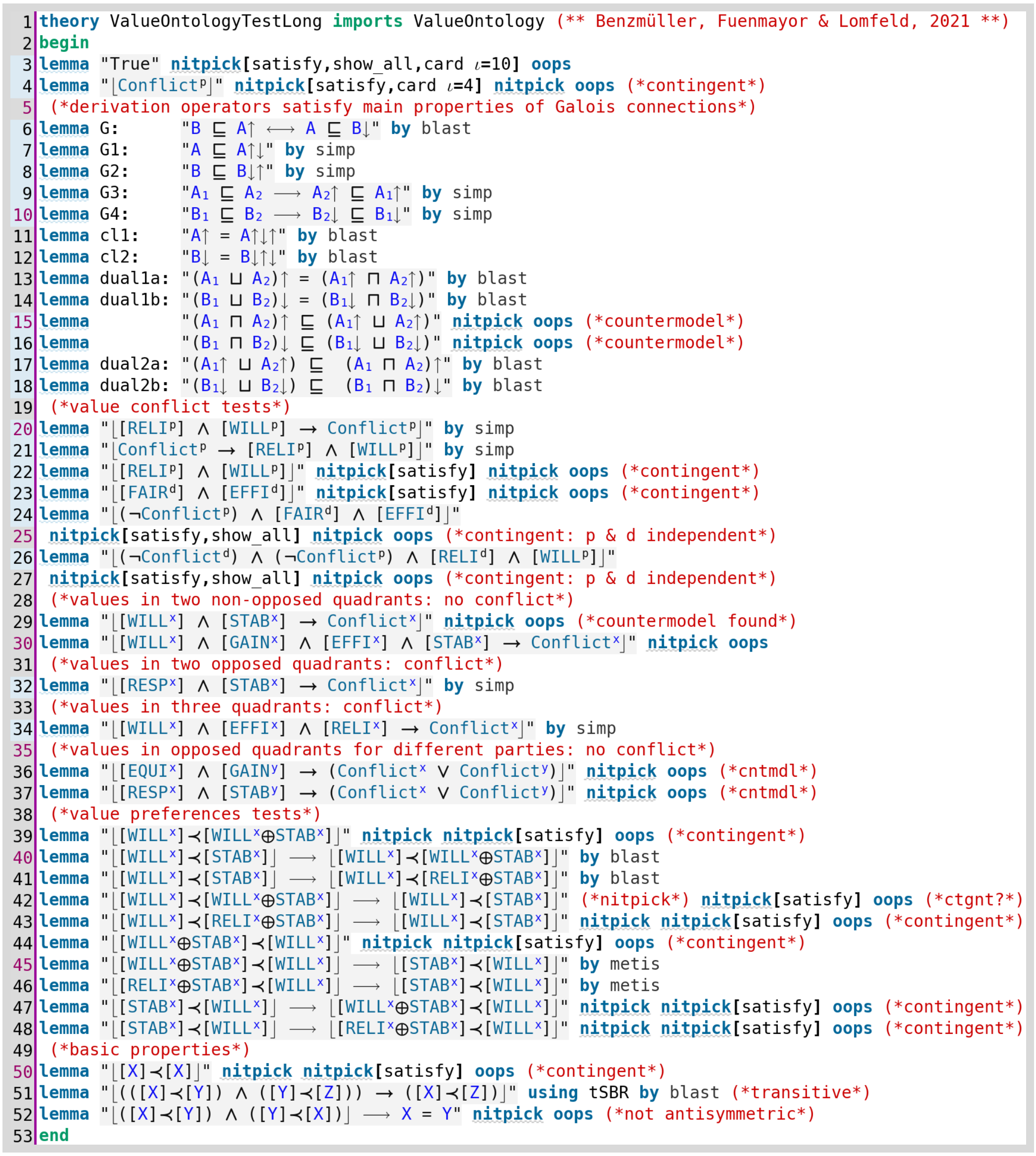
Value Conflict

Another important idea inspired from Lomfeld's value theory [29, 30] is the notion of value *conflict*. Recalling Fig. 2, values are disposed around two axis of value coordinates, with values lying at contrary poles playing antagonistic roles. For our modelling purposes it makes thus sense to consider a predicate *Conflict* on worlds (i.e. a proposition) signalling situations where value conflicts appear.

```
 (*proposition for testing for value conflict*)
abbreviation conflict ("Conflict₋") where (*conflict for value support*)
 "Conflictˣ ≡ [SECURITYˣ] ∧ [EQUALITYˣ] ∧ [FREEDOMˣ] ∧ [UTILITYˣ]"
```

[7] Observe that $\oplus_1$ is based upon the join operation on the corresponding FCA formal concepts. $\oplus_2$ is a strengthening of the first, since $(A \oplus_2 B) \subseteq (A \oplus_1 B)$.

[8] We adopt the terminology of *advancing* or *promoting* a value from the literature [16, 34, 5] understanding it in a teleological sense: a decision promoting a value principle means taking that decision *for the sake* of honouring the principle; thus seeing the value principle as a reason for taking that decision.



```
 1 theory ValueOntologyTestLong imports ValueOntology (** Benzmüller, Fuenmayor & Lomfeld, 2021 **)
 2 begin
 3 lemma "True" nitpick[satisfy,show_all,card ι=10] oops
 4 lemma "⌊Conflictᵖ⌋" nitpick[satisfy,card ι=4] nitpick oops (*contingent*)
 5  (*derivation operators satisfy main properties of Galois connections*)
 6 lemma G:      "B ⊑ A↑ ⟷ A ⊑ B↓" by blast
 7 lemma G1:     "A ⊑ A↑↓" by simp
 8 lemma G2:     "B ⊑ B↓↑" by simp
 9 lemma G3:     "A₁ ⊑ A₂ ⟶ A₂↑ ⊑ A₁↑" by simp
10 lemma G4:     "B₁ ⊑ B₂ ⟶ B₂↓ ⊑ B₁↓" by simp
11 lemma cl1:    "A↑ = A↑↓↑" by blast
12 lemma cl2:    "B↓ = B↓↑↓" by blast
13 lemma dual1a: "(A₁ ⊔ A₂)↑ = (A₁↑ ⊓ A₂↑)" by blast
14 lemma dual1b: "(B₁ ⊔ B₂)↓ = (B₁↓ ⊓ B₂↓)" by blast
15 lemma         "(A₁ ⊓ A₂)↑ ⊑ (A₁↑ ⊔ A₂↑)" nitpick oops (*countermodel*)
16 lemma         "(B₁ ⊓ B₂)↓ ⊑ (B₁↓ ⊔ B₂↓)" nitpick oops (*countermodel*)
17 lemma dual2a: "(A₁↑ ⊔ A₂↑) ⊑  (A₁ ⊓ A₂)↑" by blast
18 lemma dual2b: "(B₁↓ ⊔ B₂↓) ⊑  (B₁ ⊓ B₂)↓" by blast
19  (*value conflict tests*)
20 lemma "⌊[RELIᵖ] ∧ [WILLᵖ] → Conflictᵖ⌋" by simp
21 lemma "⌊Conflictᵖ → [RELIᵖ] ∧ [WILLᵖ]⌋" by simp
22 lemma "⌊[RELIᵖ] ∧ [WILLᵖ]⌋" nitpick[satisfy] nitpick oops (*contingent*)
23 lemma "⌊[FAIRᵈ] ∧ [EFFIᵈ]⌋" nitpick[satisfy] nitpick oops (*contingent*)
24 lemma "⌊(¬Conflictᵖ) ∧ [FAIRᵈ] ∧ [EFFIᵈ]⌋"
25  nitpick[satisfy,show_all] nitpick oops (*contingent: p & d independent*)
26 lemma "⌊(¬Conflictᵈ) ∧ (¬Conflictᵖ) ∧ [RELIᵈ] ∧ [WILLᵖ]⌋"
27  nitpick[satisfy,show_all] nitpick oops (*contingent: p & d independent*)
28  (*values in two non-opposed quadrants: no conflict*)
29 lemma "⌊[WILLˣ] ∧ [STABˣ] → Conflictˣ⌋" nitpick oops (*countermodel found*)
30 lemma "⌊[WILLˣ] ∧ [GAINˣ] ∧ [EFFIˣ] ∧ [STABˣ] → Conflictˣ⌋" nitpick oops
31  (*values in two opposed quadrants: conflict*)
32 lemma "⌊[RESPˣ] ∧ [STABˣ] → Conflictˣ⌋" by simp
33  (*values in three quadrants: conflict*)
34 lemma "⌊[WILLˣ] ∧ [EFFIˣ] ∧ [RELIˣ] → Conflictˣ⌋" by simp
35  (*values in opposed quadrants for different parties: no conflict*)
36 lemma "⌊[EQUIˣ] ∧ [GAINʸ] → (Conflictˣ ∨ Conflictʸ)⌋" nitpick oops (*cntmdl*)
37 lemma "⌊[RESPˣ] ∧ [STABʸ] → (Conflictˣ ∨ Conflictʸ)⌋" nitpick oops (*cntmdl*)
38  (*value preferences tests*)
39 lemma "⌊[WILLˣ]≺[WILLˣ⊕STABˣ]⌋" nitpick nitpick[satisfy] oops (*contingent*)
40 lemma "⌊[WILLˣ]≺[STABˣ]⌋ ⟶ ⌊[WILLˣ]≺[WILLˣ⊕STABˣ]⌋" by blast
41 lemma "⌊[WILLˣ]≺[STABˣ]⌋ ⟶ ⌊[WILLˣ]≺[RELIˣ⊕STABˣ]⌋" by blast
42 lemma "⌊[WILLˣ]≺[WILLˣ⊕STABˣ]⌋ ⟶ ⌊[WILLˣ]≺[STABˣ]⌋" (*nitpick*) nitpick[satisfy] oops (*ctgnt?*)
43 lemma "⌊[WILLˣ]≺[RELIˣ⊕STABˣ]⌋ ⟶ ⌊[WILLˣ]≺[STABˣ]⌋" nitpick nitpick[satisfy] oops (*contingent*)
44 lemma "⌊[WILLˣ⊕STABˣ]≺[WILLˣ]⌋" nitpick nitpick[satisfy] oops (*contingent*)
45 lemma "⌊[WILLˣ⊕STABˣ]≺[WILLˣ]⌋ ⟶ ⌊[STABˣ]≺[WILLˣ]⌋" by metis
46 lemma "⌊[RELIˣ⊕STABˣ]≺[WILLˣ]⌋ ⟶ ⌊[STABˣ]≺[WILLˣ]⌋" by metis
47 lemma "⌊[STABˣ]≺[WILLˣ]⌋ ⟶ ⌊[WILLˣ⊕STABˣ]≺[WILLˣ]⌋" nitpick nitpick[satisfy] oops (*contingent*)
48 lemma "⌊[STABˣ]≺[WILLˣ]⌋ ⟶ ⌊[RELIˣ⊕STABˣ]≺[WILLˣ]⌋" nitpick nitpick[satisfy] oops (*contingent*)
49  (*basic properties*)
50 lemma "⌊[X]≺[X]⌋" nitpick nitpick[satisfy] oops (*contingent*)
51 lemma "⌊(([X]≺[Y]) ∧ ([Y]≺[Z])) → ([X]≺[Z])⌋" using tSBR by blast (*transitive*)
52 lemma "⌊([X]≺[Y]) ∧ ([Y]≺[X])⌋ ⟶ X = Y" nitpick oops (*not antisymmetric*)
53 end
```

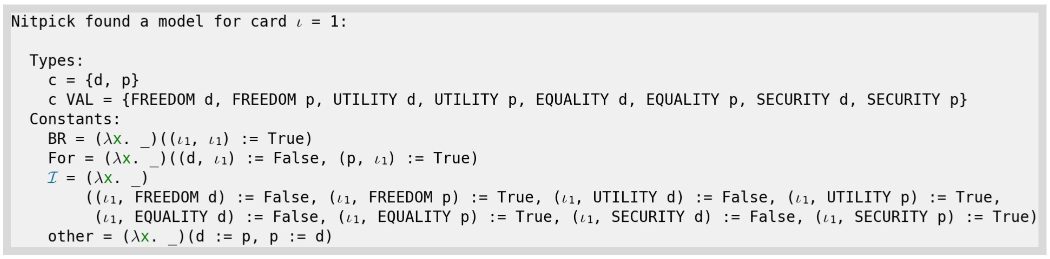
Figure 3 Testing the logic of legal values.

Testing the Encoding

In order to test the adequacy of our modelling, some implied and non-implied knowledge is studied. We briefly discuss some of the conducted tests as shown in Fig. 3.

Among others, we verify that the pair of operators for *extension* (↓) and *intension* (↑), cf. *formal concept analysis* [22], constitute indeed a Galois connection (Lines 6–18), and we carry out some further tests on the value theory concerning value aggregation and consistency (Lines 20ff.).

In our modelling of the notion of *value conflict*, promoting values (for the same party) from two opposing value quadrants, say RELI & WILL, should entail a value conflict; theorem provers quickly confirm this as shown in Fig. 3 (Line 20). However, promoting values from two non-opposed quadrants, such as WILL & STAB (Line 29) should not imply conflict:



```
Nitpick found a model for card ι = 1:

  Types:
    c = {d, p}
    c VAL = {FREEDOM d, FREEDOM p, UTILITY d, UTILITY p, EQUALITY d, EQUALITY p, SECURITY d, SECURITY p}
  Constants:
    BR = (λx. _)((ι1, ι1) := True)
    For = (λx. _)((d, ι1) := False, (p, ι1) := True)
    I = (λx. _)
        ((ι1, FREEDOM d) := False, (ι1, FREEDOM p) := True, (ι1, UTILITY d) := False, (ι1, UTILITY p) := True,
         (ι1, EQUALITY d) := False, (ι1, EQUALITY p) := True, (ι1, SECURITY d) := False, (ι1, SECURITY p) := True)
    other = (λx. _)(d := p, p := d)
```

Figure 4 Satisfying model for the statement in Line 22 of Fig. 3.

the model finder *Nitpick*[9] computes and reports a countermodel (not shown here) to the stated conjecture. A value conflict is also not implied if values from opposing quadrants are promoted for different parties (Lines 36-37).

Note that the notion of value conflict has deliberately not been aligned with inconsistency in meta-logic HOL. This way we can represent conflict situations in which, for instance, RELI and WILL (being conflicting values, see Line 20 in Fig. 3) are promoted for the plaintiff (p), without leading to a logical inconsistency in *Isabelle/HOL* (thus avoiding "explosion"). In Line 22 of Fig. 3, for example, *Nitpick* is called simultaneously in both modes in order to confirm the contingency of the statement; as expected both a model (cf. Fig. 4) and countermodel (not displayed here) for the statement are returned. This value conflict (w.r.t. p) can also be spotted by inspecting the satisfying models generated by *Nitpick*. One of such models is depicted in Fig. 4, where it is shown that (in the given possible world ι_1) all of the abstract values (EQUALITY, SECURITY, UTILITY, and FREEDOM) are simultaneously promoted for p, which implies a value conflict according to our definition.

Analysing the model structures returned by *Nitpick* has indeed been very helpful to gain a deeper insight into $\mathcal{PL}$ semantic structures. This becomes particularly relevant for complex modelling tasks where a clear understanding is often initially lacking.

Further tests in Fig. 3 (Lines 39-48) assess the behaviour of the aggregation operator $\oplus$ in combination with value preferences. We test for a correct behaviour when "strengthening", resp. "weakening", the right-hand side (Lines 39-43). As an illustration, in line 41, if STAB is preferred over WILL, then STAB combined with, say, RELI is also preferred over WILL alone. Similar test are conducted for "strengthening", resp. "weakening", the left-hand side (Lines 44-48).

Finally, we verify (lines 50–52) basic properties of the preference relation.

4 A Case Study in Property Law

To illustrate our approach, we formalise and assess, employing *Isabelle/HOL*, a well-known benchmark case in AI & Law involving the appropriation of wild animals: *Pierson vs. Post*. In a nutshell: *Pierson killed and carried off a fox which Post already was hunting with hounds on public land. The Court found for Pierson* (cf. [2, 34, 16], and also [23] for the significance of this case as a benchmark).

We start with some words on the modelling of background (legal & world) knowledge.

[9] *Nitpick* [19] searches for, respectively enumerates, finite models or countermodels to a conjectured statement/lemma. By default *Nitpick* searches for countermodels, and model finding is enforced by stating the parameter keyword "satisfy". These models are given as concrete interpretations of relevant terms in the given context so that the conjectured statement is satisfied or falsified.

4.1 Legal & World Knowledge

The realistic modelling of concrete legal cases requires further legal & world knowledge (LWK) to be taken into account. For the sake of illustration, we introduce here only a small and monolithic *Isabelle/HOL* theory[10] called "GeneralKnowledge". This includes a small excerpt of a much simplified "animal appropriation taxonomy", where we associate "animal appropriation" kinds of situations with the value preferences they imply (as conditional preference relations).

In a realistic setting this knowledge base would be further split and structured similarly to other legal or general ontologies, e.g., in the *Semantic Web*. Note, however, that the expressiveness in our approach, unlike in many other legal ontologies or taxonomies, is by no means limited to definite underlying (but fixed) logical language foundations. We could thus easily decide for a more realistic modelling, e.g., avoiding simplifying propositional abstractions. For instance, the proposition "appWildAnimal", representing the appropriation of one or more wild animals, can anytime be replaced by a more complex formula (featuring, e.g., quantifiers, modalities or defeasible conditionals).

We now briefly outline the encoding of our example LWK (see [1] for the full details).

First, some non-logical constants that stand for kinds of legally relevant situations (here: of appropriation) are introduced, and their meaning is constrained by some postulates:

```
(*LWK: kinds of situations addressed*)
consts appObject::σ  appAnimal::σ (*appropriation of objects/animals in general*)
       appWildAnimal::σ  appDomAnimal::σ (*appropriation of wild/domestic animals*)
(*LWK: postulates for kinds of situations*)
axiomatization where
 W1: "⌊appAnimal ⟶ appObject⌋" and
 W2: "⌊¬(appWildAnimal ∧ appDomAnimal)⌋" and
 W3: "⌊appWildAnimal ⟶ appAnimal⌋"  and
 W4: "⌊appDomAnimal ⟶ appAnimal⌋"
```

Then the "default" legal rules for several situations (here: appropriation of animals) are formulated as conditional preference relations:

```
(*LWK: (prima facie) value preferences for kinds of situations*)
axiomatization where
 R1: "⌊appAnimal ⟶ ([STABᵖ] ≺ [STABᵈ])⌋" and
 R2: "⌊appWildAnimal ⟶ ([WILLˣ⁻¹] ≺ [STABˣ])⌋" and
 R3: "⌊appDomAnimal  ⟶ ([STABˣ⁻¹] ≺ [RELIˣ⊕RESPˣ])⌋"
```

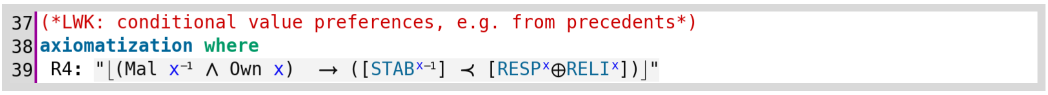
For example, rule R2 could be read as: "In a wild-animals-appropriation kind of situation, promoting STAB*ility* in favour of a party (say, the plaintiff) is preferred over promoting WILL in favour of the other party (defendant)". If there is no more specific legal rule from a precedent or a codified statute then these "default'[11] preference relations determine the result. Moreover, we can have rules conditioned on more concrete legal *factors*.[12] As a

[10] Isabelle documents are suggestively called "theories". They correspond to top-level modules bundling together related definitions, theories, proofs, etc.

[11] We use of the term "default" in the colloquial sense, noting however, that there exist in fact several (non-monotonic) logical systems aimed at modelling such a kind of *defeasible* behaviour for rules/conditionals (i.e., meaning that they can be "overruled"). One of them has been suggestively called "default logic". We refer to [25] for a discussion.

[12] The introduction of legal *factors* is an established practice in the implementation of case-based legal systems (cf. [3] for an overview). They can be conceived –as we do– as propositions abstracted from the facts of a case by the analyst/modeller in order to allow for assessing and comparing cases at a higher level of abstraction. Factors are typically either pro-plaintiff or pro-defendant, and their being true or false (resp. present or absent) in a concrete case can serve to invoke relevant precedents or statutes.

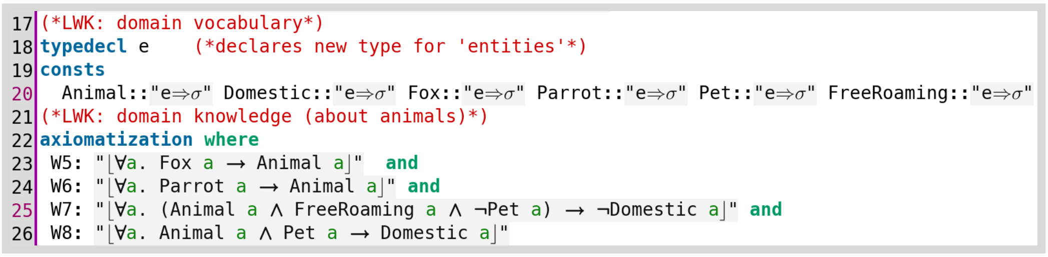
didactic example, the legal rule R4 states that the Own*ership* (say, the plaintiff's) of the land on which the appropriation took place, together with the fact that the opposing party (defendant) acted out of Mal*ice* implies a value preference of RELI*ance* and RESP*onsibility* over STAB*ility*. This last rule has indeed been chosen to reflect the famous common law precedent of Keeble vs. Hickeringill [16, 2].



```
(*LWK: conditional value preferences, e.g. from precedents*)
axiomatization where
 R4: "⌊(Mal x⁻¹ ∧ Own x)  ⟶ ([STABˣ⁻¹] ≺ [RESPˣ⊕RELIˣ])⌋"
```

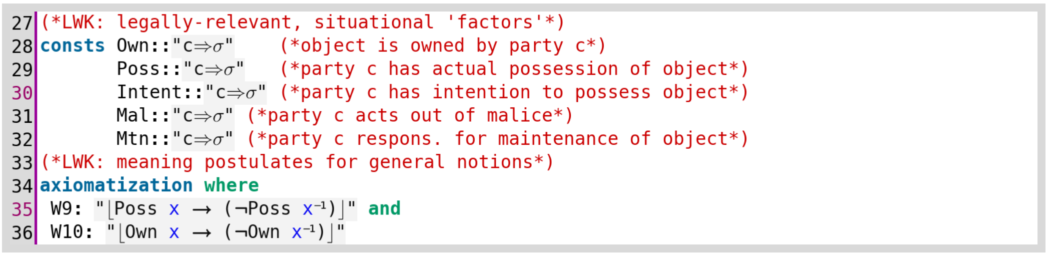
As already discussed, for ease of illustration, terms like "appWildAnimal" are modelled here as simple propositional constants. In practice, however, they may later be replaced, or logically implied, by a more realistic modelling of the relevant situational facts, utilising suitably complex (even higher-order, if needed) formulas depicting states of affairs to some desired level of granularity.

For the sake of modelling the appropriation of objects, we have introduced an additional type e (for "entities") that can be employed for terms denoting individuals (things, animals, etc.) when modelling legally relevant situations. Some simple vocabulary and taxonomic relationships (here for wild and domestic animals) are specified to illustrate this.



```
(*LWK: domain vocabulary*)
typedecl e    (*declares new type for 'entities'*)
consts
  Animal::"e⇒σ" Domestic::"e⇒σ" Fox::"e⇒σ" Parrot::"e⇒σ" Pet::"e⇒σ" FreeRoaming::"e⇒σ"
(*LWK: domain knowledge (about animals)*)
axiomatization where
 W5: "⌊∀a. Fox a ⟶ Animal a⌋"  and
 W6: "⌊∀a. Parrot a ⟶ Animal a⌋" and
 W7: "⌊∀a. (Animal a ∧ FreeRoaming a ∧ ¬Pet a) ⟶ ¬Domestic a⌋" and
 W8: "⌊∀a. Animal a ∧ Pet a ⟶ Domestic a⌋"
```

As mentioned before, we have introduced some convenient legal *factors* into our example LWK to allow for the encoding of legal knowledge originating from precedents or statutes at a more abstract level. In our approach these factors are to be logically implied (as deductive arguments) from the concrete facts of the case (as exemplified in §4 below). Observe that our framework also allows us to introduce definitions for those factors for which clear legal specifications exist. At the present stage, we will provide some simple postulates constraining factors' interpretation.



```
(*LWK: legally-relevant, situational 'factors'*)
consts Own::"c⇒σ"    (*object is owned by party c*)
       Poss::"c⇒σ"   (*party c has actual possession of object*)
       Intent::"c⇒σ" (*party c has intention to possess object*)
       Mal::"c⇒σ" (*party c acts out of malice*)
       Mtn::"c⇒σ" (*party c respons. for maintenance of object*)
(*LWK: meaning postulates for general notions*)
axiomatization where
 W9: "⌊Poss x ⟶ (¬Poss x⁻¹)⌋" and
 W10: "⌊Own x ⟶ (¬Own x⁻¹)⌋"
```

Recalling §3 we relate the introduced factors to value principles and outcomes by means of the *Promotes* predicate. Finally, the consistency of all axioms and rules provided is confirmed by *Nitpick*.

```
(*LWK: relate values, outcomes and situational 'factors'*)
axiomatization where
 F1: "Promotes (Intent x) (For x) WILLˣ" and
 F2: "Promotes (Mal x) (For x⁻¹) RESPˣ" and
 F3: "Promotes (Poss x) (For x) STABˣ" and
 F4: "Promotes (Mtn x) (For x) RESPˣ" and
 F5: "Promotes (Own x) (For x) RELIˣ"
(*Theory is consistent, (non-trivial) model found*)
lemma True nitpick[satisfy,card ι=4] oops
```

4.2 Pierson vs. Post

We illustrate our reasoning framework by encoding the classic property law case Pierson vs. Post.

Ruling for Pierson

The formal modelling of an argument in favour of Pierson is outlined next (the entire formalisation of this argument is presented in the sources [1]).

First we introduce some minimal vocabulary: a constant α of type e (denoting the appropriated animal), and the relations *pursue* and *capture* between the animal and one of the parties (of type c). A background (generic) theory as well as the (contingent) case facts as suitably interpreted by Pierson's party are then stipulated:

```
 (*case-specific 'world-vocabulary'*)
consts α::"e" (*appropriated animal (fox in this case) *)
consts Pursue::"c⇒e⇒σ" Capture::"c⇒e⇒σ"
(************** pro-defendant (Pierson) argument **************)
 (*defendant's theory*)
abbreviation "dT1 ≡ ⌊(∃c. Capture c α ∧ ¬Domestic α) → appWildAnimal⌋"
abbreviation "dT2 ≡ ⌊∀c. Pursue c α → Intent c⌋"
abbreviation "dT3 ≡ ⌊∀c. Capture c α → Poss c⌋"
abbreviation "d_theory ≡ dT1 ∧ dT2 ∧ dT3"
 (*defendant's facts*)
abbreviation "dF1 w ≡ Fox α w"
abbreviation "dF2 w ≡ FreeRoaming α w"
abbreviation "dF3 w ≡ ¬Pet α w"
abbreviation "dF4 w ≡ Pursue p α w"
abbreviation "dF5 w ≡ Capture d α w"
abbreviation "d_facts ≡ dF1 ∧ dF2 ∧ dF3 ∧ dF4 ∧ dF5"
```

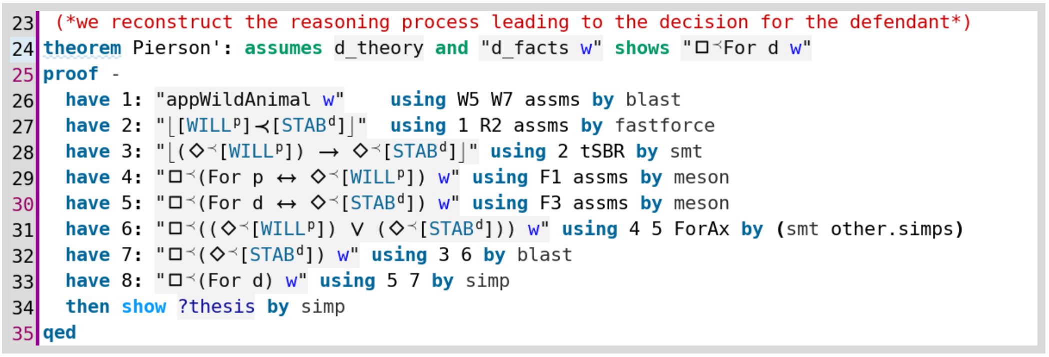
The aforementioned decision of the court for Pierson was justified by the majority opinion. The essential preference relation in the case is implied in the idea that appropriation of (free-roaming) wild animals requires actual corporal possession. The manifest corporal link to the possessor creates legal certainty, which is represented by the value *stability* (STAB) and outweighs the mere *will* to possess (WILL) by the plaintiff; cf. the arguments of classic lawyers cited by the majority opinion [23]: "pursuit alone vests no property" (Justinian institutes), and "corporal possession creates legal certainty" (Pufendorf). Recalling Fig. 2 in §3, this corresponds to a preference for the abstract value SECURITY over FREEDOM.

We can see that this legal rule R2, as introduced in the previous section (§4.1) is indeed employed by *Isabelle/HOL*'s automated tools to prove that, given a suitable defendant's theory, the (contingent) facts imply a decision in favour of Pierson in all "better' worlds (which we could read deontically as a sort of obligation):

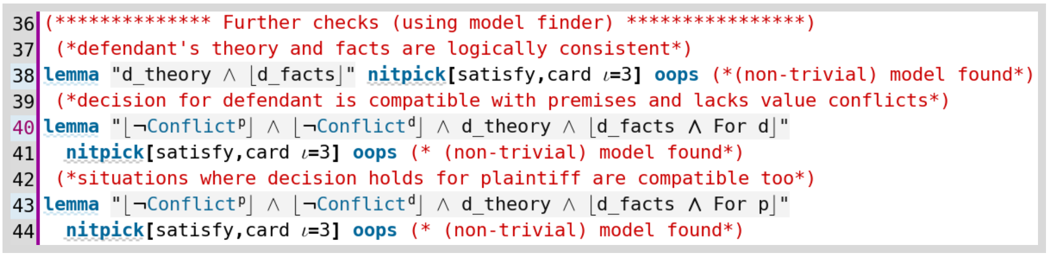
```
 (*decision for defendant (Pierson) can be proven automatically*)
theorem Pierson: "d_theory ⟶ ⌊d_facts → □≺For d⌋"
  by (smt F1 F3 ForAx R2 W5 W7 other.simps tSBR)
```

The previous "one-liner" proof has indeed been suggested by *Sledgehammer* [17, 18] which we credit, together with *Nitpick* [19], for doing the heavy lifting in our work. A proof argument in favour of Pierson that uses the same dependencies can also be constructed interactively using *Isabelle*'s human-readable proof language *Isar* [39]. The individual steps of the proof are this time formulated with respect to an explicit world/situation parameter w. The argument goes roughly as follows:

1. From Pierson's facts and theory we infer that in the disputed situation w a wild animal has been appropriated: appWildAnimal w.
2. In this context, by applying the value preference rule R2, we get that promoting STAB in favour of Pierson is preferred over promoting WILL in favour of Post: $\lfloor[\text{WILL}^p] \prec [\text{STAB}^d]\rfloor$.
3. The admissibility of promoting WILL in favour of Post thus entails the admissibility of promoting STAB in favour of Pierson: $\lfloor\Diamond^{\prec}[\text{WILL}^p] \rightarrow \Diamond^{\prec}[\text{STAB}^d]\rfloor$.
4. Moreover, after instantiating the *value promotion* schema F1 (§4.1) for Post (p), and acknowledging that his pursuing of the animal (Pursue p α) entails his intention to possess (Intent p), we obtain (for the given situation w) an obligation/recommendation to "align" any ruling for Post with the admissibility of promoting WILL in his favour: $\Box^{\prec}(\text{For } p \leftrightarrow \Diamond^{\prec}[\text{WILL}^p])\ w$.
5. Analogously, in view of Pierson's (d) capture of the animal (Capture d α), thus having taken possession of it (Poss d), we infer from the instantiation of *value promotion* schema F3 (for Pierson) an obligation/recommendation to align a ruling for Pierson with the admissibility of promoting the value principle STAB (in his favour): $\Box^{\prec}(\text{For } d \leftrightarrow \Diamond^{\prec}[\text{STAB}^d])\ w$.
6. From (4) and (5) in combination with the courts duty to find a ruling for one of both parties (ForAx) we infer, for the given situation w, that either the admissibility of promoting WILL in favour of Post or the admissibility of promoting STAB in favour of Pierson (or both) hold in every "better" world/situation (thus becoming a recommended/obligatory condition): $\Box^{\prec}(\Diamond^{\prec}[\text{WILL}^p] \vee \Diamond^{\prec}[\text{STAB}^d])\ w$.
7. From this and (3) we thus get that the admissibility of promoting STAB in favour of Pierson is recommended/obligatory in the given context w: $\Box^{\prec}(\Diamond^{\prec}[\text{STAB}^d])\ w$.
8. And this together with (5) finally implies the recomendation/obligation to rule in favour of Pierson in the given context w: $\Box^{\prec}(\text{For } d\ v)$.


```
23  (*we reconstruct the reasoning process leading to the decision for the defendant*)
24 theorem Pierson': assumes d_theory and "d_facts w" shows "□≺For d w"
25 proof -
26   have 1: "appWildAnimal w"    using W5 W7 assms by blast
27   have 2: "⌊[WILLᵖ]≺[STABᵈ]⌋"  using 1 R2 assms by fastforce
28   have 3: "⌊(◇≺[WILLᵖ]) → ◇≺[STABᵈ]⌋" using 2 tSBR by smt
29   have 4: "□≺(For p ↔ ◇≺[WILLᵖ]) w" using F1 assms by meson
30   have 5: "□≺(For d ↔ ◇≺[STABᵈ]) w" using F3 assms by meson
31   have 6: "□≺((◇≺[WILLᵖ]) ∨ (◇≺[STABᵈ])) w" using 4 5 ForAx by (smt other.simps)
32   have 7: "□≺(◇≺[STABᵈ]) w" using 3 6 by blast
33   have 8: "□≺(For d) w" using 5 7 by simp
34   then show ?thesis by simp
35 qed
```

The consistency of Pierson's assumptions (theory and facts) together with the other postulates from the previously introduced Isabelle theories "GeneralKnowledge" and "Value-Ontology" is verified by generating a (non-trivial) model using *Nitpick* (Line 38). Further tests confirm that the decision for Pierson (and analogously for Post) is compatible with the premises and, moreover, that for neither party value conflicts are implied.

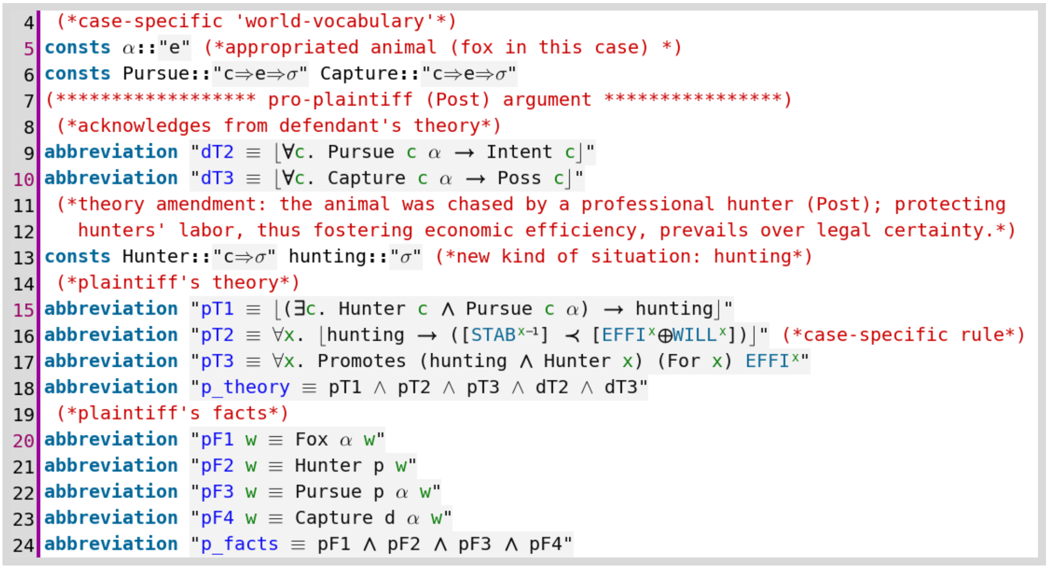


```
(************** Further checks (using model finder) ***************)
 (*defendant's theory and facts are logically consistent*)
lemma "d_theory ∧ ⌊d_facts⌋" nitpick[satisfy,card ι=3] oops (*(non-trivial) model found*)
 (*decision for defendant is compatible with premises and lacks value conflicts*)
lemma "⌊¬Conflict^p⌋ ∧ ⌊¬Conflict^d⌋ ∧ d_theory ∧ ⌊d_facts ∧ For d⌋"
  nitpick[satisfy,card ι=3] oops (* (non-trivial) model found*)
 (*situations where decision holds for plaintiff are compatible too*)
lemma "⌊¬Conflict^p⌋ ∧ ⌊¬Conflict^d⌋ ∧ d_theory ∧ ⌊d_facts ∧ For p⌋"
  nitpick[satisfy,card ι=3] oops (* (non-trivial) model found*)
```

Finally, observe that an analogous (deductively valid) argument for Post cannot follow from the given theory and situational facts. This is not surprising given that they have been deliberately chosen to suit Pierson's case. We show next, how it is indeed possible to construct a case (theory) suiting Post using our approach.

Ruling for Post

We model a possible counterargument by Post claiming an interpretation (i.e. a *distinction* in case law methodology) in that the animal, being vigorously pursued (with large dogs and hounds) by a professional hunter, is not "free-roaming". In doing this, the value preference $\lfloor[\mathrm{WILL}^p] \prec [\mathrm{STAB}^d]\rfloor$ (for appropriation of wild animals) as in the previous Pierson's argument does not obtain. Furthermore, Post's party postulates an alternative (suitable) value preference for hunting situations.



```
 4  (*case-specific 'world-vocabulary'*)
 5 consts α::"e" (*appropriated animal (fox in this case) *)
 6 consts Pursue::"c⇒e⇒σ" Capture::"c⇒e⇒σ"
 7 (****************** pro-plaintiff (Post) argument ***************)
 8  (*acknowledges from defendant's theory*)
 9 abbreviation "dT2 ≡ ⌊∀c. Pursue c α → Intent c⌋"
10 abbreviation "dT3 ≡ ⌊∀c. Capture c α → Poss c⌋"
11  (*theory amendment: the animal was chased by a professional hunter (Post); protecting
12    hunters' labor, thus fostering economic efficiency, prevails over legal certainty.*)
13 consts Hunter::"c⇒σ" hunting::"σ" (*new kind of situation: hunting*)
14  (*plaintiff's theory*)
15 abbreviation "pT1 ≡ ⌊(∃c. Hunter c ∧ Pursue c α) → hunting⌋"
16 abbreviation "pT2 ≡ ∀x. ⌊hunting → ([STAB^x-1] ≺ [EFFI^x⊕WILL^x])⌋" (*case-specific rule*)
17 abbreviation "pT3 ≡ ∀x. Promotes (hunting ∧ Hunter x) (For x) EFFI^x"
18 abbreviation "p_theory ≡ pT1 ∧ pT2 ∧ pT3 ∧ dT2 ∧ dT3"
19  (*plaintiff's facts*)
20 abbreviation "pF1 w ≡ Fox α w"
21 abbreviation "pF2 w ≡ Hunter p w"
22 abbreviation "pF3 w ≡ Pursue p α w"
23 abbreviation "pF4 w ≡ Capture d α w"
24 abbreviation "p_facts ≡ pF1 ∧ pF2 ∧ pF3 ∧ pF4"
```

Note that an alternative legal rule (i.e. a possible argument for *overruling* in case law methodology) is presented in Line 16 above, entailing a value preference of the value combination *efficiency* (EFFI) and *will* (WILL) over *stability* (STAB): $\lfloor[\mathrm{STAB}^d] \prec [\mathrm{EFFI}^p \oplus \mathrm{WILL}^p]\rfloor$. Following the argument put forward by the dissenting opinion in the original case, we might justify this new rule (inverting the initial value preference in the presence of EFFI) by pointing to the alleged public benefit of hunters getting rid of foxes, since the latter cause depredations in farms.

Accepting these modified assumptions the deductive validity of a decision for Post can in fact be proved and confirmed automatically, again, thanks to *Sledgehammer*:

```
 (*decision for plaintiff (Post) can be proven automatically (needs approx. 20s)*)
theorem Post: "p_theory ⟶ ⌊p_facts → □≺For p⌋"
  by (smt F1 F3 ForAx tBR SBR_def other.simps)
```

Similar to above, a detailed, interactive proof for the argument in favour of Post has been encoded and verified in *Isabelle/Isar*. We have also conducted further tests confirming the consistency of the assumptions and the absence of value conflicts (see sources in [1]).

5 Conclusion

Supporting interactive and automated value-oriented legal argumentation on the computer is a non-trivial challenge which we address, for reasons as defended e.g. by Bench-Capon [4], with symbolic AI techniques and formal methods. Motivated by recent pleas for *explainable and trustworthy AI*, our primary goal is to work towards the development of ethico-legal governors for future generations of intelligent system, or more generally, towards some form of (legally and ethically) *reasonable machines* [12], capable of exchanging rational justifications for the actions they take. While building up a capacity to engage in value-oriented legal argumentation is just one of a multitude of challenges this vision is faced with, it would clearly constitute an important stepping stone.

Custom software systems for legal case-based reasoning have been developed in the AI & law community, beginning with the influential HYPO system in the 1980s [36] (cf. also the review paper [3]). In later years, there was a gradual shift of interest from rule-based non-monotonic reasoning (e.g., logic programming) to argumentation-based approaches (see [35] for an overview); however, we are not aware of any other work that uses higher-order theorem proving and proof assistants (the argumentation logic of [26] is an early related effort that is worth mentioning). Another important aspect of our work concerns value-based legal reasoning and deliberation, where a considerable amount of work has been presented in response to the challenge posed by Berman and Hafner [16]. Our approach, based mainly on Lomfeld's theory [30, 29], has also been influenced by some of this work, in particular [34, 2, 5]. We think that some of the recent work that uses expressive deontic logics for value balancing (cf. [31] and the references therein) can be integrated into our approach.

The approach presented and illustrated in this work adapts and implements the multi-layered LogiKEy knowledge engineering methodology [13] to enable the application of off-the-shelf interactive and automated theorem proving technology for classical higher-order logic in ethical-legal reasoning. LogiKEy has been extended in this work to include an additional modeling layer, the value ontology. The value ontology forms a bridge between the legal and general world knowledge layer and the object logic layer in LogiKEy. Isabelle/HOL has proven to be an excellent base technology to support the presented formalization work and the conducted experiments. We are particularly pleased with the good performance of the *Nitpick* model finder and the integrated automated theorem provers (provided by *Sledgehammer*), which provided very useful feedback at all modeling layers, including fully automated proofs for formal justification of the discussed judgments.

Further work includes refining the modelling of Lomfeld's value theory in combination with providing more expressive (combinations of) object logics. With respect to the latter, the use of material implication to model defeasible or "default" rules (among others) has proven sufficient for the illustrative purposes of this paper, but it is important to note that more realistic modeling of legal cases must also provide mechanisms to deal with the inevitable emergence of conflicts and contradictions in normative reasoning (overruling, conflict resolution, etc.). In line with the LogiKEy approach, we can indeed introduce a defeasible conditional operator by reusing the modal operators of $\mathcal{PL}$(as discussed, e.g., in [20, 28]), or alternatively by the SSE of a suitable conditional logic in HOL [6]. Various kinds of paraconsistent negations could also be considered for the non-explosive representation of (and recovery from) contradictions by purely object-logical means (cf. [21] for an appropriate SSE). It is the pluralistic nature of our approach, realised within a dynamic modelling framework, that enables and supports such improvements without requiring technical adjustments to the underlying base reasoning technology.

References

1 Isabelle/HOL sources for this formalisation work. http://logikey.org, 2021. Subfolder: Preference-Logics/EncodingLegalBalancing.

2 Trevor J. M. Bench-Capon. The missing link revisited: The role of teleology in representing legal argument. *Artificial Intelligence and Law*, 10(1-3):79–94, 2002.

3 Trevor J. M. Bench-Capon. HYPO's legacy: introduction to the virtual special issue. *Artificial Intelligence and Law*, 25(2):205–250, 2017.

4 Trevor J. M. Bench-Capon. The need for good old-fashioned AI and Law. In W. Hötzendorfer, C. Tschol, and F. Kummer, editors, *In International Trends in Legal Informatics: A Festschrift for Erich Schweighofer*. Weblaw AG, 2020.

5 Trevor J. M. Bench-Capon and Giovanni Sartor. A model of legal reasoning with cases incorporating theories and value. *Artificial Intelligence*, 150:97–143, 2003.

6 Christoph Benzmüller. Cut-elimination for quantified conditional logic. *Journal of Philosophical Logic*, 46(3):333–353, 2017. doi:10.1007/s10992-016-9403-0.

7 Christoph Benzmüller. Universal (meta-)logical reasoning: Recent successes. *Science of Computer Programming*, 172:48–62, 2019. doi:10.1016/j.scico.2018.10.008.

8 Christoph Benzmüller, Ali Farjami, Paul Meder, and Xavier Parent. I/O logic in HOL. *Journal of Applied Logics – IfCoLoG Journal of Logics and their Applications (Special Issue: Reasoning for Legal AI)*, 6(5):715–732, 2019.

9 Christoph Benzmüller, Ali Farjami, and Xavier Parent. Åqvist's dyadic deontic logic E in HOL. *Journal of Applied Logics – IfCoLoG Journal of Logics and their Applications (Special Issue: Reasoning for Legal AI)*, 6(5):733–755, 2019.

10 Christoph Benzmüller, Ali Farjami, and Xavier Parent. Dyadic deontic logic in hol: Faithful embedding and meta-theoretical experiments. In Matthias Armgardt, Hans Christian Nordtveit Kvernenes, and Shahid Rahman, editors, *New Developments in Legal Reasoning and Logic: From Ancient Law to Modern Legal Systems*, volume 23 of *Logic, Argumentation & Reasoning*. Springer Nature Switzerland AG, 2021. doi:10.1007/978-3-030-70084-3.

11 Christoph Benzmüller, David Fuenmayor, and Bertram Lomfeld. Encoding legal balancing: Automating an abstract ethico-legal value ontology in preference logic, 2020. Workshop on Models of Legal Reasoning (MLR 2020), hosted by 17th Conference on Principles of Knowledge Representation and Reasoning (KR 2020). Unpublished paper available at: https://www.researchgate.net/publication/342380027.

12 Christoph Benzmüller and Bertram Lomfeld. Reasonable machines: A research manifesto. In Ute Schmid, Franziska Klügl, and Diedrich Wolter, editors, *KI 2020: Advances in Artificial Intelligence – 43rd German Conference on Artificial Intelligence, Bamberg, Germany, September 21–25, 2020, Proceedings*, volume 12352 of *Lecture Notes in Artificial Intelligence*, pages 251–258. Springer, Cham, 2020. doi:10.1007/978-3-030-58285-2_20.

13 Christoph Benzmüller, Xavier Parent, and Leendert van der Torre. Designing normative theories for ethical and legal reasoning: LogiKEy framework, methodology, and tool support. *Artificial Intelligence*, 287:103348, 2020. doi:10.1016/j.artint.2020.103348.

14 Christoph Benzmüller and Lawrence C. Paulson. Multimodal and intuitionistic logics in simple type theory. *The Logic Journal of the IGPL*, 18(6):881–892, 2010. doi:10.1093/jigpal/jzp080.

15 Christoph Benzmüller and Lawrence C. Paulson. Quantified multimodal logics in simple type theory. *Logica Universalis (Special Issue on Multimodal Logics)*, 7(1):7–20, 2013. `doi:10.1007/s11787-012-0052-y`.

16 Donald Berman and Carole Hafner. Representing teleological structure in case-based legal reasoning: the missing link. In *Proceedings 4th ICAIL*, pages 50–59. New York: ACM Press, 1993.

17 Jasmin C. Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending Sledgehammer with SMT solvers. *Journal of Automated Reasoning*, 51(1):109–128, 2013.

18 Jasmin C. Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *Journal of Formalized Reasoning*, 9(1):101–148, 2016.

19 Jasmin C. 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, *ITP 2010*, volume 6172 of *LNCS*, pages 131–146. Springer, 2010.

20 Craig Boutilier. Toward a logic for qualitative decision theory. In *Principles of knowledge representation and reasoning*, pages 75–86. Elsevier, 1994. `doi:10.1016/B978-1-4832-1452-8.50104-4`.

21 David Fuenmayor. Topological semantics for paraconsistent and paracomplete logics. *Archive of Formal Proofs*, 2020. , Formal proof development. URL: `https://isa-afp.org/entries/Topological_Semantics.html`.

22 Bernhard Ganter and Rudolf Wille. *Formal concept analysis: mathematical foundations*. Springer Berlin, 2012.

23 Thomas F. Gordon and Douglas Walton. Pierson vs. Post revisited. *Frontiers in Artificial Intelligence and Applications*, 144:208, 2006.

24 Joseph Y. Halpern. Defining relative likelihood in partially-ordered preferential structures. *Journal of Artificial Intelligence Research*, 7:1–24, 1997.

25 Robert Koons. Defeasible Reasoning. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2017 edition, 2017.

26 P. Krause, S. Ambler, Elvang-Goransson, M., and J. Fox. A logic of argumentation for reasoning under uncertainty. *Computational Intelligence*, 1995. `doi:10.1111/j.1467-8640.1995.tb00025.x`.

27 Fenrong Liu. *Changing for the better: Preference dynamics and agent diversity*. PhD thesis, Institute for Logic, Language and Computation, Universiteit van Amsterdam, 2008.

28 Fenrong Liu. *Reasoning about Preference Dynamics*. Springer Netherlands, 2011. `doi:10.1007/978-94-007-1344-4`.

29 Bertram Lomfeld. *Die Gründe des Vertrages: Eine Diskurstheorie der Vertragsrechte*. Mohr Siebeck, Tübingen, 2015.

30 Bertram Lomfeld. Grammatik der Rechtfertigung: Eine kritische Rekonstruktion der Rechts(fort)bildung. *Kritische Justiz*, 52(4), 2019.

31 Juliano Maranhão and Giovanni Sartor. Value assessment and revision in legal interpretation. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Law, ICAIL 2019, Montreal, QC, Canada, June 17-21, 2019*, pages 219–223, 2019. `doi:10.1145/3322640.3326709`.

32 L. Thorne McCarty. An implementation of Eisner v. Macomber. In *Proceedings of the 5th International Conference on Artificial Intelligence and Law*, pages 276–286, 1995.

33 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

34 Henry Prakken. An exercise in formalising teleological case-based reasoning. *Artificial Intelligence and Law*, 10(1-3):113–133, 2002.

35 Henry Prakken and Giovanni Sartor. Law and logic: A review from an argumentation perspective. *Artificial Intelligence*, 227:214–225, 2015.

36 Edwina L. Rissland and Kevin D. Ashley. A case-based system for trade secrets law. In *Proceedings of the 1st international conference on Artificial Intelligence and Law*, pages 60–66, 1987.
37 Johan van Benthem, Patrick Girard, and Olivier Roy. Everything else being equal: A modal logic for *ceteris paribus* preferences. *J. Philos. Log.*, 38(1):83–125, 2009. doi:10.1007/s10992-008-9085-3.
38 Georg Henrik von Wright. *The logic of preference*. Edinburgh University Press, 1963.
39 Makarius Wenzel. Isabelle/Isar—a generic framework for human-readable proof documents. *From Insight to Proof-Festschrift in Honour of Andrzej Trybulec*, 10(23):277–298, 2007.

Unsolvability of the Quintic Formalized in Dependent Type Theory

Sophie Bernard
Université Côte d'Azur, Inria, Sophia Antipolis, France

Cyril Cohen
Université Côte d'Azur, Inria, Sophia Antipolis, France

Assia Mahboubi
Inria, Nantes, France
Vrije Universiteit Amsterdam, The Netherlands

Pierre-Yves Strub
École Polytechnique, Palaiseau, France

Abstract

In this paper, we describe an axiom-free Coq formalization that there does not exists a general method for solving by radicals polynomial equations of degree greater than 4. This development includes a proof of Galois' Theorem of the equivalence between solvable extensions and extensions solvable by radicals. The unsolvability of the general quintic follows from applying this theorem to a well chosen polynomial with unsolvable Galois group.

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

Keywords and phrases Galois theory, Coq, Mathematical Components, Dependent Type Theory, Abel-Ruffini, General quintic

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.8

Supplementary Material *Software (Code)*: `https://github.com/math-comp/Abel/tree/1.1.2`
archived at `swh:1:dir:faac74a95b668cc8c77c9904bd27a50df236efc0`

1 Introduction

This article presents a formal study of the existence of solutions by radicals of polynomial equations. Solutions by radicals are the ones that can be expressed from the coefficients of a polynomial using operations of addition, multiplication, subtraction, division, and extraction of roots. More precisely we study the case of polynomial equations of degree greater than 4. As opposed to the case of lower degree, there is no solution by radicals to general polynomial equations of degree five or higher with arbitrary coefficients. This theorem, also known as the Abel-Ruffini theorem, is attributed to Abel for his work [14, volume 1, chapter III] published in 1826. Ruffini is credited for a first formulation and proof [21] from 1799. Abel writes about Ruffini: "[...]; but his memoir is so complicated that it is very hard to assess the correctness of his reasoning. It seems to me that his reasoning is not always satisfactory." [14, volume 2, chapter XVIII]

In fact, we developed a formal proof of the more general theorem – attributed to Galois [8] in his memoir from 1830 – which provides an explicit necessary and sufficient condition for the existence of solutions by radical, and we also formalize an example of non-solvable quintic, obtained as a corollary of the latter. This Galois theorem is an emblematic result of Galois theory, which studies field extensions of commutative fields via a correspondence with groups of permutations of roots of polynomials.

This formalization endeavor builds on an existing library covering elementary results in Galois theory, developed by Georges Gonthier and Russell O'Connor in the Mathematical Components library [25], for the purpose of the formal proof of the Odd Order theorem [11]. As there is no published description of this material, we provide where needed a description of the material from this contribution that we rely on.

The formalized proof is constructive, and relies on nothing but the axioms and rules of the foundational framework implemented by Coq. The code of this formalization is available on `https://github.com/math-comp/Abel` version 1.1.2. Every numbered definition, lemma or theorem in this paper is our contribution, and we hyperlinked red underlined definitions.

2 Background and outline

Throughout this section, we consider a field K of characteristic 0 and a polynomial $P \in K[X]$. We study the solvability by radicals of the equation $P(X) = 0$, also termed the solvability by radicals of P. An easy case is when all the roots of P are in K, i.e., when F *splits* P. In the general case, the idea is to consider successive *field extensions* F over K, i.e., fields F such that $K \subset F$. These extensions are built so as to gradually encompass all the roots of P.

In the rest of the paper, we write F/K to denote that F is a field extension over K. Given such an extension, the larger field F is a K-vector space and we can consider its dimension – called the *degree of the extension* and written $[F : K]$. A field extension is said to be *finite* when its degree is finite. In the present paper, all the field extensions under consideration are finite and we sometimes simply refer to them as "field extensions". If $x_0, \ldots, x_n$ are elements of F, we denote by $K(x_1, \ldots, x_n)$ the smallest field which contains K and x_i for all $i \leq n$. Note that both $K(x_1, \ldots, x_n)/K$ and $F/K(x_1, \ldots, x_n)$ are field extensions. The *splitting field* of $P \in F[X]$ is the smallest field extension of F which splits P.

Let F/K be a field extension. An element x of F is said to be *algebraic over* K if it is a root of some nonzero polynomial with coefficients in K. The field extension F/K is called *algebraic* when all its elements are algebraic over K. Moreover if F is a splitting field for some polynomial in $K[X]$, the extension F/K is said to be *normal*. Last, the *minimal polynomial* of an element x of F is the monic polynomial of minimal degree among all the nonzero polynomials with coefficients in K and having x as a root.

▶ **Definition 1** (radical, solvable by radicals)**.** *Let F/K be a field extension. F/K is called a* simple radical extension *if there exists $x \in F$ and a positive integer $n \in \mathbb{N}^*$ such that $x^n \in K$ and $F = K(x)$. A* radical series *is a tower $F_0 \subset \cdots \subset F_n$ where F_k/F_{k-1} is a simple radical extension for $k \in \{1, \ldots, n\}$. A field extension F/K is a* radical extension *if there is a radical series $K = F_0 \subset \cdots \subset F_n = F$. It is a* solvable by radicals extension *if there is a radical extension E/K such that $F \subset E$.*

A polynomial $P \in K[X]$ is solvable by radicals *if it splits in a radical extension of K.*

The crux of the method is, given a splitting field F of a polynomial P over K, to study the field automorphisms of F that fix K point-wise, thereby permuting the roots P.

More generally, given a field (finite) extension F/K, the set of automorphisms of F that fix K point-wise is always a group. We call it $\mathrm{Gal}(F/K)$, the *Galois group* of the extension F/K. Moreover, if $\mathrm{Gal}(F/K)$ fixes *exactly* K, the extension F/K is then said to be a *Galois extension*. In this case, the order of the Galois group $\mathrm{Gal}(F/K)$ is equal to the degree of the extension $[F : K]$. Some properties of $\mathrm{Gal}(F/K)$ hold without F/K being Galois, e.g., the inclusion $\mathrm{Gal}(F/M) \subset \mathrm{Gal}(F/K)$ when $K \subset M$. Every Galois extension is a normal extension and since we assumed K has characteristic zero, every normal extension F/K is a Galois extension.

The first theorem that has been formally proven in this paper states that the Galois group of a polynomial P contains all the information about the solvability of the corresponding polynomial equation:

▶ **Theorem 2** (Galois). *A polynomial $P \in F[X]$ is solvable by radicals if and only if its Galois group is solvable.*

We recall that a group G is *solvable* if it is close to being abelian, in the sense that there exists a normal series $\{e\} = G_0 \lhd \cdots \lhd G_n = G$ of G, whose factors G_{k+1}/G_k are all abelian.

Proof. Lemma 11 from Section 4 addresses the right to left direction. Lemma 19 from Section 5 shows the converse direction. Section 6.1 proves the theorem for $F = \mathbb{Q}$. Finally Section 8.3 explains how to generalize this both in constructive and classical logic contexts. ◀

In other words, Theorem 2 reduces the problem of the solvability by radicals of a polynomial to the analysis of the solvability of its Galois group and allows us to deduce the following one:

▶ **Theorem 3** (Abel-Ruffini). *There is no solution by radicals to general polynomial equations of degree five or higher.*

Proof. It suffices to show that there is a polynomial over $\mathbb{Q}$ which is not solvable by radicals because otherwise the general solution would apply. Theorem 22 in Section 6.3 shows that the polynomial $X^5 - 4X + 2$ is not solvable by radicals. ◀

For the sake of clarity, and unless otherwise stated, in the rest of the paper we focus on the specific case where the base field K has characteristic zero. For instance, the base field of Theorem 3 is simply $\mathbb{Q}$, the field of rational numbers. However, in the formal development, we have striven to provide definitions that are general enough to also apply to the positive characteristic case. Typically, in the case of nonzero characteristic, a normal (hence algebraic) extension F/E is Galois only in the case where it is also *separable* – i.e. if for any $x \in F$, the minimal polynomial of x is separable, i.e., has only simple roots. A substantial amount of our formal development thus applies to the case of positive characteristic as well. We discuss this more in details in Section 9.

3 Formal definitions

Throughout this paper, and unless explicitly mentioned, we consider a (finite) field extension L/F_0, which will serve as an ambient larger locus, fixing a common type for the elements of the various fields at stake. As discussed in Section 2, the reader can safely assume that L has characteristic zero.

In fact, we also assume this extension to be *normal*, that is, that L is the splitting field of a certain polynomial in $F_0[X]$. We will thus use letters E, F, K for sub-fields of L that are themselves extensions over F_0. This formalization choice can be compared to the use of an ambient `finGroupType` in the formalization of finite group theory [11, 16].

In Coq, these assumptions amount to opening a section sharing variables `F0` and `L`, as well as implicit type declarations for letters `E,F,K`:

```
Variables (F0 : fieldType) (L : splittingFieldType F0).
Implicit Types (E F K : {subfield L}).
```

Considering a normal ambient field extension L/F_0 ensures, without loss of generality, that the ambient L is large enough so that for each subfield E of L, it is possible to find a Galois extension F/E, where F is a subfield of L.

Of course, when F/E is itself a field extension, it remains possible to see F as a vector space over E: for instance `\dim_E F` refers to the dimension of F as a vector space over E, i.e., to the degree $[F : E]$ of the extension. Note that as a rule of thumb, notations are designed so as to be well-formed as often as possible. For example, `\dim_E F` is actually defined as the Euclidean quotient of $[F : F_0]$ by $[E : F_0]$, and thus does not require `E` to be included in `F`. These formalization choices, inherited from the design of the Mathematical Components library for linear algebra [10], significantly contribute to reduce the bureaucratic workload in proofs.

In this work, we benefit from the formalized basic concepts and results in Galois theory available in the Mathematical Components library [11], notably from the available proof of the fundamental theorem of Galois theory. The corresponding libraries actually introduce the vocabulary related to field extensions and Galois groups. In particular, `'Gal(F/E)` refers to the Galois group of a field extension F/E. Here as well, this notation is well formed for any `E,F : {subfield L}`, regardless of any inclusion property, and actually refers to $\mathrm{Gal}(F/E \cap F)$ and is a group, regardless of whether F/E is a Galois extension.

We lack space to further comment on all the Coq definitions involved in the present formal proof, but we provide in Figure 1 a correspondence table between the Mathematical Components syntax and the related mathematical objects.

4 From solvable Galois groups to solvable extensions

In this section, we consider E and F two sub-fields of an ambient common normal extension L and we study sufficient conditions for the field extension F/E to be solvable by radical. As these conditions may involve assumptions of primitive roots of unity, we thus enrich the formal context given in Section 3 with the following declarations:

```
Implicit Types (w : L) (n : nat).
```

First, we prove the result in the case of an abelian Galois extension, that is, a Galois extension which Galois group is abelian. In this case, we can prove that the extension is radical.

▶ **Lemma 4.** *An abelian Galois extension F/E of degree n is radical as soon as E contains a primitive n^{th} root of unity.*

```
Lemma abelian_radical_ext w E F (n := \dim_E F) : n.-primitive_root w →
  w \in E → galois E F → abelian 'Gal(F / E) → radical.-ext E F.
```

Proof. The proof goes by exhibiting a basis (r_i) of F, seen as a vector space over E, such that for any u in $G = \mathrm{Gal}(F/E)$ and for any $i \in \{1, \ldots, n\}$, $u(r_i) = \lambda r_i$, where λ is some n^{th} root of unity. Indeed, as in this case $u(r_i^n) = r_i^n$, we have $r_i^n \in E$ for any i, which concludes the proof.

Let u be an element of G. Since $|G| = [F : E] = n$, by Lagrange's theorem of finite group theory, we have $u^n = \mathrm{id}$. Therefore the minimal polynomial of u in E divides the polynomial $X^n - 1$. But since the latter is square-free and splits over E (for E contains a primitive n^{th} root of unity), so is the minimal polynomial of u, and u is thus diagonalizable. Moreover,

`R : ringType`	R is a ring, whose elements are the terms `x : R`
`p %= q`	the polynomials P and Q are equal up to a unit of R
`'X`	$X \in R[X]$ the indeterminate
`x *: p`	the polynomial xP with $x \in R$ and $P \in R[X]$
`x%:P`	the constant polynomial $x \in R[X]$
`p ^^ f`	the image of $P \in R[X]$ by a ring morphism $f : R \to R'$
`F0 : fieldType`	F_0 is a field, whose elements are the terms `x : L`
`prime n`	the natural number $n \in \mathbb{N}$ is prime
`n != 0 :> F0`	n is nonzero in F_0
`has_char0 F0`	F_0 has characteristic 0
`n.-primitive_root w`	ω is a primitive n^{th} root of unity
	(we use the ASCII character `w` for the greek letter ω)
`x : L`	x is an element of the field L
`E, F, K : {subfield L}`	E, F, K are subfields of L, with base field F_0
`\dim_E F`	the dimension of F over E, i.e., the degree $[F : E]$
`x \in E`	x is in the subset E of L
`E ≤ F`	$E \subset F$, i.e., E is a subfield of F
`1 : {subfield L}`	F_0, seen as a subfield of L
`{:L} : {subfield L}`	L, seen as a subfield of L
	by definition we always have `x \in {: L}` for `x : L`
`<<E ; x>> : {subfield L}`	$E(x)$, the smallest field generated by E and $x \in L$
`<<E & s>> : {subfield L}`	$E(s)$, the smallest field generated by E and the sequence s
`E :&: F : {subfield L}`	$E \cap F$, the field $\{x \mid x \in E \wedge x \in F\}$
`E * F : {subfield L}`	the compositum EF, the field $\{xy \mid x \in E, y \in F\}$
`iota : 'AHom(L,L')`	$\iota : L \to L'$ is an F_0-algebra morphism
`iota @: E`	$\iota(E)$, the image of E by ι, a subfield of L'
`splittingFieldFor E p F`	$F = E(\vec{x})$ where $p \in L[X]$ has roots $\vec{x}$ and coefficients in E
`L : splittingFieldType F0`	L is a splitting field extension of the field F_0, as a type;
	this is equivalent to the existence of `p` with coefficients
	in `L`, such that `splittingFieldFor 1 p {:L}`
`minPoly E x : {poly L}`	the minimal polynomial of x over E
`normalField E F : {subfield L}`	the subfield extension F/E is normal
`separable E F : {subfield L}`	the subfield extension F/E is separable
`galois E F : {subfield L}`	the subfield extension F/E is Galois
`radical E x n`	$x^n \in E$ with $n > 0$, i.e., the element x is radical in E
`pradical E x p`	$x^p \in E$ and p is prime
`r.-ext E F`	F/E is `r`, where `r` is either `radical` or `pradical`
`solvable_by r E F`	F/E is solvable by `r`, where `r` is either `radical` or `pradical`
`'Gal(F/E)`	the Galois group of the subfield extension F/E
`phi @* G`	the image of the group G by the morphism φ
`abelian G`	G is abelian
`solvable G`	G is solvable

Figure 1 Correspondence between Coq syntax and mathematical vocabulary.

since G is abelian, all its elements are co-diagonalizable. As a consequence, there exists a common basis (r_i) of eigenvectors for all elements of G, i.e., a basis (r_i) such that for all u in G, $u(r_i) = \lambda r_i$ for some eigenvalue λ in E. Since these eigenvalues are roots of the minimal polynomial $X^n - 1$ of u, we have $\lambda^n = 1$. ◀

Lemma 4 illustrates the role of linear algebra in Galois theory. However, at the start of this project, the corresponding chapter, about standard results on the diagonalization of matrices, was completely missing from the Mathematical Components library. Formalizing this chapter is one of the spin-off contributions of the present work.

The next step is to generalize the result to the case of a *solvable* Galois group: in this case the corresponding field extension is called a solvable extension. The proof goes by applying Lemma 4 to each of the (abelian) quotients involved in the corresponding normal series, and concludes by gluing radical extensions.

▶ **Lemma 5.** *A solvable Galois extension F/E of degree n is radical, as soon as E contains a primitive n^{th} root of unity.*

```
Lemma solvableWradical_ext w E F (n := \dim_E F) : n.-primitive_root w →
  w \in E → galois E F → solvable 'Gal(F / E) → radical.-ext E F.
```

Proof. We proceed by strong induction on n, the degree of the field extension. Let F/E be a Galois extension of degree n, and suppose that its Galois group G is solvable. If $n = 1$, the extension is trivial, hence G is solvable. Otherwise, by definition, G has a normal and solvable subgroup H of prime index. In particular $H \neq G$ and the quotient G/H is abelian. Let F^H be the field fixed by H (point-wise). Then, the extension F/F^H is Galois and solvable, of degree strictly smaller than n, and F^H/E is an abelian Galois extension. We conclude that F/E is radical by combining the induction hypothesis with Lemma 4. ◀

The main ingredient in the proof of Lemma 5 is the properties of the field extensions F/F^H and F^H/E. These were obtained from the theory of F^H, for H subgroup of a Galois group, already present in the Mathematical Components library.

We can relax the hypothesis that E should contain the n^{th} roots of unity, and transfer it to the ambient field, to the price of weakening the conclusion: in this case, F is only solvable by radicals. This crux of the proof relies on the properties of the Galois group of a compositum extension, which were not present in the Mathematical Components library. In particular, we use the following fact:

▶ **Lemma 6.** *Let E/K be a Galois extension and F a sub-field of E. Then:*

$$\mathrm{Gal}(KF/F) \simeq \mathrm{Gal}(K/K \cap F)$$

```
Lemma galois_isog (k K F : {subfield L}) : galois k K → k ≤ F →
  'Gal((K * F) / F) \isog 'Gal (K / K :&: F)
```

Proof. See for instance Lang's proof [15, VI, §1, Theorem 1.12]. ◀

▶ **Lemma 7.** *A solvable Galois extension F/E of degree n is solvable by radicals, as soon as E and F are sub-fields of a common normal extension L, which contains a primitive n^{th} root of unity in L.*

```
Lemma galois_solvable_by_radical w E F (n := \dim_E F) : n.-primitive_root w →
  galois E F → solvable 'Gal(F / E) → solvable_by radical E F.
```

Proof. Let F/E a Galois extension of degree n, with E, F sub-fields of F. Let $\omega \in L$ be a primitive n^{th} root of unity. The proof goes by showing that the extension $FE(\omega)/E$ is radical. Since $E(\omega)/E$ is a simple radical extension, it suffices to show $FE(\omega)/E(\omega)$ is radical.

Since F/E is a Galois extension, then so is $FE(\omega)/E(\omega)$. Let m be the degree of $FE(\omega)/E(\omega)$. By Lemma 6, $\mathrm{Gal}(FE(\omega)/E(\omega))$ is isomorphic to $\mathrm{Gal}(F/F \cap E(\omega))$, which is thus of order m as well. But since $\mathrm{Gal}(F/F \cap E(\omega))$ is a subgroup of $\mathrm{Gal}(F/E)$, its order m divides n, the order of $\mathrm{Gal}(F/E)$. Consider $\omega' = \omega^{\frac{n}{m}}$. It is an element of $E(\omega)$, and thus of $FE(\omega)$, and a primitive root of unity. We can apply Lemma 5 on the extension $FE(\omega)/E(\omega)$, and the m^{th} primitive root of unity ω' as soon as we show that $\mathrm{Gal}(FE(\omega)/E(\omega))$ is solvable. Which is the case because it is isomorphic to $\mathrm{Gal}(F/F \cap E(\omega))$, itself solvable as a subgroup of $\mathrm{Gal}(F/E)$. ◀

The final result of the section trades the assumption on the solvability of the Galois group for the solvability of the extension itself, i.e., for the solvability of the Galois group of the extension by the normal closure.

▶ **Definition 8.** *The* normal closure $\mathrm{NCl}_E(F)/E$ *of* F/E *is the smallest (for field inclusion) field extension of* F *that is normal over* E*.*

▶ **Definition 9.** *An extension* F/E *is* solvable *if* F/E *(is separable) and* $\mathrm{Gal}(\mathrm{NCl}_E(F)/E)$ *is solvable.*

▶ Remark 10. Note that in the case of zero characteristic, the separability requirement vanishes. A Galois extension F/E is solvable if and only if $\mathrm{Gal}(F/E)$ is solvable (as a group).

By definition of the normal closure, if an extension F/E is solvable, then $\mathrm{NCl}_E(F)/E$ is Galois. Therefore, solvability by radicals follows from the solvability of an extension, as an immediate corollary of Lemma 7.

▶ **Lemma 11.** *Let* F/E *be a solvable extension, and* n *the degree of the extension* $\mathrm{NCl}_E(F)/E$*.* F/E *is solvable by radicals as soon as* L *contains a primitive* n^{th} *root of unity.*

```
Lemma ext_solvable_by_radical w E F (n := \dim_E (normalClosure E F)) :
  n.-primitive_root w → solvable_ext E F → solvable_by radical E F.
```

Proof. Since F/E is solvable, $\mathrm{Gal}(\mathrm{NCl}_E(F)/E)$ is solvable. Thus Lemma 7 applies and proves that $\mathrm{NCl}_E(F)/E$ is solvable by radicals. Since $F \subset \mathrm{NCl}_E(F)$, then F/E is solvable by radical as well. ◀

5 From solvable by radicals extensions to solvable extensions

Recall that L/F_0 is an ambient normal field extension. We first establish two useful results on simple radical extensions $E(x)/E$ for E a sub-field of L. When x is a root of unity, the extension $E(x)/E$ is called a *cyclotomic* extension. A cyclotomic extension is a Galois and solvable extension.

▶ **Lemma 12.** *Suppose that L contains ω, an n^{th} primitive root of unity for n a positive integer. Consider E a sub-field of L and $x \in L$ such that $x^n \in E$. Then, the extension $E(\omega, x)/E$ is Galois. In particular if $\omega \in E$, then $E(x)/E$ is Galois.*

```
Lemma galois_cyclo_radical (n : nat) (w x : L) (E : {subfield L}):
    p.-primitive_root w → p > 0 → x ^+ p \in E → galois E << <<E; w>> ; x >>.
```

Proof. If $x \in E$, the conclusion is immediate. We can thus suppose that $x \neq 0$ and $n > 1$. In this case, the polynomial $P = X^p - x^n \in E[X]$ is separable, since it has n distinct roots, of the form $x\omega^i$, for $i = 0 \ldots n-1$. Moreover,

$$\begin{aligned} E(x, x\omega, \ldots, x\omega^{n-1}) &= E(x, x\omega)(x\omega^2, \ldots, x\omega^{n-1}) && \text{since } n > 0 \\ &= E(\omega, x)(x\omega^2, \ldots, x\omega^{n-1}) && \text{since } x \neq 0 \\ &= E(\omega, x) && \text{since } x\omega^i \in E(\omega, x) \end{aligned}$$

It follows that $E(\omega, x)$ is a splitting field of P, and therefore that $E(\omega, x)/E$ is Galois. ◀

▶ **Lemma 13.** *Suppose that L contains ω, a p^{th} primitive root of unity for p a prime number. Consider E a subfield of L and $x \in L$ such that $x^p \in E$, but $x \notin E$. Then, the minimal polynomial of x over E is $X^p - x^p$.*

As a consequence, Gal$(E(x)/E)$ *is of prime order and is thus cyclic, hence abelian (and solvable).*

```
Lemma minPoly_pradical (p : nat) (w x : L) (E : {subfield L}):
    p.-primitive_root w → prime p → w \in E → x \notin E → x ^+ p \in E →
  minPoly E x = 'X^p - (x ^+ p)%:P.
```

Proof. Let $P \in E[X]$ be the minimal polynomial of x over E. By minimality, P divides any polynomial over E that cancels x. In particular, P divides $X^p - x^p = \prod_{i<p}(X - x\omega^i)$. Hence there is a subset S of $I_p = \{i \mid i < p\}$ such that $P = \prod_{i \in S}(X - x\omega^i)$. Since P cancels x, S contains x, therefore $|S|$ is positive. It suffices to show $|S| \geq p$, because then $S = I_p$ and $P = X^p - x^p$. Since $|S|$ is positive, it is sufficient to prove that p divides $|S|$.

First, note that p divides any k such that $x^k \in E$. Indeed, if k and p were coprime, Bézout's identity would provide $m, n \in \mathbb{Z}$ such that $km + pn = 1$. As a consequence, we would have $x = (x^k)^m + (x^p)^n \in E$, contradicting our assumption that $x \notin E$.

Now the constant coefficient of P is $x^{|S|}\Omega$, where $\Omega = \prod_{i \in S} \omega^i$ is a nonzero element of E, hence $x^{|S|} \in E$ and p divides $|S|$. ◀

In order to get rid of the assumption that the ambient L contains a suitable root of the unity, we prove that the normal closure of a subfield of L, as well as the Galois group of an extension in L, are preserved up to isomorphism when L is extended with some roots of unity.

Consider L/F_0 and L'/F_0 two normal extensions, $\iota : L \to L'$ an F_0-algebra morphism, and F/E a field extension in L.

▶ **Lemma 14.** *There is a group isomorphism* $\mathrm{Gal}(F/E) \to \mathrm{Gal}(\iota(F)/\iota(E))$*, which we also denote ι.*

In Coq the group (iso)morphism corresponding to ι is called `map_gal`.

```
Lemma map_gal_inj : 'injm (map_gal iota).
Lemma img_map_gal : map_gal iota @* 'Gal(F / E) = 'Gal(iota @: F / iota @: E).
```

The properties of this morphism are key to the preservation of normal extensions, separable extensions, Galois extensions, normal closures and solvable extensions under the associated algebra isomorphism.

▶ **Lemma 15.** *The extension $\iota(F)/\iota(E)$ is normal (resp. separable, Galois, solvable) if and only if F/E is normal (resp. separable, Galois, solvable), and $\iota(\mathrm{NCl}_E(F)) = \mathrm{NCl}_{\iota(E)}(\iota(F))$.*

```
Lemma normalField_aimg    : normalField (iota @: E) (iota @: F)  = normalField E F.
Lemma separable_aimg      : separable (iota @: E) (iota @: F)    = separable E F.
Lemma galois_aimg         : galois (iota @: E) (iota @: F)       = galois E F.
Lemma solvable_ext_aimg   : solvable_ext (iota @: E) (iota @: F) = solvable_ext E F.
Lemma aimg_normalClosure :
  iota @: normalClosure E F = normalClosure (iota @: E) (iota @: F).
```

The combination of Lemma 15 with Lemma 16 makes possible to extend, if needed, the ambient field with a primitive root of unity so as to prove that a certain extension is normal (resp. separable, Galois, or solvable).

▶ **Lemma 16.** *Let L/F_0 be an ambient normal field extension and n a natural number coprime with the characteristic of F_0. There is an ambient normal field extension L'/F_0, a primitive n^{th} root of unity $\omega \in L'$ and an F_0-algebra morphism $\iota : L \to L'$, such that $\iota(L)(\omega) = L'$.*

We can now state and prove properties of simple (prime) radical extensions which do not require any assumption on the presence of a root of unity.

▶ **Lemma 17.** *Let p be prime number such that $p \neq 0$ in F_0. Let $x \in L$ and E a subfield of L such that $x^p \in E$. The extension $E(x)/E$ is solvable.*

Note that because of the definition of a solvable extension, $E(x)/E$ need not be Galois.

```
Lemma pradical_solvable_ext (p : nat) (x : L) (E : {subfield L}) :
  prime p → p != 0 :> F0 → x ^+ p \in E → solvable_ext E <<E; x>>.
```

Proof. Without loss of generality, we can assume the existence of $\omega \in L$ a primitive p^{th} root of unity. Indeed, Lemma 16 gives the existence of a field extension L' and an embedding $\iota : L \to L'$, where L' contains a p^{th} primitive root of unity (since $p \neq 0$ in F_0). Now we may prove $\iota(E(x))/\iota(E)$ is Galois and "transfer" the result to $E(x)/E$ using Lemma 15.

In order to prove $E(x)/E$ is solvable, it suffices to find a Galois extension of $E(x)$ that is solvable. Because of Lemma 12, $E(\omega, x)/E$ is Galois. Now both $E(\omega)/E$ (because it is cyclotomic) and $E(\omega, x)/E(\omega)$ (by Lemma 13) are Galois and solvable. Hence, $\mathrm{Gal}(E(\omega, x)/E(\omega))$ is a normal subgroup of $\mathrm{Gal}(E(\omega)/E)$, therefore $E(\omega, x)/E$ is solvable. ◀

The final lemma of this section is often stated in the literature in the following way: "If F/E is Galois and solvable by radicals then $\mathrm{Gal}(F/E)$ is solvable". While this is true, this does not allow for a proof by induction as such since intermediate extensions of the radical series of F/E need not be Galois over E. Rigorous proofs must strengthen the induction. One way to do so is by introducing the notion of solvable extension which, contrarily to the notion of Galois extension, is transitive:

▶ **Lemma 18** (solvability of extensions is transitive). *If F/E and K/F are solvable extensions, then K/E is solvable.*

Proof. We essentially follow the proof from Lang [15, VI, §7, Proposition 7.1], except that instead of in-lining the definition of the normal closure in a particular case, we define and study normal closures for their own interest, which eventually results in a shorter proof. ◀

We are now ready to state the final and main result of this section, and to avoid assuming that the extension F/E is Galois, in addition to being solvable by radicals. The proof is a straightforward induction on the height of the radical series.

▶ **Lemma 19.** *If F/E is solvable by radicals then F/E is a solvable extension.*

```
Lemma radical_ext_solvable_ext (E F : {subfield L}) : has_char0 L → E ≤ F →
  solvable_by radical E F → solvable_ext E F.
```

Proof. Let F/E be a solvable by radicals extension, it is also solvable by prime radicals, so there exists a prime radical extension tower $E = E_0 \subset E_1 \subset \ldots \subset E_n$ such that $F \subset E_n$. Since every intermediate extension E_{i+1}/E_i is solvable, by Lemma 17, we conclude by induction and by Lemma 18 that E_n/E_0 is solvable. Since $F \subset E_n$, F/E is also solvable. ◀

Note that this proof goes by induction on the length of the tower. Curiously, some references (such as the French wikipedia page on the Abel-Ruffini Theorem as of 2021-04-20) do not rely on solvable extensions, or define it as "being Galois and solvable" instead of "having a Galois field extension that is solvable". Unfortunately, under such variations, we lack a transitivity property analogue to Lemma 18, which dooms to failure any attempt of a similar proof by induction. Actually, we conjecture[1] that there is a tower of cyclic extensions of height two $\mathbb{Q}^{\mathrm{ab}} \subset K \subset L$ where both $K/\mathbb{Q}^{\mathrm{ab}}$ and L/K are simple radical Galois extensions, but where $L/\mathbb{Q}^{\mathrm{ab}}$ is not Galois (even though $\mathbb{Q}^{\mathrm{ab}}$ contains all roots of unity). Such a counterexample would imply that any proof by induction where the induction hypothesis has the form "E_n/E_0 is Galois and [...]" is bound to fail.

Hence, some references end up applying Galois' fundamental theorem in a context where a premise – that some extension is Galois – does not hold. And those who exhibit a correct proof without relying on solvable extensions must reconstruct a radical series gradually, by adding all possible conjugates over the smallest field of the tower, at each step, which is exactly what is factored out in the definition of a solvable extension and in Lemma 18.

Moreover, all the proofs we found in the literature – including the ones relying on solvable extensions, such as in *Algebra*, Lang [15, VI, §7, Theorem 7.2] – delve into details about picking an appropriate primitive root of unity ω (e.g., using the least common multiple of all the prime exponents involved in the radical series) and reconstruct the full radical extension starting with the cyclotomic field extension $E(\omega)/E$ before starting an induction. We observe here that this detour is completely unnecessary when using solvable extensions.

6 Galois and Abel-Ruffini theorems

In this section we specialize results to $\mathbb{Q}$, which is sufficient to obtain the unsolvability of the general quintic. For possible generalizations of the results stated here, we refer to the discussions in Sections 8 and 9.

[1] `https://mathoverflow.net/questions/381824`

6.1 Galois' theorem

For a given polynomial in $P \in \mathbb{Q}[X]$, splitting fields for P over $\mathbb{Q}$ always exist, are isomorphic to each other and embed in the algebraic numbers (noted $\bar{\mathbb{Q}}$ in math style and `algC` in Coq) and though this embedding can be seen as a number field. We pick such a splitting field and call it $\mathbb{Q}(P)$, *the* splitting field of P. We pose the convention $\mathbb{Q}(0) = \mathbb{Q}$.

We write `numfield p` for $\mathbb{Q}(P)$ in Coq, it has type `splittingFieldType rat`, and there is a morphism `numfield_inC : {rmorphism numfield p → algC}` embedding $\mathbb{Q}(P)$ in $\bar{Q}$. There is also a function `numfield_roots : {poly rat} → seq (numfield p)` which lists the roots of P.

A polynomial P is solvable by radical if there is a field L that splits P, and such that L/K is solvable by radical. Note that L need not be $\mathbb{Q}(P)$, indeed the radicals involved in the decomposition of L may not belong to $\mathbb{Q}(P)$.

▶ **Definition 20.** *A nonzero polynomial $P \in \mathbb{Q}[X]$ is solvable by radicals if there is a field extension L and a subfield K of L which is a splitting field for P, and such that the extension $K/\mathbb{Q}$ is solvable by radicals.*

In Coq we use a slightly different definition (see Section 8) which we prove equivalent to the mathematical one.

```
Lemma solvable_poly_ratP (p : {poly rat}) : p != 0 →
  solvable_by_radical_poly p ↔
  ∃ L : splittingFieldType rat, ∃ K : {subfield L},
    splittingFieldFor 1 (p ^^ in_alg L) K ∧ solvable_by radical 1 K.
```

We can now recall Theorem 2 (Galois) and prove it formally for $F = \mathbb{Q}$:

▶ **Theorem 2** (Galois)**.** *A polynomial $P \in F[X]$ is solvable by radicals if and only if its Galois group is solvable.*

```
Theorem AbelGaloisPolyRat (p : {poly rat}) :
  solvable_by_radical_poly p ↔ solvable 'Gal({: numfield p} / 1).
```

Proof. First notice that by Remark 10 the right hand side of the equivalence "Gal($\mathbb{Q}(P)/\mathbb{Q}$) is solvable", is the same as $\mathbb{Q}(P)/\mathbb{Q}$ is a solvable extension. The left to right side is then a trivial application of Lemmas 19. And the right to left side consists in first extending $\mathbb{Q}(P)$ with a $[\mathbb{Q}(P) : \mathbb{Q}]^{\text{th}}$ primitive root of unity before applying Lemma 11. ◀

Now, in order to prove the Abel-Ruffini theorem, it suffices to exhibit a polynomial of degree 5 which Galois group is unsolvable. As in the literature, we pick $\mathfrak{S}_5$ and prove a certain class of polynomials has Galois group $\mathfrak{S}_5$: the irreducible rational polynomials with Prime Degree and Two Non Real Roots.

6.2 Irreducible rational polynomials of Prime Degree with exactly Two Non Real Roots

▶ **Lemma 21.** *Irreducible polynomials $P \in \mathbb{Q}[X]$ of prime degree p with exactly two non real roots have a Galois group over $\mathbb{Q}$ isomorphic to $\mathfrak{S}_p$.*

```
Lemma PDTNRR.isog_gal (p : {poly rat}) :
    irreducible_poly p → prime (size p).-1 →
    count [pred x | numfield_inC p x \isn't Creal] (numfield_roots p) = 2 →
  'Gal({: numfield p} / 1) \isog 'Sym_('I_(size p).-1)
```

Proof. Let $P \in \mathbb{Q}[X]$ be an irreducible polynomial of prime degree p, a sequence $s = (s_i)_i$ of its roots, and $G = \mathrm{Gal}(\mathbb{Q}(P)/\mathbb{Q})$ its Galois group. We define a group morphism $\varphi : G \to \mathfrak{S}_p$, so that $\forall i < p,\ \forall u \in G,\ s_{\varphi(u)(i)} = u(s_i)$. In other words φ maps an element u of the Galois group of P to a permutation of the indices of the sequence s that is compatible with the action of u on the roots s of P. Now, it suffices to show that φ is injective and surjective to conclude.

- φ is injective: let u be such that $\varphi(u) = \mathrm{id}$, it suffices to show that $u = \mathrm{id}$. Let $x \in \mathbb{Q}(P)$, x can be decomposed as a multivariate polynomial μ over $\mathbb{Q}$ applied to the sequence s, i.e., $x = \mu(s)$. Then $u(x) = u(\mu(s)) = \mu\left((u(s_i))_i\right) = \mu\left((s_{\varphi(u)(i)})_i\right) = \mu(s) = x$.
- φ is surjective: it suffices to show that there is a transposition τ and an element of order p in $\varphi(G)$. Indeed, since p is prime number we have $\mathfrak{S}_p = \langle \tau, c\rangle$.
  - Since P has exactly two non real roots, there are $i < j < p$ such that, $s_i = s_j^\star$ and $s_k = s_k^\star$ if $k \notin \{i, j\}$. The complex conjugation $(\cdot^\star)$ belongs to G and $\varphi(\cdot^\star) = (i\ j) = \tau$.
  - The natural number p divides $[\mathbb{Q}(P) : \mathbb{Q})]$ because P is irreducible. Since p is prime and divides G, by Cauchy's theorem, there is an element of order p in G. ◀

In Coq we did not link the theory of multivariate polynomials with the theory of field automorphism yet, instead we simply iterate on the sequence s and use univariate polynomials in each s_i.

6.3 $X^5 - 4X + 2$ is not solvable by radicals

There is no general formula for solving equations of degree greater than four (Theorem 3) because if there were, the equation $x^5 - 4x + 2 = 0$ would be solvable.

▶ **Theorem 22** (Insolvability of the quintic). *$X^5 - 4X + 2$ is not solvable by radicals.*

```
Theorem example_not_solvable_by_radicals :
  ¬ solvable_by_radical_poly ('X^5 - 4 *: 'X + 2 : {poly rat}).
```

Proof. By Theorem 2, it suffices to thow the galois group of $\mathbb{Q}(Q)/\mathbb{Q}$ is not solvable, where $Q = X^5 - 4X + 2$.

- By Lemma 21, it suffices to show Q is irreducible and has exactly two non real roots. Irreducibility is directly given by Eisenstein criterion. Q has at least three real roots in $\bar{\mathbb{Q}}$ because there are at least three sign changes: $Q(-2)Q(-1) < 0$, $Q(-1)Q(1) < 0$, and $Q(1)Q(2) < 0$. Finally since the derivative $Q' = 5X^4 - 4$ has exactly two real roots $(\pm\sqrt{\frac{2}{\sqrt{5}}})$, it means Q has at most three real roots, hence exactly three.
- To show $\mathfrak{S}_5$ is not solvable it suffices to show its normal subgroup $\mathfrak{A}_5$ is not solvable either. We conclude by contradiction with the fact that $\mathfrak{A}_5$ is simple of order $5 \times 4 \times 3$ and a simple solvable group must have prime order. ◀

7 Solvability by radicals is what you think

We now link the solvability by radical, as defined above, to the existence or not of analytic expressions for computing the roots of a given polynomial. Most of the time, this last step is considered mundane and is left to the reader. Here we give a formal treatment to it, both for intellectual satisfaction but also as a hint that our definition of a radical extension is correct.

More formally, for a field F, we define the grammar of radical expressions $\mathbb{E}_F$ over F as the set of terms that can be recursively defined from the symbols 0, 1, $x \in F$, $+$, $-$, $*$, $\cdot^{-1}$, $\sqrt[n]{\cdot}$ and ω_n where $\sqrt[n]{e}$ (resp. ω_n) stands for a n^{th}-root of e (resp. a n^{th}-primitive root of unity):

$$e \in \mathbb{E} ::= 0 \mid 1 \mid e_1 + e_2 \mid -e \mid e_1 * e_2 \mid e^{-1} \mid \sqrt[n]{e} \mid \omega_n \quad (n \in \mathbb{N}^*)$$

In Coq, as expected, we encode this set using an algebraic datatype. We then give an interpretation for terms in $\mathbb{E}$ in terms of algebraic numbers and w.r.t. an evaluation function `(iota : F → algC)`:

```
Variables (F : fieldType) (iota : F → algC).
Fixpoint algT_eval (f : algterm F) : algC :=
  match f with
  | Base x         => iota x
  | 0              => 0
  | 1              => 1
  | f1 + f2        => algT_eval f1 + algT_eval f2
  | - f            => - algT_eval f
  | f1 * f2        => algT_eval f1 * algT_eval f2
  | f ^-1          => (algT_eval f)^-1
  | f ^+ n         => (algT_eval f) ^+ n
  | n.+1-root f    => n.+1.-root (algT_eval f)
  | j.+1-prim1root => prim1root j.+1
  end.
```

It is worth mentioning that, in the listing above, the expressions on the left of `=>` are syntax whereas the ones on the right of `=>` are semantic, i.e., values in the type `algC` of algebraic numbers.

We now have all the necessary ingredients to state and prove the equivalence between being a solvable by radical polynomials and having roots expressible as a radical expression, as defined above:

```
Lemma solvable_formula (p : {poly rat}) : p != 0 →
  solvable_by_radical_poly p ↔
  {in root (p ^^ ratr), ∀ x, ∃ f : algterm rat, algR_eval ratr f = x}.
```

8 Classical reasoning in a constructive setting

8.1 Boolean reflection and effective Galois theory

The present contribution takes over the main design choices deployed in Mathematical Components library, and in particular its use of boolean reflection [18] for formalizing effective mathematics. Notably, the defining signature of algebraic structures, like rings or fields, involve boolean predicates, e.g., for comparison or discrimination of units. More generally, decidable predicates, that is predicates for which excluded-middle holds constructively, are formalized as boolean predicates. Consequently, equivalences between such boolean propositions are stated as equalities, as for instance in Lemma 15. Besides often saving the user from the technicalities of setoid rewriting, boolean specifications are provably proof-irrelevant, by Hedberg's theorem [13], and this feature is extensively used for defining and using proof-irrelevant dependent pairs.

The present development heavily relies on the effective perspective provided by the underlying linear algebra component [10]. In this library, vector spaces of finite dimension and their sub-spaces, always come with an explicit basis, and are in fact internally represented as matrices. This way, most properties of linear algebra in finite dimension are effective, thanks to variants of Gaussian elimination: computing the dimension of a sub-space, testing whether a family of vectors is free, whether it generates a given sub-space, testing the inclusion or the equality between sub-spaces, etc. When a larger vector space is in fact an algebra (resp. a field extension) over a given base field, it is decidable whether a given subspace is in fact a sub-algebra U (resp. a sub-field U): it suffices to test whether pairwise products of elements of the basis of U belong to U. Note however that effectivity does not mean that the computations are necessarily tractable in practice: turning these effective definition into formally verified algebra that can be executed on concrete entries would require a non-trivial additional effort [7, 24].

The main effect of this effective take on linear algebra in the case of (finite) field extensions is the definition of boolean functions for testing whether a (finite) field extension is normal, separable or Galois. In addition, the construction of normal closure is effective, as well as that of the Galois group of an extension. As the finite group theory component of the Mathematical Components library provides a boolean test for the solvability of finite group, solvability of an extension is decidable as well.

8.2 Non-effective results

However, important properties in commutative algebra, such as testing a polynomial in $F[X]$ for irreducibility for F an arbitrary field, remain non-effective, even in the case of a field F with a decidable equality. As a consequence, in a constructive setting, some facts like Lemma 16 cannot be proved as such. The way out is to change their statement for a classically equivalent one, typically, a double-negated version, so as to restore constructive provability. For this purpose, we use the `classically` monadic predicate [11]: for any `P : Prop` , `classically P` is equivalent to the double-negation `¬(¬P)`. For instance, the construction of a larger normal field extension performed in Lemma 16 is not effective in general. Here is a typical example of non-effective statement:

```
Lemma classic_baseCycloExt F n : (n%:R != 0 :> F) → classically
  { L' : splittingFieldType F & { w : L' &
     <<1; w>> = {: L'} & n.-primitive_root w }}.
```

The `classically` monad thus seals the sigma-type, which is itself an effective existential statement. However thanks to the formal definition of the `classically` predicate, a hypothesis of the form `classically P` can be used directly as if it were of the form `P` in particular for proving a boolean statement (and because `¬(¬b) ↔ b` holds constructively).

Continuing our example, Lemma `classic_baseCycloExt` is used in the proof of Lemma 17, in order to establish that a simple extension $E(x)/E$ is solvable, which is stated formally as `solvable_ext E <<E; x>>`. Since the `solvable` predicate is boolean (see Section 8.1), lemma `classic_baseCycloExt` can be used without propagating the `classically` monad to the final formal statement of Lemma 17.

8.3 Stating Galois' theorem in characteristic zero

In a constructive setting, it is not possible to rely on the existence of an algebraic closure when needed, as is commonly assumed in the standard literature, and this even in the case of a base field F_0 with zero characteristic. Our current formal statement of Galois' theorem for arbitrary field extensions in zero characteristic thus reads:

▶ **Theorem 23.** *Let L/F_0 be a normal extension of characteristic zero and F/E a field extension in L. Suppose that $\omega \in L$ is a primitive $[\mathrm{NCl}_E(F) : E]^{th}$ root of unity. Then F/E is solvable by radicals if and only if it is solvable.*

```
Theorem AbelGalois  (F0 : fieldType) (L : splittingFieldType F0) (w : L)
    (E F : {subfield L}) : (E ≤ F) → has_char0 L →
    (\dim_E (normalClosure E F)).-primitive_root w →
  solvable_by radical E F ↔ solvable_ext E F.
```

In the literature "F solvable by radicals" is defined as the existence of a certain radical extension containing F. This definition actually allows us to get rid of the assumption on the existence of a root of unity, as in the above theorem. This assumption, which is only needed for the right-to-left implication (see Lemma 19), would indeed be encompassed by the definition of "solvable by" in the right-to-left implication.

Alas, strengthening the definition of "solvable by radicals" in order to match the variant found in the literature – and thus dropping the assumption on the existence of a root of unity – would not make the right-to-left implication a direct consequence of Lemma 11 in the current state of the formalization. It is actually not clear to us whether this would be provable at all constructively. Indeed, we know no constructive way to test the presence of a primitive root of unity in L, or to extend L with such a hypothetical primitive root of unity.

We could however use classical axioms, or the `classically` monad of Section 8.2, to recover the standard wording found in the literature. Another option would be to construct, effectively, extensions of number fields with an arbitrary algebraic element, e.g.,with a primitive root of unity. This way, results from Section 6 that have been specialized to $\mathbb{Q}$ could in principle be generalized to any number field, or, even to factorial fields [20], i.e., fields equipped with an effective irreducibility test for polynomials.

9 Conclusion

Comparison to related formalization in Coq

This work represents a significant extension of the Mathematical Components library, both in size and in contents. This background proved to be sufficiently mature so that we didn't need to change the definitions and formalization choices. This work is grounded on the three main algebraic hierarchies which are the backbone of the Mathematical Components library: hierarchies of structures (from additive groups to field extensions, and real closed fields), hierarchies of morphisms (of additive groups, rings, algebra, and fields), and hierarchies of predicates (sub-groups, vector sub-spaces, sub-algebras, sub-fields).

These hierarchies are designed using Coq's canonical structures mechanism [22], more precisely with the packed class methodology [9], in order to achieve ad-hoc polymorphism [12, 17]. This inference mechanism is crucial to combine the different components of the library: finite group theory, linear algebra, theory of field extensions and Galois theory. Inference of structures is used at almost every single line of code and its efficiency is crucial for making amenable such a development.

Comparison to related formalization in other systems

The only formalization of Galois theory we are aware of has been carried in Lean/mathlib. This work is at an early stage of development as only the Galois correspondence is currently proven. This development relies on previously defined algebraic structures by the Lean/mathlib community [19], such as fields, vector spaces, algebras and their morphisms.

A formalization of field extensions and algebraic closure [6] was carried out in the Isabelle/HOL theorem prover. Despite the lack of dependent types, this library comes with a definition of the algebraic closure of an abstract field as opposed to the a more elementary construction for a fixed field such as $\mathbb{Q}$. However, it is unclear whether the methodology used there can be further extended for the formalization of Galois theory. At least, dependent types play a central role in the design choices at stake in the present development.

The Mizar library contains core definitions and results related to field extensions [23].

Last, there exists an unfinished development related to the formalization of Galois theory and unsolvability of the quintic in LEGO [1]. However, only the unsolvability of the symmetric group [3] has been formally addressed.

Comparison to the pen and paper literature

In this paper, we give a comprehensive outline of the Abel-Ruffini theorem. This outline serves as a basis to our formal development and has only been made possible by a careful synthesis work of the numerous definitions and proofs from the literature.

We noticed several variations in the definitions of “radical extensions” and “solvable by radical” (extension), which are the same but may denote two different things: one corresponding to our definition of “radical extension” and the other corresponding to our definition “solvable by radical”. Indeed both definitions are useful and we must give a precise name to each. Perhaps the most surprising takeaways from this synthesis work are the remarks that follow the proof of Lemma 19. Many references give a fine-grained description of a modification of the radical series which would give the right induction hypothesis, which can be avoided by the definition of a solvable extension.

The proof of unsolvability of X^5-4X+2 involves counting its real roots. The most common way of doing this relies on building *sign tables*. However, the Mathematical Components library does not give any formal treatment of sign tables and we had to roll out our own solution. Fortunately, the MathComp-Real-Closed [5] library provides results related to the study of the variations of a polynomial with coefficients in an algebraically closed field. This allowed us to give lower and upper bounds on the number of reals without having to formalize sign tables. However, we expect that a formal treatment of sign tables to be a useful addition to the Mathematical Components library.

On the same subject, the library MathComp-Real-Closed contains a quantifier elimination procedure and a root counting procedure. In theory, in order to obtain the number of real roots, it would have been possible to simply run this procedure on the targeted polynomial. However, in practice, due to the very inefficient nature of the involved datatypes (starting from the use of unary natural numbers), the methodology proved to be too inefficient. A possible future work would be to extend CoqEAL [7, 4] to make effective these procedures.

The case of positive characteristic

Even if a large part of our development is independent from the characteristic of the fields under consideration, for the sake of simplifying, we sometime restricted ourselves to the case of characteristic zero – as this is the case in the file `abel.v` for example. For instance, we specialized the notion of radical extension to fields of characteristic zero, which is enough to show the unsolvability of a polynomial over $\mathbb{Q}$. However, we expect that the zero-characteristic assumption could be dropped in the near future. For example, the definition of radical extensions in a field of an arbitrary characteristic p could be generalized by following the definition from Lang [15, VI, §7, Remark], thus adding a second kind of radical extensions $K(a)/K$ such that $a^p - a \in K$. The proof that cyclic extensions of degree p are of that form would then rely on the additive version of Hilbert Theorem 90. (The multiplicative version is already formalized – it could be used in place of Lemma 4 – and we do not expect any difficulty in formalizing its additive counterpart.)

Reasoning up to isomorphisms

A substantial amount of proof scripts is devoted to the transfer of properties from one object to an isomorphic one (See Lemma 15 for an example). This part is largely left implicit on paper and it is indeed quite mundane. It would be interesting to see if the ongoing work around *Homotopy Type Theory* [26, 24, 2] could apply here.

References

1 Peter Aczel. Galois: a theory development project. *manuscript, University of Manchester*, 1993.

2 Carlo Angiuli, Evan Cavallo, Anders Mörtberg, and Max Zeuner. Internalizing representation independence with univalence. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. `doi:10.1145/3434293`.

3 Gilles Barthe. A formal proof of the unsolvability of the symmetric group over a set with five or more elements. URL: `https://ftp.cs.ru.nl/CSI/CompMath.Found/sn.ps.Z`.

4 Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for Free! In *Certified Programs and Proofs*, pages 147–162, Melbourne, Australia, 2013. `doi:10.1007/978-3-319-03545-1_10`.

5 Cyril Cohen and Assia Mahboubi. Formal proofs in real algebraic geometry: from ordered fields to quantifier elimination. *Logical Methods in Computer Science*, 8(1:02):1–40, 2012. `doi:10.2168/LMCS-8(1:02)2012`.

6 Paulo Emílio de Vilhena and Lawrence C. Paulson. Algebraically closed fields in isabelle/hol. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 204–220, Cham, 2020. Springer International Publishing.

7 Maxime Dénès, Anders Mörtberg, and Vincent Siles. A refinement-based approach to computational algebra in COQ. In Lennart Beringer and Amy Felty, editors, *ITP - 3rd International Conference on Interactive Theorem Proving - 2012*, volume 7406 of *Lecture Notes In Computer Science*, pages 83–98, Princeton, United States, August 2012. Springer. `doi:10.1007/978-3-642-32347-8_7`.

8 Evariste Galois. *Mémoire sur les conditions de résolubilité des équations par radicaux*, volume XI of *Journal de mathématiques pures et appliquées*. Joseph Liouville, 1846. URL: `https://www.bibnum.education.fr/sites/default/files/galois_memoire_sur_la_resolubiblite.pdf`.

9 François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging Mathematical Structures. working paper or preprint, 2009. URL: `https://hal.inria.fr/inria-00368403`.

10 Georges Gonthier. Point-Free, Set-Free Concrete Linear Algebra. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving - ITP 2011*, volume 6898 of *Lecture Notes in Computer Science*, pages 103–118, Berg en Dal, Netherlands, 2011. Radboud University of Nijmegen, Springer. `doi:10.1007/978-3-642-22863-6_10`.

11 Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A Machine-Checked Proof of the Odd Order Theorem. In Sandrine Blazy, Christine Paulin, and David Pichardie, editors, *ITP 2013, 4th Conference on Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 163–179, Rennes, France, 2013. Springer. doi:10.1007/978-3-642-39634-2_14.

12 Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. How to make ad hoc proof automation less ad hoc. *SIGPLAN Not.*, 46(9):163–175, 2011. doi:10.1145/2034574.2034798.

13 Michael Hedberg. A coherence theorem for Martin-Löf's Type Theory. *J. Funct. Program.*, 8(4):413–436, 1998. doi:10.1017/S0956796898003153.

14 Abel Niels Henrik. *Œuvres complètes de Niels Henrik Abel, mathématicien, nouvelle édition publiée aux frais de l'État norvégien par L. Sylow et S. Lie.* Imprimerie de Grøndahl & søn, 1881. URL: https://www.abelprize.no/c54178/artikkel/vis.html?tid=54181.

15 Serge Lang. *Algebra.* Graduate Texts in Mathematics. Springer New York, 2005.

16 Assia Mahboubi. The rooster and the butterflies. In Jacques Carette, David Aspinall, Christoph Lange, Petr Sojka, and Wolfgang Windsteiger, editors, *Intelligent Computer Mathematics - MKM, Calculemus, DML, and Systems and Projects 2013, Held as Part of CICM 2013, Bath, UK, July 8-12, 2013. Proceedings*, volume 7961 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013. doi:10.1007/978-3-642-39320-4_1.

17 Assia Mahboubi and Enrico Tassi. Canonical structures for the working coq user. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, pages 19–34, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

18 Assia Mahboubi and Enrico Tassi. *Mathematical Components.* Zenodo, 2021. doi:10.5281/zenodo.4457887.

19 The mathlib Community. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 367–381, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372885.3373824.

20 R. Mines, F. Richman, and W. Ruitenburg. *A Course in Constructive Algebra.* Universitext. Springer New York, 1987.

21 P. Ruffini. *Teoria generale delle equazioni: in cui si dimostra impossibile la soluzione algebraica delle equazioni generali di grad superiore al quarto.* Number pt. 1 in Nineteenth Century Collections Online (NCCO): Science, Technology, and Medicine: 1780-1925. Nella stamperia di S. Tommaso d'Aquino, 1799.

22 Amokrane Saibi. Typing algorithm in type theory with inheritance. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 292–301, 1997.

23 Christoph Schwarzweller. On roots of polynomials over F[X]/(p). *Formaliz. Math.*, 27(2):93–100, 2019. doi:10.2478/forma-2019-0010.

24 Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. Equivalences for free: univalent parametricity for effective transport. *Proc. ACM Program. Lang.*, 2(ICFP):92:1–92:29, 2018. doi:10.1145/3236787.

25 The Mathematical Components Team. The Mathematical Components library. https://github.com/math-comp/math-comp, 2007.

26 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

Itauto: An Extensible Intuitionistic SAT Solver

Frédéric Besson
Inria, Univ Rennes, Irisa, Rennes, France

Abstract

We present the design and implementation of `itauto`, a Coq reflexive tactic for intuitionistic propositional logic. The tactic inherits features found in modern SAT solvers: definitional conjunctive normal form; lazy unit propagation and conflict driven backjumping. Formulae are hash-consed using native integers thus enabling a fast equality test and a pervasive use of Patricia Trees. We also propose a hybrid proof by reflection scheme whereby the extracted solver calls user-defined tactics on the leaves of the propositional proof search thus enabling theory reasoning and the generation of conflict clauses. The solver has decent efficiency and is more scalable than existing tactics on synthetic benchmarks and preliminary experiments are encouraging for existing developments.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Computing methodologies → Theorem proving algorithms

Keywords and phrases SAT solver, proof by reflection

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.9

Supplementary Material *Software*: `https://gitlab.inria.fr/fbesson/itauto`
archived at `swh:1:dir:986dc28c9844ba2faedc3d134b2acfb31e8c27c4`

Acknowledgements Thanks are due to Alix Trieu for sharing with us his verified Patricia Tree library. Thanks are also due to Samuel Gruetter for stress testing the tactic; his feedbacks have been very valuable.

1 Introduction

Using an ideal proof-assistant, proofs would be written at high-level and mundane proof tasks would be discharged by automated procedures. However, automated reasoning is hard; even more so for *sceptical* [17] proof-assistants: generating and verifying proofs at scale, even for decidable logic fragments, is a challenge requiring sophisticated implementation strategies [7, 4, 2]. For the Coq proof-assistant, the situation is made slightly worse because intuitionistic logic is not mainstream for automated provers. Thus, the Satisfiability Modulo Theory (SMT) approach that is based on a classical SAT solver needs to be revisited.

1.1 Propositional Reasoning and Theory Reasoning in Coq

There are a variety of Coq tactics which perform intuitionistic propositional and theory reasoning. We describe here their main features and explain some limitations, thus motivating the need for a novel (extensible) solver for intuitionistic propositional logic (IPL). We defer to Section 6 the discussion of related approaches rooted in a classical setting and interfacing with external provers.

`tauto` [1] is a complete decision procedure for IPL based on the LJT* calculus [14]. `rtauto` [2] is another decision procedure for IPL verifying proof certificates using proof by reflection. These decision procedures are usually efficient enough for interactive use but do not perform theory reasoning. `lia`[3] is a decision procedure for linear arithmetic. It has a classical

[1] `https://coq.inria.fr/refman/proof-engine/tactics.html#coq:tacn.tauto`
[2] `https://coq.inria.fr/refman/proof-engine/tactics.html#coq:tacn.rtauto`
[3] `https://coq.inria.fr/refman/addendum/micromega.html#coq:tacn.lia`

understanding of propositional connectives but abstracts away non-arithmetic propositions. `congruence`[4] [11] does not perform propositional reasoning but decides the theory of equality with constructors. Another tactic that is worth mentioning is `intuition tac`[5]; it first performs propositional reasoning and calls the leaf tactic `tac` when it gets stuck. (`tauto` is actually `intuition fail`.)

In a classical setting, calling a theory solver on the leaves of a propositional proof search is the basis of the *DPLL*(T) algorithm [16]. Unfortunately, in an intuitionistic setting, completeness is lost. Example 1 illustrates the issue.

▶ **Example 1** (Incomplete Combination). Consider the following goals.

```
Lemma Ex1:∀(p:Prop)(x:Z), x=0 → (x>=0 → p) → p.
Lemma Ex2:∀(A:Type)(p:Prop)(a b c:A),a=b → b=c → (a=c → p) → p.
```

For `Ex1` and `Ex2`, `intuition lia` (resp. `intuition congruence`) fail whereas the goal can be decided using a combination of propositional logic with the theory of linear arithmetic and the theory of equality, respectively. The reason is that the `intuition` part does nothing and therefore calls the leaf tactic on the unmodified goal. For `Ex1`, `lia` abstracts away non-arithmetic propositions[6] and is left with the non-theorem `x=0→(x>=0→True)→False`. For `Ex2`, `congruence` abstracts away propositions and is also left with a non-theorem: `a=b→b=c→False`.

Besides completeness, `intuition tac` may also call the leaf tactic `tac` more often than necessary. This is illustrated by Example 2.

▶ **Example 2** (Spurious Leaf Tactic Call). Consider the following goal.

```
Lemma Ex3:∀(A:Type)(x y t:A)(p q:Prop),x=y→p∨q→(p→y=t)→(q→y=t)→x=t.
```

In this case, `intuition congruence` performs a case-split over `p\/q`, and derives `y=t` by *modus-ponens*. Eventually, it calls the leaf tactic `congruence` twice. However, in both branches, `congruence` solves the same theory problem *i.e.*, `x=y → y=t → x=t`.

1.2 Contribution

A first contribution is the design and implementation of a reflexive intuitionistic SAT solver in Coq. The SAT solver obtains decent performances using features found in state-of-the-art SAT solvers such as hash-consing, lazy unit propagation, backjumping and theory learning. It also makes a pervasive use of native machine integers[7] and Patricia Trees [22].

Another contribution is a variation of the proof by reflection approach whereby the verified extracted SAT solver is first run inside the proof engine. Theory reasoning is then performs by calling user-defined tactics on the leaves of the propositional proof search. Following the *DPLL*(T) approach, minimal conflict clauses obtained from the tactic proof-terms are passed back to the SAT solver. Eventually, a proof is obtained by asserting the conflict clauses and re-running the reflexive SAT solver with an empty theory module. To our knowledge, this design combining reflexive code with tactics is original.

[4] `https://coq.inria.fr/refman/proof-engine/tactics.html#coq:tacn.congruence`
[5] `https://coq.inria.fr/refman/proof-engine/tactics.html#coq:tacn.intuition`
[6] The propositional variable `p` is replaced by `True` or `False` depending on its polarity.
[7] `https://coq.inria.fr/refman/language/core/primitive.html#primitive-integers`

The rest of the paper is organised as follows. In Section 2, we present the main features of our SAT solver and its implementation in Coq. The structure of the soundness proof is detailed in Section 3. In Section 4, we explain how to interface the SAT solver with the proof engine and user-defined tactics. We show experimental results in Section 5. Related work is presented in Section 6 and Section 7 concludes.

Note that the code snippets in the paper have been edited and idealised for clarity.

2 Design of an Intuitionistic SAT solver

Our algorithm is reusing several key components of modern SAT solvers. Fortunately, they are only slightly adapted to the intuitionistic setting.

2.1 Syntax and Semantics of Formulae

Our SAT solver takes as input hash-consed [1] formulae defined by the inductive type `LForm`.

```
Inductive LForm : Type :=
| LFF | LAT : int → LForm | LOP : lop → list (HCons.t LForm) → LForm
| LIMPL : list (HCons.t LForm) →  (HCons.t LForm)  → LForm.
```

The syntax is mostly standard and models n-ary propositional operators. `LFF` represents the proposition `False`. `LAT` i represents the propositional variable $\mathtt{p}_i$. `LOP o` $[f_1;\ldots;f_n]$ where `o` $\in$`{AND, OR}` represents a n-ary conjunction or disjunction. `LIMPL` $[f_1;\ldots;f_n]$ f represents a n-ary implication with f_1, ..., f_n being the premisses and f the conclusion. The negation of a formula f is encoded by `LIMPL` $[f]$ `LFF`. All sub-formulae are hash-consed *i.e.* `f:HCons.t LForm` is a pair made of a formula and a unique index. Because it enables a fast equality test of formulae in $O(1)$, hash-consing is essential for efficiency. N-ary operators allow for a sparser conjunctive normal form (see *e.g.* [20]).

The interpretation of a formula `f:LFORM` is given by structural recursion with respect to an environment `e:int → Prop` mapping indexes *i.e.*, propositional variables, to propositions.

$$
\begin{array}{lcl}
[\![\mathtt{LFF}]\!]_e & = & \mathtt{False} \\
[\![\mathtt{LAT}\ i]\!]_e & = & e\ i \\
[\![\mathtt{LOP\ AND}\ [f_1;\ldots;f_n]]\!]_e & = & [\![f_1]\!]_e \wedge \cdots \wedge [\![f_n]\!]_e \\
[\![\mathtt{LOP\ OR}\ [f_1;\ldots;f_n]]\!]_e & = & [\![f_1]\!]_e \vee \cdots \vee [\![f_n]\!]_e \\
[\![\mathtt{LIMPL}\ [f_1;\ldots;f_n]\ f]\!]_e & = & [\![f_1]\!]_e \to \cdots \to [\![f_n]\!]_e \to [\![f]\!]_e
\end{array}
$$

In the following, as the environment e is fixed, we drop the subscript and write $[\![f]\!]$ for $[\![f]\!]_e$.

2.2 Intuitionistic Clausal Form

In a classical setting, to prove that a formula f is a tautology, a modern SAT solver proves that the conjunctive normal form (CNF) of the negation of f is unsatisfiable. In other words, a SAT solver exploits the fact that $\neg(\neg f)$ is equivalent to f and that the CNF conversion preserves provability. In an intuitionistic setting, this approach is not feasible because *reductio ad adsurdum* is not logically sound and the usual CNF conversion requires De Morgan laws which are not admissible. Yet, Claessen and Rosén [10] show that it is possible to transform an intuitionistic formula f into an equi-provable formula of the form $\bigwedge F \wedge \bigwedge I \to q$ where q is a variable ; $f \in F$ is a so-called *flat clause* of the form $p_1 \to \ldots p_n \to q_1 \vee \cdots \vee q_m$ where the p_i and q_i are variables and $i \in I$ is a so-called *implication clause* of the form $(a \to b) \to c$ where a, b and c are variables. The transformation is based on Tseitin definitional CNF [26] with the modification that a clause is written $p_1 \to \cdots \to p_n \to q_1 \vee \cdots \vee q_n$ instead of $\neg p_1 \vee \cdots \vee \neg p_n \vee q_1 \cdots \vee q_n$. The *implication clauses* are reminiscent of the fact that double arrows (*e.g.*, in the LJT proof system) need a treatment that is specific to intuitionistic logic.

Our implementation is along these lines but optimises the transformation in order to reduce both the set of *flat clauses* and *implication clauses*. To reduce the set of *flat clauses*, our definitional CNF is based on Plaisted and Greenbaum CNF [24] which exploits the polarity of formulae and uses memoization thus avoiding recomputing the CNF of identical sub-formulae. In order to reduce the set of *implication clauses*, we exploit the fact that, if the conclusion q is decidable, intuitionistic logic reduces to classical logic. In that case, it is therefore admissible to replace an *implication clause* $(a \to b) \to c$ by the equivalent *flat clauses* $\{a \vee c; b \to c\}$. Moreover, before performing CNF conversion, formulae are flattened using the associativity of $\wedge$ and $\vee$. This has the advantage of augmenting the arity of the operators, reducing the depth of the formulae, and therefore reducing the number of intermediate propositional variables.

2.2.1 Pre-processing

The input formula is only using binary operators. To obtain n-ary operators, we recursively apply the following equivalences (the operator $+\!\!+$ is the concatenation of lists):

$$\begin{array}{lcl}
\texttt{LOP OR}\ (\texttt{LOP OR}\ l_1) :: l_2 & = & \texttt{LOP OR}\ (l_1 +\!\!+ l_2) \\
\texttt{LOP AND}\ (\texttt{LOP AND}\ l_1) :: l_2 & = & \texttt{LOP AND}\ (l_1 +\!\!+ l_2) \\
\texttt{LIMPL}\ (\texttt{LOP AND}\ l_1) :: l_2\ r & = & \texttt{LIMPL}\ (l_1 +\!\!+ l_2)\ r \\
\texttt{LIMPL}\ l_1(\texttt{LIMPL}l_2\ r) & = & \texttt{LIMPL}\ (l_1 +\!\!+ l_2)\ r
\end{array}$$

2.2.2 Literals

Tseitin style CNF [26] consists in introducing fresh propositional variables. In a proof-assistant, modelling freshness may incur some proof overhead. Fortunately, in our case, the new propositional variables are not arbitrary but correspond to sub-formulae. As a result, we represent a propositional variable by a hash-consed formula (`HFormula` = `HCons.t LForm`) and a literal is a positive or negative hash-consed formula.

```
Inductive literal : Type := | POS (f: HFormula) | NEG (f: HFormula)
```

As usual, a clause is a list of positive and negative literals but with the following interpretation.

$$[\![\texttt{NEG}f :: l]\!]_e = [\![f]\!]_e \to [\![l]\!]_e \qquad [\![\texttt{POS}f :: l]\!]_e = [\![f]\!]_e \vee [\![l]\!]_e \qquad [\![[]]\!]_e = \texttt{False}$$

For readability, we will write $\lfloor f \rfloor$ for $\texttt{POS}f$ and $\lfloor f \rfloor \to$ for $\texttt{NEG}f$. A singleton clause $[\texttt{NEG}f]$ will be written $\lfloor f \rfloor \to \bot$. In the remaining, we have the invariant that the negative literals are always before the positive literals. As a result, a general clause $[\texttt{NEG}f_1; \ldots; \texttt{NEG}f_i; \texttt{POS}g_1; \ldots; \texttt{POS}g_j]$ will be written $\lfloor f_1 \rfloor \to \cdots \to \lfloor f_i \rfloor \to \lfloor g_1 \rfloor \vee \cdots \vee \lfloor g_j \rfloor$ or more compactly $\bigwedge_i \lfloor f_i \rfloor \to \bigvee_j \lfloor g_j \rfloor$. Moreover, we define the negation of a literal l: $\neg l$ is such that $\neg \lfloor f \rfloor = \lfloor f \rfloor \to$ and $\neg \lfloor f \to \rfloor = \lfloor f \rfloor$.

2.2.3 Introduction Rule

Before running the CNF conversion *per se*, we inspect the formula and perform *reductio ad adsurdum* if possible. The function `intro_impl` performs the introduction rule for implication with the twist that the conclusion is double negated if it is decidable. Therefore, `intro_impl: HFormula → (list literal * HFormula)` takes as input a hash-consed formula and returns a pair (l, c) where l are hypotheses and c is the conclusion.

```
intro_impl (LIMPL  l r) = if is_dec r then ((NEG r):: map POS l , HLFF)
                                       else (map POS l, r)
intro_impl f    = if is_dec f then ([NEG f] , HLFF) else ([], f)
```

The constant HLFF is the hash-consed formula LFF *i.e.*, the syntax for the proposition False. The is_dec predicate is implemented by a boolean flag stored together with the hash-cons of the formula. It is recursively propagated from atomic propositions that are known to be classical *i.e.*, we have $r \vee \neg r$.

2.2.4 Construction of the CNF

The next step consists in computing the CNF of the literals and of the formula in the conclusion. We recursively compute for a positive literal CNF- *i.e.*, a set of clauses modelling the elimination rule; and for a negative literal (or the conclusion) CNF+ *i.e.*, a set of clauses modelling the introduction rule.

$$\frac{f = (f_1 \wedge \cdots \wedge f_n)}{\begin{array}{l}\texttt{AND-}(f) = \{\lfloor f\rfloor \to \lfloor f_1\rfloor ; \ldots ; \lfloor f\rfloor \to \lfloor f_n\rfloor\} \\ \texttt{AND+}(f) = \{\lfloor f_1\rfloor \to \cdots \to \lfloor f_n\rfloor \to \lfloor f\rfloor\}\end{array}} \qquad \frac{f = (f_1 \vee \cdots \vee f_n)}{\begin{array}{l}\texttt{OR-}(f) = \{\lfloor f\rfloor \to \lfloor f_1\rfloor \vee \cdots \vee \lfloor f_n\rfloor\} \\ \texttt{OR+}(f) = \{\lfloor f_1\rfloor \to \lfloor f\rfloor ; \ldots ; \lfloor f_n\rfloor \to \lfloor f\rfloor\}\end{array}}$$

$$\frac{f = (f_1 \to \cdots \to f_n \to r)}{\begin{array}{l}\texttt{IMPL-}(f) = \{\lfloor f\rfloor \to \lfloor f_1\rfloor \to \cdots \to \lfloor f_n\rfloor \to \lfloor r\rfloor\} \\ \texttt{IMPL+}(f) = \{\lfloor r\rfloor \to \lfloor f\rfloor\} \cup \bigcup_{\texttt{is\_dec}\ f_i}\{\lfloor f_i\rfloor \vee \lfloor f\rfloor\}\end{array}}$$

This clausal encoding is correct and the generated clauses are both classical and intuitionistic tautologies. Except for IMPL+, the clausal encoding is also complete *e.g.*, we have

$$\begin{array}{lll}\texttt{CNF+}(f_1 \wedge \cdots \wedge f_n) \cup \{\lfloor f_1\rfloor \wedge \cdots \wedge \lfloor f_n\rfloor\} & \vdash & \lfloor f_1 \wedge \cdots \wedge f_n\rfloor \\ \texttt{CNF-}(f_1 \wedge \cdots \wedge f_n) \cup \{\lfloor f_1 \wedge \cdots \wedge f_n\rfloor\} & \vdash & \lfloor f_1\rfloor \wedge \cdots \wedge \lfloor f_n\rfloor\end{array}$$

For IMPL+, if not all the f_i are classical propositions, the clausal encoding is partial and we also keep the *implication clause* $(\lfloor f_1\rfloor \to \cdots \to \lfloor r\rfloor) \to \lfloor f\rfloor$ for later processing. An exception is when we intend to prove False. In that case, the IMPL+ rule may drop the requirement on the decidability of the f_i thus providing a complete clausal encoding. This is sound because for any $f = (f_1 \to \cdots \to f_n \to r)$, $(\lfloor r\rfloor \to \lfloor f\rfloor) \wedge \bigwedge_{f_i}(\lfloor f_i\rfloor \vee \lfloor f\rfloor) \to \texttt{False}$ is an intuitionistic tautology.

▶ **Example 3.** Suppose that we intend to prove the tautology $(a \to (b \wedge c)) \to (b \vee (a \to c))$ where a, b and c are intuitionistic propositional variables. After running the introduction rule, we obtain the following hypothesis and conclusion

$$\{\lfloor a \to (b \wedge c)\rfloor\} \vdash b \vee (a \to c)$$

We compute CNF- for the hypothesis *i.e.* IMPL-($\lfloor a \to (b \wedge c)\rfloor$), and CNF+ for the conclusion *i.e.*, OR+($\lfloor d \vee (a \to c)\rfloor$). Recursively, we compute AND-($b \wedge c$) and IMPL+($a \to c$) and eventually obtain

$$\left\{\begin{array}{l}\lfloor a \to (b \wedge c)\rfloor, \lfloor a \to (b \wedge c)\rfloor \to \lfloor a\rfloor \to \lfloor b \wedge c\rfloor, \\ \lfloor b \wedge c\rfloor \to \lfloor b\rfloor, \lfloor b \wedge c\rfloor \to \lfloor c\rfloor, \\ \lfloor b\rfloor \to \lfloor b \vee (a \to c)\rfloor, \lfloor a \to c\rfloor \to \lfloor b \vee (a \to c)\rfloor \\ \lfloor c\rfloor \to \lfloor a \to c\rfloor, (\lfloor a\rfloor \to \lfloor c\rfloor) \to \lfloor a \to c\rfloor\end{array}\right\} \vdash b \vee (a \to c)$$

Note that because a is an intuitionistic proposition, we keep the *implication clause* $(\lfloor a\rfloor \to \lfloor c\rfloor) \to \lfloor a \to c\rfloor$.

2.3 Lazy Unit Propagation

Once a clause is reduced to a single literal, say l, *unit propagation* simplifies all the remaining clauses using l. There are three cases to consider. If a clause c does not mention l, it is left unchanged. If the literal l belongs to the clause c, the clause is redundant and it is removed (see `R1` and `R2`). If the negation of the literal l belongs to the clause, we deduce the simplified clause $c \setminus \neg l$ (see `M1` and `M2`). In logic terms, we have the following inferences:

$$\texttt{R1}\,\frac{p}{p \vee r} \qquad \texttt{R2}\,\frac{p \to \bot}{p \to r} \qquad \texttt{M1}\,\frac{p \quad p \to r}{r} \qquad \texttt{M2}\,\frac{p \to \bot \quad p \vee r}{r}$$

In the following, a literal l is said to be *watched* if neither $[l]$ nor $[\neg l]$ is a unit clause. We also say that a literal l is *assigned* if it is not *watched*.

A naive unit propagation algorithm linearly traverses every clause and is therefore inefficient. A key observation is that the purpose of unit propagation is to produce new unit clauses. Said otherwise, it not necessary to traverse clauses where at least 2 literals are watched. As a result, a clause that is neither the empty clause nor the unit clause is represented by the type `watched_clause` given below.

```
Record watched_clause:={
  watch1 : literal;  watch2 : literal; unwatched : list literal }.
```

This is purely functional variant of so called head/tail lists[8] [28, 29] that is simpler than the 2-watched literals optimisation [21]. Watched clauses are indexed on their watched literals but also on whether the watched literals are positive or negative. To get an efficient representation of sets of clauses, each clause is given a unique identifier and is stored in a Patricia Tree [22]. In order to implement so-called *non-chronological backtracking* (see Section 2.4), we also track the set of literals that are needed to deduce a clause. As a result, each clause is annotated with a set of literals `LitSet.t`.

```
  Record Annot (A: Type) := { elt := A ; deps := LitSet.t }
```

Therefore, unit propagation operates on the following `watch_map` data-structure.

```
Definition clause_set  := ptrie (Annot watched_clause).
Definition watch_map   := ptrie (clause_set * clause_set).
```

A `watch_map` m maps a propositional variable f *i.e.*, a hash-consed formula, to two sets (n, s) of clauses where clauses in n are watched by $\lfloor f \rfloor \to$ and clauses in p are watched by $\lfloor f \rfloor$. More precisely, we have

$$\begin{array}{lll} c \in n & \textit{iff} & c.watch1 = (\lfloor f \rfloor \to) \vee (c.watch2 = \lfloor f \rfloor \to) \\ c \in p & \textit{iff} & c.watch1 = \lfloor f \rfloor \vee c.watch2 = \lfloor f \rfloor \end{array}$$

Suppose that we perform unit propagation for the literal $\lfloor f \rfloor$ for a `watched_map` m such that $m[f] = (n, p)$. The clauses in p are redundant and can be dropped; the clauses in n need to be processed and reduced. The reduction takes as argument the set of literals that are currently assigned, a watched clause c and returns an annotated clause `Annot clause` where the type `clause` is given below

```
Inductive clause:=|EMPTY |TRUE |UNIT (l:literal)|CLAUSE (wc:watched_clause)
```

[8] Head/tail lists give access to the first and last element of a list in $O(1)$.

`EMPTY` represents the *empty* clause, *i.e.* a contradiction. In that case, unit propagation concludes the proof. `TRUE` represents a redundant clause that will be dropped. `UNIT l` is a unit clause to be propagated and `CLAUSE wc` is a watched clause.

Without loss of generality, suppose that $\lfloor f \rfloor$ belongs to the set of assigned literals `s` and that we have a clause c of the form:

$$c = \{|\texttt{watch1} := \neg\lfloor f \rfloor; \texttt{watch2} := x; \texttt{unwatched} := [y_1; \ldots; y_n;]|\}$$

The `reduce` function takes as input the watched literal x and aims at finding in the list $[y_1; y_n; \ldots; y_n]$ a watched literal y_i (*i.e.*, y_i is not assigned in `s`) and return as clause the rest of the list.

```
reduce s d x [] = {| elt := UNIT x ; deps := d |}
reduce s d x (y::l) = {| elt := TRUE ; deps := d |} when y ∈ s
reduce s d x (y::l) = reduce s ((deps (s y)) ∪ d) l when ¬ y ∈ s
reduce s d x (y::l) =
  let wc := {| watch1 := x ; watch2 := y ; unwatched := l |} in
  {| elt := CLAUSE wc ; deps := d |} when y ∉ s
```

If the list is empty, we produce the unit clause `UNIT` x and unit propagation will be recursively called for x. Let y be the head of the list l. If y is not assigned in s, we return a novel clause where the watched literals are x and y. If y is already assigned with the same polarity in s ($y \in s$), the clause is redundant and can be dropped. If y is already assigned but with opposite polarity ($\neg y \in$ `s`), the reduction is recursively called threading along the dependencies of the literal y.

▶ **Example 4.** Suppose that the set of assigned literals is given by `s` $= \{f; \neg y_1; \neg y_n\}$ and we perform unit propagation over the clause c. By construction, we know that neither x nor $\neg x$ are assigned in `s`. To get a well-defined watched clause with 2 watched literals, we need to find a replacement for the literal $\neg\lfloor f \rfloor$ in the list $[y_1; y_2; \ldots; y_n]$. As $\{\neg y_1, \neg y_n\} \subseteq$ `s`, a greedy unit propagation would deduce that y_1 and y_n can be removed but as the prohibitive cost of a linear scan of the whole clause. The idea of lazy unit propagation is to stop at the first unassigned literal *i.e.* y_2. Therefore, we generate the watched clause

$$\{\texttt{watche1} := x; \texttt{watche2} := y_2; \texttt{unwatched} := [y_3; \ldots; y_n]\}$$

2.4 Case Splitting and Backjumping

When all the unit clauses are propagated, the solver performs a case-split over a clause. The clause needs to represent a disjunction. If the conclusion is `False`, any clause may be selected. Otherwise, the clause may only contain either positive literals or negative *classical* literals. The soundness of this argument is expressed by the following inferences.

$$\frac{\bigwedge_i \Gamma \cup (p_i \to \bot) \vdash \bot \quad \bigwedge_j \Gamma \cup q_j \vdash \bot}{\Gamma \cup \bigwedge_i p_i \to \bigvee_j q_j \vdash \bot} \qquad \frac{\bigwedge_i (p_i \vee \neg p_i) \quad \bigwedge_i \Gamma \cup (p_i \to \bot) \vdash g \quad \bigwedge_j \Gamma \cup q_j \vdash g}{\Gamma \cup \bigwedge_i p_i \to \bigvee_j q_j \vdash g}$$

A naive algorithm consists in doing a recursive call for each literal, say l_i, of the clause. However, if for one of the cases, the proof does not depend on the literal l_i, it is sound to ignore the other cases and immediately return (*backjump*) to a previous case-split. This is illustrated by Example 5.

▶ **Example 5** (Backjumping)**.** Consider the following goal where each clause H_i is tagged with a set of dependencies d_i where the d_i are disjoint and do not contain the literals $\{l, m, n, o\}$.

$$\{H_1 : (l \vee m)_{d_1}, H_2 : (n \vee o)_{d_2}, H_3 : (n \to \bot)_{d_3}, H_4 : (o \to \bot)_{d_4}\} \vdash \bot$$

Suppose that we first perform a case-split over H_1. For the first case, we introduce the unit clause $H_5 : l_{\{l\}}$ with a singleton dependency *i.e.*, the literal only depends on itself. As no unit propagation is possible, we perform another case split over $H_2 : (n \vee o)_{d_2}$. For the case n, using H_3, we derive the empty clause $\bot_{d_3 \cup \{n\}}$ and for the case o, we derive the empty clause $\bot_{d_4 \cup \{o\}}$. Therefore, gathering both sub-cases, we have $\bot_{d_2 \cup d_3 \cup d_4}$. As $l \notin d_2 \cup d_3 \cup d_4$, the case-split over H_1 is irrelevant for the proof and, therefore, there is no need to explore the second case of the case-split over H_2.

A propositional prover has the idealised type `ProverT` defined below.

```
ProverT := state → HFormula → option LitSet.t
```

It takes as input the prover state `st`, and the formula to prove `g` and returns, upon success, the set of literals that are needed for the proof. Details about the components of `state` are given in Section 2.7. The `case_split` algorithm is parametrised by a prover `Prover:ProverT`. It takes as input a clause `cl` and returns a prover performing a case-analysis over all the literals of the input clause `cl`. In addition to a set of literals, it returns a boolean indicating whether backjumping is possible.

```
Fixpoint case_split cl st g  :=
 match cl with
 | [] => Some (false, LitSet.empty)
 | f::cl => match Prover (st ∪ f) g with
           | None => None
           | Some d  =>
              if f∉ d && st ⊢ d then Some (true,d)
              else match case_split cl st g with
                   | None => None
                   | Some(b, d') => if b then Some(b,d')
                                   else Some(false, d' ∪ (d \ {f}))
                   end end end.
```

If the clause is empty *i.e.*, it denotes `False`, the proof is finished. Suppose that `f` is the first literal of the clause `cl`, the prover is recursively called with the state `st` augmented with the unit clause f. If the proof `d` does not require `f` and all the literals in `d` are assigned in the current state (`st` ⊢ d), the whole case-split is spurious and the prover returns immediately. Otherwise, we recursively call `case_split` and return the updated set of literals needed by the proof. The literal `f` is momentarily removed from the dependencies of `d` but the dependencies are adjusted just after the `case_split` call by adding the dependencies of the clause `cl`.

2.5 Implication clauses

When there is no clause to branch over, the prover considers *implication clauses* and tries to (recursively) prove one of them.

```
Fixpoint prover_arrows (l : list literal) (st: state) (g: HFormula)  :=
 match l with
 | [] => None
 | f :: l => match Prover st f with
             | Some _ => Prover (st ∪ f) g  | None => prover_arrows l st g
             end  end.
```

The function `prover_arrows` takes as argument a list `l` of literals. A literal `f` in the list is of the form `POS(LIMPL` $[a_1;...;a_n]$ b`)` which encodes an *implication clause* $\lfloor a_1 \rfloor \to \cdots \to \lfloor a_n \rfloor \to (\lfloor a_1 \to \cdots \to a_n \to b \rfloor)$ which was left aside during the CNF conversion. The prover is recursively called with, as goal, the formula $\bigwedge_i a_i \to b$. If the proof succeeds, the literal `f` holds and the proof continues with a state augmented with `f`. Otherwise, the prover tries another literal.

2.6 Theory Reasoning

At leaves of the proof search, the solver has assigned a set of literals that do not lead to a propositional conflict. At this stage, a SAT solver reports that a model is found. Yet, this propositional model may be invalidated by performing theory reasoning. Our solver is parametrised by a theory reasoner. Essentially, it takes as input a list of literals and returns (upon success) a clause. The clause is a tautology of the theory that is added to the clauses of the SAT solver. The interface of a theory reasoner is given below.

```
Record Thy := {
 thy_prover : hmap → list literal → option (hmap * clause);
 thy_prover_sound : ∀ hm hm' cl cl', thy_prover hm cl = Some (hm',cl')
    → ⟦cl'⟧  ∧ hm ⊑ hm' ∧ ∀ l ∈ cl', l ∈ hm'  }
```

A `thy_prover` takes as input a hash-cons map `hm` and a list of literals `cl`. The hash-cons map `hm` contains the currently hash-consed terms. The literals in the list `cl` are obtained from the current state of the SAT solver and are restricted to atomic formulae *i.e.*, `LAT i` for some `i`. The theory prover may either fail to make progress or return an updated hash-cons map `hm'` and a clause `cl'` such that

i) the clause `cl'` holds;
ii) the literals in `cl'` are correctly hash-consed in `hm'`;
iii) the updated hash-cons map `hm'` contains more hash-consed formulae than `hm`.

Typically, the clause `cl'` is a conflict clause and therefore the literals of `cl'` are including in those of `cl`. However, we open the possibility to perform theory propagation and generate a clause made of new literals.

2.7 Solver State and Main Loop

Using the previous components, we are ready to detail the proof state.

```
Record state := {
 fresh_clause_id : int; hconsmap : hmap; arrows : list literal;
 wneg            : iSet; defs : iSet * iSet;
 units           : ptrie (Annot bool); unit_stack : list (Annot literal);
 clauses         : watch_map }.
```

- The field `fresh_clause_id` is a fresh index that is incremented each time a novel clause is created.

- The field `hconsmap` is only updated by the theory prover and is used to ensure well-formedness conditions about hash-consing.
- The field `arrows` contains a list of literals. Each literal is of the form $\texttt{IMPL}[a_1; \ldots; a_n]b$ and represent an *implication clauses i.e.*, $(\lfloor a_1 \rfloor \to \cdots \to \lfloor a_i \rfloor \to \lfloor b \rfloor) \to Lit a_1 \to \cdots \to a_i \to b$ which could not be turned into a proper *flat* clause during CNF conversion.
- The field `wneg` contains a set of hash-cons indexes corresponding to watched negative literals. These literals are added to the list of literals sent to the theory prover.
- The field `defs` is a pair of sets of hash-cons indexes. These are used for memoizing the CNF+ and CNF- computations.
- The field `units` encodes using a Patricia Tree the set of assigned literals. The boolean indicates whether the literal is positive or negative and the annotation tells which sets of initial literals were needed for the deduction.
- The field `unit_stack` is the stack of literals for which unit propagation needs to be run.
- The field `clauses` contains the indexed `watched_clauses`.

The type of the implemented prover is slightly more complicated that what we explained in the Section 2.4. In addition to a set of literals, it also threads along a `hmap` and a list of clauses learnt by theory reasoning. Theory reasoning is only running if the boolean `use_prover` is set. Termination is ensured by provided some fuel computed (without proof) from the size of the formula.

```
Fixpoint prover thy use_prover fuel st g :=
 match n with
 | O => Fail OutOfFuel
 | S n => let ProverRec := prover thy use_prover n in
   (prover_unit n ; prover_case_split ProverRec ;
    prover_impl_arrows ProverRec ; prover_thy ProverRec thy use_prover) st g
 end.
```

If the `prover` does not run out of fuel, it calls in sequence the different provers described in the previous sections. After performing *unit propagation*, it performs a case-split and recursively calls the prover. If no case-split is possible, it tries to prove one of the *implication clause* by recursively calling the prover. Eventually, if no *implication clause* can be derived, theory reasoning is called.

▶ **Example 6** (Example 3 continued). Suppose that we have the proof state obtained after constructed the CNF for $a \to (b \wedge c) \to (d \vee (a \to c))$ (see Example 3). The literal $\lfloor a \to (b \wedge c) \rfloor$ triggers unit propagation and generates the clause $\lfloor a \rfloor \to \lfloor b \wedge c \rfloor$. At this stage, neither unit propagation nor case-splitting is possible. Yet, we have a single *implication clause* $(\lfloor a \rfloor \to \lfloor c \rfloor) \to \lfloor a \to c \rfloor$. Hence, we call `prover_arrows` with the singleton list $[\lfloor a \to c \rfloor]$. To prove $\lfloor a \to c \rfloor$, we introduce $\lfloor a \rfloor$ and attempt to prove c. By unit propagation, we derive first $\lfloor b \wedge c \rfloor$ and then $\lfloor b \rfloor$ and $\lfloor c \rfloor$; thus concluding the sub-proof. As a result, we augment the context with $\lfloor a \to c \rfloor$ and conclude the goal by unit propagation.

3 Soundness Proof

In this part, we give some insights about the soundness proof that is based on three main properties: well-formedness, soundness of dependencies and soundness of provers. The only axioms of the development are those of the `Int63` standard library which define native machine integers.

3.1 Well-formedness

Hash-consing has the advantage that the equality of terms can be decided by an equality of native integers without in-depth inspection of terms. However, this requires ensuring that the initial formula is correctly hash-consed and that the prover always operates with hash-consed literals. Fortunately, most of the generated literals are sub-formulae that are obtained by the CNF. The only exception is theory reasoning. The hash-consed formulae are stored in a map `m : hmap := ptrie (bool*LForm)`. The keys of the map are the hash-cons indexes and the boolean indicates whether the formula is a *classical* proposition. The set of hash-consed formulae `has_form m` is inductively defined below.

$$\frac{m(i) = (true, \texttt{LFF})}{\texttt{LFF}_i^{true} \in \texttt{has\_form}\ m} \qquad \frac{m(i) = (b, \texttt{LAT i}) \quad b \leftrightarrow [\![\texttt{LAT}\ i]\!] \vee \neg [\![\texttt{LAT}\ i]\!]}{(\texttt{LAT i})_i^b \in \texttt{has\_form}\ m}$$

$$\frac{\begin{array}{c} m(i) = (b, \texttt{LOP}\ o\ [\_^{b_1}_{i_1}; \ldots; \_^{b_n}_{i_n}]) \\ (f_1)^{b_1}_{i_1} \in \texttt{has\_form}\ m \quad \ldots \quad (f_n)^{b_n}_{i_n} \in \texttt{has\_form}\ m \\ b \leftrightarrow b_1 \wedge \cdots \wedge b_n \end{array}}{(\texttt{LOP}\ o\ [(f_1)^{b_1}_{i_1}; \ldots; (f_n)^{b_n}_{i_n}])^b_i \in \texttt{has\_form}\ m}$$

$$\frac{\begin{array}{c} m(i) = (b, \texttt{LIMPL}[\_^{b_1}_{i_1}; \ldots; \_^{b_n}_{i_n}]\ \_^{b_0}_{i_0}) \\ (f_0)^{b_0}_{i_0} \in \texttt{has\_form}\ m \quad \ldots \quad (f_n)^{b_n}_{i_n} \in \texttt{has\_form}\ m \\ b \leftrightarrow b_0 \wedge \cdots \wedge b_n \end{array}}{(\texttt{LIMP}[(f_1)^{b_1}_{i_1}; \ldots; (f_n)^{b_n}_{i_n}]\ (f_0)^{b_0}_{i_0})^b_i \in \texttt{has\_form}\ m}$$

Essentially, this consists in checking that all the sub-formulae are stored within the hmap `m` when only considering the top constructor and the hash-cons index of the sub-formulae. The advantage of this formulation is that $f_i^b \in \texttt{hash\_form}\ m$ can be checked algorithmically by a linear pass over the formula f. This definition also entails that formulae with the same hash-cons index are the same.

$$f_i^{b_1} \in \texttt{has\_form}\ m \wedge g_i^{b_2} \in \texttt{has\_form}\ m \rightarrow f_i^{b_1} = g_i^{b_2}$$

Our well-formedness conditions state that a solver state `st:state` only contains hash-consed formulae. In particular, all the literals in the clauses are hash-consed formulae. As Patricia Trees come with structural invariants, every Patricia Tree needs to be well-formed. In the prover state, the set of assigned literals (see the `units` field) is represented by a Patricia Tree `ptrie (Annot.t bool)` where the keys are hash-consed indexes. In this case, the well-formedness conditions requires that there exists a hash-consed formula corresponding to this index. As all the formulae with the same hash-cons index are equal this identifies a unique formula.

$$\texttt{wf\_units\_lit}\ u\ m = \forall i, v, u(i) = v \rightarrow \exists f_i^b, \texttt{has\_form}\ m\ f_i^b$$

The proofs of preservation are compositional and do not pose any particular issue.

3.2 Soundness of Dependencies

The logical objects in the proof state are the set of indexed watched clauses (`clauses`) , the set of assigned literals (`units`) and the stack of literals that are yet to be unit-propagated (`unit_stack`). Each clause (or literal) is also annotated by the set of literals needed for the deduction. These sets are represented by Patricia Trees using hash-consed indexes as keys.

We write c_d (resp. l_d) for a (watched) clause (resp. literal) annotated with a set of literals d. For each operation, we prove that the annotation is sound *i.e.* the conjunction of the literals in d entails the clause c (resp. the literal l). The interpretation is indexed by a hash-cons map m linking indexes to hash-consed formulae.

$$[\![c_{\{l_1,\dots,l_n\}}]\!]_m = [\![l_1]\!]_m \land \dots [\![l_n]\!]_m \to [\![c]\!]$$

A literal l is introduced by either the introduction rule or a case-split. In that case, the set d is the singleton $\{l\}$ and $[\![l_{\{l\}}]\!]$ holds. New clauses are obtained by unit propagation and the soundness of the deduced clauses is obtained from the following deduction rules:

$$\frac{d_1 \to p \qquad d_2 \to (p \to p_1 \to \dots p_n \to q_1 \lor \dots q_n)}{d_1 \land d_2 \to (p_1 \to \dots p_n \to q_1 \lor \dots q_n)} \qquad \frac{d_1 \to \neg q \qquad d_2 \to (q \lor q_1 \lor \dots q_n)}{d_1 \land d_2 \to (q_1 \lor \dots q_n)}$$

Syntactically, the conjunction $d_1 \land d_2$ is modelled by the union $d_1 \cup d_2$.

$$[\![d_1 \cup d_2]\!]_m = [\![d_1]\!]_m \land [\![d_1]\!]_m$$

Given a prover state `st`, $[\![\texttt{st}]\!]^{dep}$ holds if all the clauses (resp. literals) in the state have correct dependencies. The proofs that the dependencies are correct are also compositional but require well-formedness conditions to hold.

3.3 Soundness of Provers

Upon success, a prover `p:ProverT` returns a set of valid learnt clause, an updated hash-cons map but also a set of literals that are sufficient to entail a conflict. The soundness of provers is then stated by the following definition.

```
Definition sound_prover (prover: ProverT) (st: state) :=
 ∀ g hm lc d, wf_state st → $g_i^b$ ∈ hconsmap st →
 prover st g = Success (m,lc,d) →
 ([[st]] → [[st]]$^{dep}$ → [[g]]) ∧ ([[st]]$^{dep}$ → [[d]]$_m$ → [[g]])  ∧ $\bigwedge_{c\in lc}$ [[c]]
```

Therefore, soundness requires to prove that the goal formula `g` is entailed by either the clauses in the state and their dependencies or the dependencies alone and the set of literals d. Both properties are needed in order to prove that *backjumping* is correct. We also have to prove that the clauses `lc` obtained by theory reasoning are sound. Sometimes, the fact that we use classical reasoning in an intuitionistic context is a source of complication for the proof because this prevents a direct forward reasoning. For instance, in a pure classical context, the CNF of a formula f generates tautologies and the correctness can be directly stated by

$$\forall cl, cl \in \texttt{cnf}\ f \to [\![cl]\!]$$

The correctness of our CNF depends on the goal formula. For instance, if the conclusion is `False`, De Morgan laws are admissible. Therefore, our formulation is the following.

$$((\forall cl, cl \in \texttt{cnf}\ f \to [\![cl]\!]) \to [\![g]\!]) \to [\![g]\!]$$

Note that if $g = \texttt{False}$, we get a double negation. For most of our proof state transformation, say `T`, a simplified correctness lemma has the form

$$\texttt{wf\_state}\ st\ \to ([\![\texttt{T}\ st]\!] \to [\![g]\!]) \to ([\![st]\!] \to [\![g]\!])$$

4 Proof by Hybrid Reflection

Having a SAT solver for deciding propositional logic, one option is to directly perform proof by reflection [9, Chap. 16]. Using this mode, we get a tactic that is similar to the existing `tauto` with the advantage that our reification process detects decidable propositions declared using the type-class mechanism [25].

In order to perform theory reasoning, the proof by reflection approach demands that we implement the interface of Section 2.6. However, a closer look shows that the interface does not expose a syntax for atoms: they are only represented by indexes. In a nutshell, the interface is purposely an empty shell. A fully reflexive approach would require to enrich the interface with an environment mapping atoms to theory-specific syntactic terms. Here, we follow a different path and implement theory reasoning using classic user-defined tactics. The advantages are that existing decision procedures can be readily reused and that user-defined tactics are more flexible. Yet, user-defined tactics and proof by reflection do not operate at the same level and embedding a powerful reflective tactic language would be a challenge by itself.

To reach our goal, we take the opposite approach and leverage the Coq extraction mechanism, thus allowing to run our SAT solver inside the Coq proof engine, *i.e.*, at the level of tactics. There, the theory reasoning interface may be implemented by calling user-defined tactics. The theory prover `thy_prover` (see Section 2.6) takes a list of literals and attempts to produce a clause. Therefore, to interface with tactics, the followings tasks need to be performed:

1. Construct a goal from the literals provided by the SAT solver.
2. (Optionally) perform theory propagation
3. Run the tactic and obtain a proof-term.
4. Return a reduced clause.

Within the proof engine, we maintain a mapping from literals to actual Coq terms. We interpret the list of literal as a clause to be discharged by the user-provided tactic. We obtain a goal G of the general form: $G{:=}\forall_{i\in I}(x_i : p_i), \bigvee_{j\in J} q_j$. Actually, the p_i and the q_j do not depend on the x_i. Yet, this dependent product notation is convenient to explain our minimisation procedure.

The fact that the conclusion is a disjunction $\bigvee_{j\in J} q_j$ may trigger, for some tactics, a costly case-analysis which would be more efficiently performed at the level of the SAT solver. For this purpose, we leverage the type-class mechanism to perform theory propagation at the level of individual propositions. For instance, if one of the q_j is an equality over `Z`, say `x=y`, we directly return to the SAT solver the tautology clause `x<y` $\vee$ `x=y` $\vee$ `y<x`.

If the tactic fails, the unsuccessful goal G is a potential counter-example and is returned to the user for inspection. If the tactic proves the goal, the clause G holds and is therefore a conflict clause. A sound, but naive, approach is to return this clause containing all the input literals. Yet, to improve the efficiency of the SAT solver, it is desirable to reduce the clause and produce a minimal *unsatisfiable core*. In other words, we search for sets $I' \subseteq I$ and $J' \subseteq J$ such that $G'{:=}\forall_{i\in I'}(x_i : p_i), \bigvee_{j\in J'} q_j$ is still a tautology. In our context, if the tactic succeeds, it also produces a proof-term. Using a syntactic analysis, we track the subset of the x_i that are used in the proof-term and therefore obtain a set $I' \subseteq I$ containing the needed hypotheses. To minimise the set J, a finer grained analysis of the proof term may be possible. However, there is a sweet spot when the positive literals are classical propositions. In that case, the goal G is equivalent to G'

$$G'{:=}\forall_{i\in I}(x_i : p_i), \forall_{j\in J}(y_j : \neg p_j), \texttt{False}$$

Using this formulation, the same syntactic analysis extracts both a subset $I' \subseteq I$ of the negative literals and a subset $J' \subseteq J$ of the positive literals. When the propositions are not classical, we perform some partial iterative proof-search and try to prove each of the q_i, one at a time. Once the minimisation is done, we adjust the proof-term for the minimised conflict clause: this consists in recomputing the correct De Bruijn indexes to accommodate for the removed hypotheses.

If the extracted SAT solver succeeds, we have the guarantee that the goal is provable but the proof elements still need to be pieced together. The SAT solver returns the set of needed literals. We exploit this information to remove from the context the propositions that are irrelevant for the proof. Now, we enrich the context with all the conflict clauses that were generated during the SAT solver run, re-using the cached proof terms. At this stage, we have a goal that is a propositional tautology. It is solved by re-running the SAT solver, without theory reasoning, by a classic proof by reflection.

Using the extracted SAT solver has the advantage that the possible bugs are limited to the interface with user-provided tactic. Another possible design would be to rely on an external untrusted (intuitionistic) SAT solver. This would have some speed advantage for the generation of conflict clauses but would increase the code base. In our case, we reuse the same verified component both in the Coq proof engine and to perform proof by reflection.

5 Experiments

Before showing some larger scale experiments, we come back to the motivating examples (see Section 1.1) and explain how they are solved by our tactic.

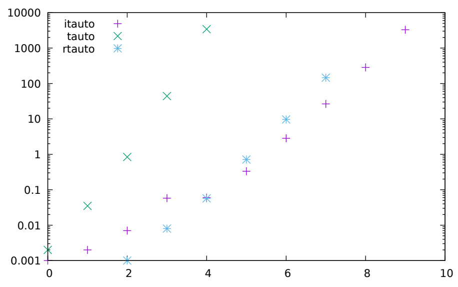
5.1 Back to Motivating Examples

Consider again the goals in Example 1. For `Ex1`, the SAT solver has knowledge that `x>=0` is a classical proposition and therefore performs a case-split over `x>=0→ p`. The first sub-case is solved by theory reasoning (`x=0 → ~ x>= 0 → False` holds); and the second by propositional reasoning (`p → p`). For `Ex2`, the SAT solver does not make progress. However, it asks the leaf tactic `congruence` whether a watched negative literal (*i.e.*, `a=c`) may be deduced from the assigned literal (*i.e.*, `a=b` and `b=c`). As `a=b → b=c → a=c` can be proved by `congruence`, the clause is asserted by theory reasoning. The proof follows by propositional reasoning.

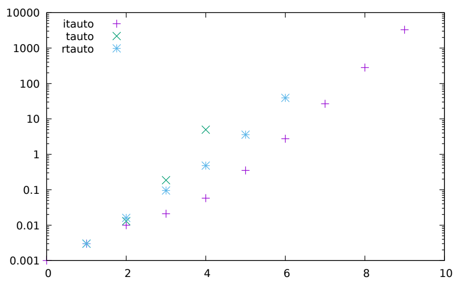
For Example 2, the goal is solved by `intuition congruence` using two identical calls to the leaf tactic `congruence`. Because our SAT solver threads learnt clauses along the computation, the same goal is solved by `itauto congruence` by calling `congruence` only once.

5.2 Pigeon Hole

The Pigeon Hole Principle, stating that there is no way to fit n+1 pigeons in n holes, is a hard problem for resolution based SAT solvers. The algorithm presented here is no exception and the running time will be exponential in n. This is however a useful benchmark for assessing scalability. We have benchmarked our `itauto` tactic against the existing `tauto` and `rtauto` tactics using a timeout of 3600s and a memory limit of 15GB on a laptop (Intel Core i7 at 1.8GHz with 32GB of RAM). The results are shown in Fig. 1. For `itauto`, the running time of the tactic and the type-checking time (`Qed` time) are similar. As we perform a pure proof by reflection, this is not surprising. Though `itauto` scales slightly better, the running time



(a) Running time of tactics.



(b) Running time of type-checking.

Figure 1 Pigeon Hole for `itauto`, `tauto` and `rtauto`.

are similar to `rtauto` which only performs proof by reflection to check certificates. `tauto` is the least scalable and the proof search reaches timeout for 5 pigeons. `rtauto` exhausts its memory quota when checking its certificate for 7 pigeons. `itauto` reaches the time limit of 3600s for 10 pigeons.

5.3 On existing developments

We have benchmarked `itauto` against the existing Coq tactics `tauto` and `intuition` for the Bedrock2[9] and CompCert[10] developments using Coq 8.14+alpha. We have replaced calls to `tauto` and `intuition tac` for `tac` $\in$ {`idtac`, `assumption`, `discriminate`, `congruence`, `lia`, `auto`, `eauto` }. We have ruled out calls to `intuition` when they generate sub-goals. In terms of completeness, `itauto` is able to solve the vast majority of the goals. One representative case of failure is given in Example 7.

▶ **Example 7.** The following goal is solved by `intuition congruence`.

```
Goal true = true ↔ (Z → (False ↔ False)).
```

Yet, `itauto congruence` fails because `Z` (*i.e.*, the type of integers) has type `Set` and therefore the whole expression `Z → (False ↔ False)` is reified as an opaque atomic proposition. Our solution is to call `itauto` recursively *i.e.*, `itauto (itauto congruence)` so that an hypothesis `x:Z` is explicitly introduced. The same approach works for goals with inner universal quantifiers.

For a few instances, we also strengthen the leaf tactic replacing *e.g.*, `idtac` by `reflexivity`, and, `discriminate` by `congruence`. This is due to the fact that `intuition` implicitly calls `reflexivity` and that our handling of theory reasoning sometimes introduces negations which fool the `discriminate` tactic.

After modification, Bedrock2 performs 1621 calls to `itauto`. For 40% of the goals, the running time differs by less than 1ms and `itauto` outperforms the historic tactics for 40% of the cases. Overall, `itauto` is faster and there is an slight speedup of 1.07. Yet, all the calls are quickly solved; the slowest goal is solved in 0.18s.

CompCert performs 924 calls to `itauto`. For 76% of the goals, the running time also differs by less than 1ms. `itauto` outperforms the historic tactics for 19% percent of the goals.

[9] https://github.com/mit-plv/bedrock2
[10] https://github.com/AbsInt/CompCert

Yet, when `itauto` is faster it can be by several orders of magnitudes: the 19% percent of goals are solved more than 20 times faster by `itauto`. For instance, the maximum running time of `tauto` is 5.26s to be compared to 0.81s for `itauto`. Overall, `itauto` performs better with a speedup of 2.8 for solving all the goals.

In summary, the results are rather positive though the advantage may be slim. It seems that `itauto` shows a decisive advantage for goals that are very slow with the existing tactics. As shown by the Pigeon Hole experiment, this would indicate a better scalability. It would not be surprising for `itauto` to be slower on simple goals where the overhead of setting up the proof by reflection cannot be amortised. `itauto` spends time in different proof tasks: SAT solver, theory reasoning and proof by reflection. We are confident that the SAT solver is scalable and reasonably fast. Yet, this is not always the bottleneck, and, the positive results reported here were only made possible by fine tuning other tasks *e.g.*, the reification code.

6 Related Work

For propositional logic, Weber and Amjad [27] for HOL theorem provers and Armand *et al.* [3] for Coq show how to efficiently validate resolution proofs that can be generated by modern SAT solvers using Conflict Driven Clause Learning (*e.g.* zChaff [21]). Satisfiability Modulo Theory (SMT) solvers (*e.g.*, veriT [8], Z3 [12] , CVC4 [13]) produce proof artefacts. Böhme and Weber [7] show how to efficiently perform proof reconstruction for HOL provers. Armand *et al.* [2] and Besson *et al.* [4] extract proof certificates from SMT proofs that are validated in Coq. Sledgehammer [5] interfaces Isabelle/HOL with a variety of provers; Metis [18, 23] is in charge of performing proof reconstruction. We follow a different approach and verify a SAT solver interfaced with the tactics of Coq. What we loose in efficiency, we gain in flexibility because the user might use her own fine-tuned domain specific tactics.

Claessen and Rosén [10] build a prover for intuitionistic logic on top of a black-box SAT solver. Their implementation is more efficient than ours but is not integrated inside a proof-assistant. Lescuyer and Conchon [19, 20] formalise a reflexive SAT solver for classical logic. Their implementation features a lazy definitional CNF that is not fully exploited because of the lack of hash-consing. Compared to ours, their SAT solver performs neither lazy unit propagation nor backjumping. Blanchette *et al.* [6] formalise a Conflict Driven Clause Learning (CDCL) SAT solver and derive a verified implementation. Using this framework, Fleury, Blanchette and Lamich [15] derive an imperative implementation using watched literals. Our implementation has less sophisticated features but is integrating as a reflective proof procedure and allows to perform theory reasoning.

7 Conclusion and Future Work

We have presented `itauto` a reflexive tactic for intuitionistic propositional logic which can be parametrised by user-provided tactics. The SAT solver is optimised to leverage classical reasoning thus limiting, when possible, costly intuitionistic reasoning. Our SAT solver has several features found in modern SAT solver. Yet, the implementation could be improved further. A first improvement would be to generate more sophisticated learnt clauses. This could be done by attaching to each clause, not only the needed literals, but also the performed unit propagations. Another improvement would be to use primitive persistent arrays (available since Coq 8.13) allowing to implement efficiently more imperative algorithms *e.g.*, 2-watched literals.

References

1 John Allen. *Anatomy of LISP*. McGraw-Hill, Inc., USA, 1978.

2 Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In *CPP*, volume 7086 of *LNCS*, pages 135–150. Springer, 2011. `doi:10.1007/978-3-642-25379-9_12`.

3 Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. Extending Coq with Imperative Features and Its Application to SAT Verification. In *ITP*, volume 6172 of *LNCS*, pages 83–98. Springer, 2010. `doi:10.1007/978-3-642-14052-5_8`.

4 Frédéric Besson, Pierre-Emmanuel Cornilleau, and David Pichardie. Modular SMT Proofs for Fast Reflexive Checking Inside Coq. In *CPP*, volume 7086 of *LNCS*, pages 151–166. Springer, 2011. `doi:10.1007/978-3-642-25379-9_13`.

5 Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending Sledgehammer with SMT Solvers. *J. Autom. Reason.*, 51(1):109–128, 2013. `doi:10.1007/s10817-013-9278-5`.

6 Jasmin Christian Blanchette, Mathias Fleury, Peter Lammich, and Christoph Weidenbach. A verified SAT Solver Framework with Learn, Forget, Restart, and Incrementality. *J. Autom. Reason.*, 61(1-4):333–365, 2018. `doi:10.1007/s10817-018-9455-7`.

7 Sascha Böhme and Tjark Weber. Fast LCF-Style Proof Reconstruction for Z3. In *ITP*, volume 6172 of *LNCS*, pages 179–194. Springer, 2010. `doi:10.1007/978-3-642-14052-5_14`.

8 Thomas Bouton, Diego Caminha Barbosa De Oliveira, David Déharbe, and Pascal Fontaine. veriT: An Open, Trustable and Efficient SMT-Solver. In *CADE*, volume 5663 of *LNCS*, pages 151–156. Springer, 2009. `doi:10.1007/978-3-642-02959-2_12`.

9 Pierre Castéran and Yves Bertot. *Interactive theorem proving and program development. Coq'Art: The Calculus of inductive constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004. URL: `https://hal.archives-ouvertes.fr/hal-00344237`.

10 Koen Claessen and Dan Rosén. SAT Modulo Intuitionistic Implications. In *LPAR-20*, volume 9450 of *LNCS*, pages 622–637. Springer, 2015. `doi:10.1007/978-3-662-48899-7_43`.

11 Pierre Corbineau. Deciding Equality in the Constructor Theory. In *TYPES*, volume 4502 of *LNCS*, pages 78–92. Springer, 2006. `doi:10.1007/978-3-540-74464-1_6`.

12 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. `doi:10.1007/978-3-540-78800-3_24`.

13 Morgan Deters, Andrew Reynolds, Tim King, Clark W. Barrett, and Cesare Tinelli. A tour of CVC4: How it works, and how to use it. In *FMCAD*, page 7. IEEE, 2014. `doi:10.1109/FMCAD.2014.6987586`.

14 Roy Dyckhoff. Contraction-Free Sequent Calculi for Intuitionistic Logic. *J. Symb. Log.*, 57(3):795–807, 1992. `doi:10.2307/2275431`.

15 Mathias Fleury, Jasmin Christian Blanchette, and Peter Lammich. A verified SAT solver with watched literals using imperative HOL. In *CPP*, pages 158–171. ACM, 2018. `doi:10.1145/3167080`.

16 Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast Decision Procedures. In *CAV*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004. `doi:10.1007/978-3-540-27813-9_14`.

17 John Harrison and Laurent Théry. A Skeptic's Approach to Combining HOL and Maple. *J. Autom. Reason.*, 21(3):279–294, 1998. `doi:10.1023/A:1006023127567`.

18 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*, pages 56–68, 2003.

19 Stéphane Lescuyer and Sylvain Conchon. A Reflexive Formalization of a SAT Solver in Coq. In *Emerging Trends of TPHOLs*, 2008.

20 Stéphane Lescuyer and Sylvain Conchon. Improving Coq Propositional Reasoning Using a Lazy CNF Conversion Scheme. In *FroCoS*, volume 5749 of *LNCS*, pages 287–303. Springer, 2009. `doi:10.1007/978-3-642-04222-5_18`.

21 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC*, pages 530–535. ACM, 2001. `doi:10.1145/378239.379017`.

22 Chris Okasaki and Andrew Gill. Fast mergeable integer maps. In *Workshop on ML*, pages 77–86, 1998.

23 Lawrence C. Paulson and Kong Woei Susanto. Source-Level Proof Reconstruction for Interactive Theorem Proving. In *TPHOLs*, volume 4732 of *LNCS*, pages 232–245. Springer, 2007. `doi:10.1007/978-3-540-74591-4_18`.

24 David A. Plaisted and Steven Greenbaum. A Structure-Preserving Clause Form Translation. *J. Symb. Comput.*, 2(3):293–304, 1986. `doi:10.1016/S0747-7171(86)80028-1`.

25 Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In *TPHOLs*, volume 5170 of *LNCS*, pages 278–293. Springer, 2008. `doi:10.1007/978-3-540-71067-7_23`.

26 G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer, 1983. `doi:10.1007/978-3-642-81955-1_28`.

27 Tjark Weber and Hasan Amjad. Efficiently checking propositional refutations in HOL theorem provers. *J. Appl. Log.*, 7(1):26–40, 2009. `doi:10.1016/j.jal.2007.07.003`.

28 Hantao Zhang and Mark E. Stickel. An Efficient Algorithm for Unit Propagation. In *AI-MATH*, pages 166–169, 1996.

29 Hantao Zhang and Mark E. Stickel. Implementing the Davis-Putnam Method. *J. Autom. Reason.*, 24(1/2):277–296, 2000. `doi:10.1023/A:1006351428454`.

Verified Progress Tracking for Timely Dataflow

Matthias Brun
Department of Computer Science, ETH Zürich, Switzerland

Sára Decova
Department of Computer Science, ETH Zürich, Switzerland

Andrea Lattuada
Department of Computer Science, ETH Zürich, Switzerland

Dmitriy Traytel
Department of Computer Science, University of Copenhagen, Denmark

Abstract

Large-scale stream processing systems often follow the dataflow paradigm, which enforces a program structure that exposes a high degree of parallelism. The Timely Dataflow distributed system supports expressive cyclic dataflows for which it offers low-latency data- and pipeline-parallel stream processing. To achieve high expressiveness and performance, Timely Dataflow uses an intricate distributed protocol for tracking the computation's progress. We modeled the progress tracking protocol as a combination of two independent transition systems in the Isabelle/HOL proof assistant. We specified and verified the safety of the two components and of the combined protocol. To this end, we identified abstract assumptions on dataflow programs that are sufficient for safety and were not previously formalized.

2012 ACM Subject Classification Security and privacy → Logic and verification; Computing methodologies → Distributed algorithms; Software and its engineering → Data flow languages

Keywords and phrases safety, distributed systems, timely dataflow, Isabelle/HOL

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.10

Related Version Extended report that provides proof sketches and further formalization details.
Full Version: `https://www.github.com/matthias-brun/progress-tracking-formalization/raw/main/rep.pdf` [9]

Supplementary Material *Software (Isabelle Formalization)*: `https://www.isa-afp.org/entries/Progress_Tracking.html`

Funding Dmitriy Traytel is supported by a Novo Nordisk Fonden Start Package Grant (NNF20OC0063462). Andrea Lattuada is supported by a Google PhD Fellowship.

Acknowledgements We thank David Basin and Timothy Roscoe for supporting this work and Frank McSherry for providing valuable input on our formalization, e.g, by suggesting to consider the implied_frontier notion and to show that it is what local propagation computes. David Cock, Jon Howell, and Frank McSherry provided helpful feedback after reading early drafts of this paper.

1 Introduction

The dataflow programming model represents a program as a directed graph of interconnected operators that perform per-tuple data transformations. A message (an incoming datum) arrives at an input (a root of the dataflow) and flows along the graph's edges into operators. Each operator takes the message, processes it, and emits any resulting derived messages.

This model enables automatic and seamless parallelization of tasks on large multiprocessor systems and cluster-scale deployments. Many research-oriented and industry-grade systems have employed this model to describe a variety of large scale data analytics and processing tasks. Dataflow programming models with timestamp-based, fine-grained coordination, also called time-aware dataflow [24], incur significantly less intrinsic overhead [26].

In a time-aware dataflow system, all messages are associated with a timestamp, and operator instances need to know up-to-date (timestamp) *frontiers* – lower bounds on what timestamps may still appear as their inputs. When informed that all data for a range of timestamps has been delivered, an operator instance can complete the computation on input data for that range of timestamps, produce the resulting output, and retire those timestamps.

A *progress tracking mechanism* is a core component of the dataflow system. It receives information on outstanding timestamps from operator instances, exchanges this information with other system workers (cores, nodes) and disseminates up-to-date approximations of the frontiers to all operator instances.

The progress tracking mechanism must be correct. Incorrect approximations of the frontiers can result in subtle concurrency errors, which may only appear under certain load and deployment circumstances. In this work, we formally model in Isabelle/HOL and prove the safety of the progress tracking protocol of *Timely Dataflow* [1, 26, 27] (Section 2), a time-aware dataflow programming model and a state-of-the-art streaming, data-parallel, distributed data processor.

In Timely Dataflow's progress tracking, worker-local and distributed coordination are intertwined, and the formal model must account for this asymmetry. Individual agents (operator instances) on a worker generate coordination updates that have to be asynchronously exchanged with all other workers, and then propagated locally on the dataflow structure to provide local coordination information to all other operator instances.

This is an additional (worker-local) dimension in the specification when compared to well-known distributed coordination protocols, such as Paxos [21] and Raft [28], which focus on the interaction between symmetric communicating parties on different nodes. In contrast our environment model can be simpler, as progress tracking is designed to handle but not recover from fail-stop failures or unbounded pauses: upon crashes, unbounded stalls, or reset of a channel, the system stops without violating safety.

Abadi et al. [4] formalize and prove safety of the distributed exchange component of progress tracking in the TLA^{+} Proof System. We present their *clocks protocol* through the lens of our Isabelle re-formalization (Section 3) and show that it subtly fails to capture behaviors supported by Timely Dataflow [26, 27]. We then significantly extend the formalized protocol (Section 4) to faithfully model Timely Dataflow's modern reference implementation [1].

The above distributed protocol does not model the dataflow graph, operators, and timestamps within a worker. Thus, on its own it is insufficient to ensure that up-to-date frontiers are delivered to all operator instances. To this end, we formalize and prove the safety of the *local propagation* component (Section 5) of progress tracking, which computes and updates frontiers for all operator instances. Local propagation happens on a single worker, but operator instances act as independent asynchronous actors. For this reason, we also employ a state machine model for this component. Along the way, we identify sufficient criteria on dataflow graphs, that were previously not explicitly (or only partially) formulated for Timely Dataflow.

Finally, we combine the distributed component with local propagation (Section 6) and formalize the global safety property that connects initial timestamps to their effect on the operator frontier. Specifically, we prove that our combined protocol ensures that frontiers always constitute safe lower bounds on what timestamps may still appear on the operator inputs.

Related Work

Data management systems verification. Timely Dataflow is a system that supports low-latency, high-throughput data-processing applications. Higher level libraries [24, 25] and SQL abstractions [2] built on Timely Dataflow support high performance incremental view maintenance for complex queries over large datasets. Verification and formal methods efforts in the data management and processing space have focused on SQL and query-language semantics [6, 11, 13] and on query runtimes in database management systems [7, 23].

Distributed systems verification. Timely Dataflow is a distributed, concurrent system: our modeling and proof techniques are based on the widely accepted state machine model and refinement approach as used, e.g., in the TLA$^+$ Proof System [10] and Ironfleet [16]. Recent work focuses on proving consistency and safety properties of distributed storage systems [14, 15, 22] and providing tools for the implementation and verification of general distributed protocols [20, 31] leveraging domain-specific languages [30, 33] and advanced type systems [17].

Model of Timely Dataflow. Abadi and Isard [3] define abstractly the semantics of a Timely Dataflow programming model [26]. Our work is complementary; we concretely compute their *could-result-in* relation (Section 6) and formally model the implementation's core component.

2 Timely Dataflow and Progress Tracking

Our formal model follows the progress tracking protocol of the modern Rust implementation of Timely Dataflow [1]. The protocol has evolved from the one reported as part of the classic implementation Naiad [26]. Here, we provide an informal overview of the basic notions, for the purpose of supporting the presentation of our formal model and proofs.

Dataflow graph. A Timely Dataflow computation is represented by a graph of operators, connected by channels. Each worker in the system runs an instance of the entire dataflow graph. Each instance of an operator is responsible for a subset, or shard, of the data being processed. Workers run independently and only communicate through reliable message queues – they act as communicating sequential processes [18]. Each worker alternately executes the progress tracking protocol and the operator's processing logic. Figure 1 shows a Timely Dataflow operator and the related concepts described in this section.

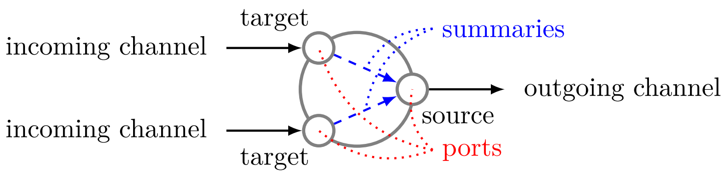


Figure 1 A Timely Dataflow operator.

Pointstamps. A pointstamp represents a datum at rest at an operator, or in motion on one of the channels. A pointstamp (l, t) refers to a location l in the dataflow and a timestamp t. Timestamps encode a semantic (causal) grouping of the data. For example, all data resulting from a single transaction can be associated with the same timestamp. Timestamps are usually tuples of positive integers, but can be of any type for which a partial order $\preceq$ is defined.

Locations and summaries. Each operator has an arbitrary number of input and output ports, which are locations. An operator instance receives new data through its input ports, or target locations, performs processing, and produces data through its output ports, or source locations. A dataflow channel is an edge from a source to a target. Internal

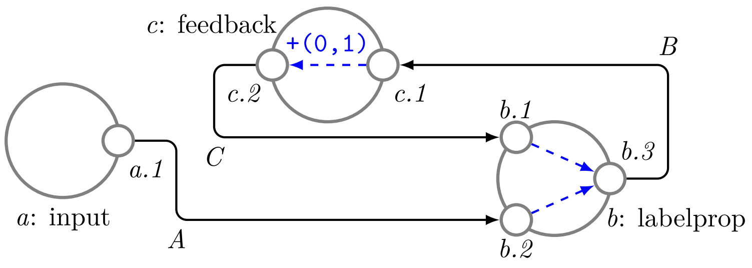


Figure 2 A timely dataflow that computes weakly connected components.

operator connections are edges from a target to a source, which are additionally described by one or more summaries: the minimal increment to timestamps applied to data processed by the operator.

Frontiers. Operator instances must be informed of which timestamps they may still receive from their incoming channels, to determine when they have a complete view of data associated with a certain timestamp. The progress tracking protocol tracks the system's pointstamps and summarizes them to one frontier per operator port. A frontier is a lower bound on the timestamps that may appear at the operator instance inputs. It is represented by an antichain F indicating that the operator may still receive any timestamp t for which $\exists t' \in F.\ t' \preceq t$.

Progress tracking. Progress tracking computes frontiers in two steps. A distributed component *exchanges* pointstamp changes (Sections 3 and 4) to construct an approximate, conservative view of all the pointstamps present in the system. Workers use this global view to locally *propagate* changes on the dataflow graph (Section 5) and update the frontiers at the operator input ports. The combined protocol (Section 6) asynchronously executes these two components.

▶ **Running Example** (Weakly Connected Components by Propagating Labels)**.** Figure 2 shows a dataflow that computes weakly connected components (WCC) by assigning integer labels to vertices in a graph, and propagating the lowest label seen so far by each vertex to all its neighbors. The input graph is initially sent by operator a as a stream of edges `(s,d)` with timestamp `(0,0)`. Each input port has an associated sharding function to determine which data should be sent to which operator instance: port *b.2* shards the incoming edges `(s,d)` by `s`.

The input operator a will continue sending additional edges in the graph as they appear, using increasing timestamps by incrementing one coordinate: `(1,0)`, `(2,0)`, etc. The computation is tasked with reacting to these changes and performing incremental re-computation to produce correct output for each of these input graph versions. The first timestamp coordinate represents logical consistency boundaries for the input and output of the program. We will use the second timestamp coordinate to track the progress of the unbounded iterative algorithm.

The operator a starts with a pointstamp (*a.1*, `(0,0)`) on port *a.1*, representing its intent to send data with that timestamp through the connected channel. When it sends messages on channel A, these are represented by pointstamps on the port *b.2*; e.g., (*b.2*, `(0,0)`) for the initial timestamp `(0,0)`. When it ceases sending data for a certain timestamp, e.g., `(0,0)`, operator a drops the corresponding pointstamp on port *a.1*. The frontier at *b.2* reflects whether pointstamps with a certain timestamp are present at either *a.1* or *b.2*: when they both become absent (when all messages are delivered) each instance of *b* notices that its frontier has advanced and determines it has received its entire share of the input (the graph) for a timestamp.

Each instance of *b* starts with a pointstamp on *b.3* at timestamp `(0,0)`; when it has received its entire share of the input, for each vertex with label `x` and each of its neighbors `n`, it sends `(n,x)` at timestamp `(0,0)`. This stream then traverses operator *c*, that increases the timestamp associated to each message by `(0,1)`, and reaches port *b.1*, which shards the incoming tuples `(n,x)` by `n`. Operator *b* inspects the frontier on *b.1* to determine when it has received all messages with timestamp `(0,1)`. These messages left *b.3* with timestamp `(0,0)`. The progress tracking mechanism will correctly report the frontier at *b.1* by taking into consideration the summary between *c.1* and *c.2*.

Operator *b* collects all label updates from *b.1* and, for those vertices that received a value that is smaller than the current label, it updates internal state and sends a new update via *b.3* with timestamp `(0,1)`. This process then repeats with increasing timestamps, `(0,2)`, `(0,3)`, etc., for each trip around the loop, until ultimately no new update message is generated on port *b.3* by any of the operator instances, for a certain family of timestamps (t_1, t_2) with a fixed t_1 corresponding to the input version being considered. Operator *b* determines it has correctly labeled all connected components for a given t_1 when the frontier at *b.1* does not contain a (t_1, t_2) such that $t_2 \preceq$ the graph's diameter. In practice, once operator *b* determines it has computed the output for a given t_1, the operator would also send the output on an additional outgoing channel to deliver it to the user. Later, operator *b* continues processing for further input versions, indicated by increasing t_1, with timestamps $(t_1,0)$, $(t_1,1)$, etc. ◀

3 The Clocks Protocol

In this section, we present Abadi et al.'s approach to modeling the distributed component of progress tracking [4], termed the *clocks protocol*. Instead of showing their TLA$^+$ Proof System formalization, we present our re-formalization of the protocol in Isabelle. Thereby, this section serves as an introduction to both the protocol and the relevant Isabelle constructs.

The clocks protocol is a distributed algorithm to track existing pointstamps in a dataflow. It models a finite set of workers. Each worker stores a (finite) multiset of pointstamps as seen from its perspective and shares updates to this information with all other workers. The protocol considers workers as black boxes, i.e., it does *not* model their dataflow graph, locations, and timestamps. We extend the protocol to take these components into account in Section 5.

In Isabelle, we use the type variable $'w :: \mathit{finite}$ to represent workers. We assume that $'w$ belongs to the *finite* type class, which assures that $'w$'s universe is finite. Similarly, we model pointstamps abstractly by $'p :: \mathit{order}$. The *order* type class assumes the existence of a partial order $\leqslant\ ::\ 'p \Rightarrow\ 'p \Rightarrow \mathit{bool}$ (and the corresponding strict order $<$).

We model the protocol as a transition system that acts on configurations given as follows:

record $('w :: \mathit{finite}, 'p :: \mathit{order})\ \mathit{conf} =$
  rec :: $'p\ \mathit{zmset}$
  msg :: $'w \Rightarrow\ 'w \Rightarrow\ 'p\ \mathit{zmset\ list}$
  temp :: $'w \Rightarrow\ 'p\ \mathit{zmset}$
  glob :: $'w \Rightarrow\ 'p\ \mathit{zmset}$

Here, rec c denotes the global multiset of pointstamps (or records) that are present in a system's configuration c. We use the type $'p\ \mathit{zmset}$ of *signed multisets* [8]. An element $M\ ::\ 'p\ \mathit{zmset}$ can be thought of as a function of type $'p \Rightarrow \mathit{int}$, which is non-zero only for finitely many values. (In contrast, an unsigned multiset M :: $'p\ mset$ corresponds to a function of type $'p \Rightarrow nat$.) Signed multisets enjoy nice algebraic properties; in particular, they form a group. This significantly simplifies the reasoning about subtraction. However, rec c will always store only non-negative pointstamp counts. The other components of a configuration c are

- the progress message queues msg $c\ w\ w'$, which denote the progress update messages sent from worker w to worker w' (not to be confused with data messages, which are accounted for in rec c but do not participate in the protocol otherwise);
- the temporary changes temp $c\ w$ in which worker w stores changes to pointstamps that it might need to communicate to other workers; and
- the local approximation glob $c\ w$ of rec c from the perspective of worker w (we use Abadi et al. [4]'s slightly misleading term glob for the worker's *local* view on the global state).

In contrast to rec, these components may contain a negative count $-i$ for a pointstamp p, which denotes that i occurrences of p have been discarded.

The following predicate characterizes the protocol's initial configurations. We write $\{\#\}_z$ for the empty signed multiset and $M\ \#_z\ p$ for the count of pointstamp p in a signed multiset M.

definition Init :: $('w, 'p)\ conf \Rightarrow bool$ **where**
  Init $c = (\forall p.\ \text{rec}\ c\ \#_z\ p \geqslant 0) \wedge (\forall w\ w'.\ \text{msg}\ c\ w\ w' = [\,]) \wedge$
    $(\forall w.\ \text{temp}\ c\ w = \{\#\}_z) \wedge (\forall w.\ \text{glob}\ c\ w = \text{rec}\ c)$

In words: all global pointstamp counts in rec must be non-negative and equal to each worker's local view glob; all message queues and temporary changes must be empty.

Referencing our WCC example described in Section 2, the clocks protocol is the component in charge of distributing pointstamp changes to other workers. When one instance of the input operator a ceases sending data for a certain family of timestamps `(`t_1`,0)` it drops the corresponding pointstamp: the clocks protocol is in charge of *exchanging* this information with other workers, so that they can determine when all instances of a have ceased producing messages for a certain timestamp. This happens for all pointstamp changes in the system, including pointstamps that represent messages in-flight on channels.

The configurations evolve via one of three actions:

perf_op: A worker may perform an operation that causes a change in pointstamps. Changes may remove certain pointstamps and add others. They are recorded in rec and temp.
send_upd: A worker may broadcast some of its changes stored in temp to all other workers.
recv_upd: A worker may receive an earlier broadcast and update its local view glob.

Overall, the clocks protocol aims to establish that glob is a safe approximation for rec. Safe means here that no pointstamp in rec is less than any of glob's minimal pointstamps. To achieve this property, the protocol imposes a restriction on which new pointstamps may be introduced in rec and which progress updates may be broadcast. This restriction is the *uprightness* property that ensures that a pointstamp can only be introduced if simultaneously a smaller (supporting) pointstamp is removed. Formally, a signed multiset of pointstamps is upright if every positive entry is accompanied by a smaller negative entry:

definition supp :: $'p\ zmset \Rightarrow\ 'p \Rightarrow bool$ **where** supp $M\ p = (\exists p' < p.\ M\ \#_z\ p' < 0)$
definition upright :: $'p\ zmset \Rightarrow bool$ **where** upright $M = (\forall p.\ M\ \#_z\ p > 0 \longrightarrow \text{supp}\ M\ p)$

Abadi et al. [4] additionally require that the pointstamp p' in supp's definition satisfies $\forall p'' \leqslant p'.\ M\ \#_z\ p'' \leqslant 0$. The two variants of upright are equivalent in our formalization because signed multisets are finite and thus minimal elements exist even without $\leqslant$ being well-founded. The extra assumption on p' is occasionally useful in proofs.

definition perf_op :: $'w \Rightarrow 'p\ mset \Rightarrow 'p\ mset \Rightarrow ('w, 'p)\ conf \Rightarrow ('w, 'p)\ conf \Rightarrow bool$ **where**
  perf_op $w\ \Delta_{neg}\ \Delta_{pos}\ c\ c' =$ <u>let</u> $\Delta = \Delta_{pos} - \Delta_{neg}$ <u>in</u> $(\forall p.\ \Delta_{neg} \# p \leqslant \mathsf{rec}\ c \#_z p) \wedge \mathsf{upright}\ \Delta \wedge$
    $c' = c⦇\mathsf{rec} = \mathsf{rec}\ c + \Delta,\ \mathsf{temp} = (\mathsf{temp}\ c)(w := \mathsf{temp}\ c\ w + \Delta)⦈$

definition send_upd :: $'w \Rightarrow 'p\ set \Rightarrow ('w, 'p)\ conf \Rightarrow ('w, 'p)\ conf \Rightarrow bool$ **where**
  send_upd $w\ P\ c\ c' =$ <u>let</u> $\gamma = \{\#p \in\#_z \mathsf{temp}\ c\ w.\ p \in P\#\}$ <u>in</u>
    $\gamma \neq \{\#\}_z \wedge \mathsf{upright}\ (\mathsf{temp}\ c\ w - \gamma) \wedge$
    $c' = c⦇\mathsf{msg} = (\mathsf{msg}\ c)(w := \lambda w'.\ \mathsf{msg}\ c\ w\ w' \cdot [\gamma]),\ \mathsf{temp} = (\mathsf{temp}\ c)(w := \mathsf{temp}\ c\ w - \gamma)⦈$

definition recv_upd :: $'w \Rightarrow 'w \Rightarrow ('w, 'p)\ conf \Rightarrow ('w, 'p)\ conf \Rightarrow bool$ **where**
  recv_upd $w\ w'\ c\ c' = \mathsf{msg}\ c\ w\ w' \neq [] \wedge$
    $c' = c⦇\mathsf{msg} = (\mathsf{msg}\ c)(w := (\mathsf{msg}\ c\ w)(w' := \mathsf{tl}\ (\mathsf{msg}\ c\ w\ w'))),$
        $\mathsf{glob} = (\mathsf{glob}\ c)(w' := \mathsf{glob}\ c\ w' + \mathsf{hd}\ (\mathsf{msg}\ c\ w\ w'))⦈$

definition Next :: $('w, 'p)\ conf \Rightarrow ('w, 'p)\ conf \Rightarrow bool$ **where**
  Next $c\ c' = (c = c') \vee (\exists w\ \Delta_{neg}\ \Delta_{pos}.\ \mathsf{perf\_op}\ w\ \Delta_{neg}\ \Delta_{pos}\ c\ c') \vee$
    $(\exists w\ P.\ \mathsf{send\_upd}\ w\ P\ c\ c') \vee (\exists w\ w'.\ \mathsf{recv\_upd}\ w\ w'\ c\ c')$

Figure 3 Transition relation of Abadi et al.'s clocks protocol.

In practice, uprightness means that operators are only allowed to transition to pointstamps forward in time, and cannot re-introduce pointstamps that they relinquished. This is necessary to ensure that the frontiers always move to later timestamps and remain a conservative approximation of the pointstamps still present in the system. An advancing frontier triggers computation in some of the dataflow operators, for example to output the result of a time-based aggregation: this should only happen once all the relevant incoming data has been processed. This is the intuition behind the safety property of the protocol, Safe, discussed later in this section.

Figure 3 defines the three protocol actions formally as transition relations between an old configuration c and a new configuration c' along with the definition of the overall transition relation Next, which in addition to performing one of the actions may stutter, i.e., leave $c' = c$ unchanged. The three actions take further parameters as arguments, which we explain next.

The action perf_op is parameterized by a worker w and two (unsigned) multisets Δ_{neg} and Δ_{pos}, corresponding to negative and positive pointstamp changes. The action's overall effect on the pointstamps is thus $\Delta = \Delta_{pos} - \Delta_{neg}$. Here and elsewhere, subtraction expects signed multisets as arguments and we omit the type conversions from unsigned to signed multisets (which are included in our Isabelle formalization). The action is only enabled if its parameters satisfy two requirements. First, only pointstamps present in rec may be dropped, and thus the counts from Δ_{neg} must be bounded by the ones from rec. (Arguably, accessing rec is problematic for distributed workers. We rectify this modeling deficiency in Section 4.) Second, Δ must be upright, which ensures that we will never introduce a pointstamp that is lower than any pointstamp in rec. If these requirements are met, the action can be performed and will update both rec and temp with Δ (expressed using Isabelle's record and function update syntax).

The action send_upd is parameterized by a worker (sender w) and a set of pointstamps P, the outstanding changes to which, called γ, we want to broadcast. The key requirement is that the still unsent changes remain upright. Note that it is always possible to send all changes or all positive changes in temp, because any multiset without a positive change is upright. The operation enqueues γ in all message queues that have w as the sender. We model first-in-first-out queues as lists, where enqueuing means appending at the end ($\_ \cdot [\_]$).

Finally, the action recv_upd is parameterized by two workers (sender w and receiver w'). Given a non-empty queue msg $c\ w\ w'$, the action dequeues the first message (head hd gives the message, tail tl the queue's remainder) and adds it to the receiver's glob.

An execution of the clocks protocol is an infinite sequence of configurations. Infinite sequences of elements of type $'a$ are expressed in Isabelle using the coinductive datatype (short codatatype) of streams defined as **codatatype** $'a\ stream$ = Stream $'a$ ($'a\ stream$). We can inspect a stream's head and tail using the functions shd :: $'a\ stream \Rightarrow\ 'a$ and stl :: $'a\ stream \Rightarrow\ 'a\ stream$. Valid protocol executions satisfy the predicate Spec, i.e., they start in an initial configuration and all neighboring configurations are related by Next:

definition Spec :: $('w, 'p)\ conf\ stream \Rightarrow bool$ **where**
Spec s = now Init $s \wedge$ alw (relates Next) s

The operators now and relates lift unary and binary predicates over configurations to executions by evaluating them on the first one or two configurations respectively: now $P\ s$ = P (shd s) and relates $R\ s = R$ (shd s) (shd (stl s)). The coinductive operator alw resembles a temporal logic operator: alw $P\ s$ holds if P holds for all suffixes of s.

coinductive alw :: $('a\ stream \Rightarrow bool) \Rightarrow\ 'a\ stream \Rightarrow bool$ **where**
$P\ s \longrightarrow$ alw P (stl s) $\longrightarrow$ alw $P\ s$

We use the operators now, relates, and alw not only to specify valid execution, but also to state the main safety property. Moreover, we use the predicate vacant to express that a pointstamp (and all smaller pointstamps) are not present in a signed multiset:

definition vacant :: $'p\ zmset \Rightarrow\ 'p \Rightarrow bool$ **where** vacant $M\ p = (\forall p' \leqslant p.\ M\ \#_z\ p' = 0)$

Safety states that if any worker's glob becomes vacant up to some pointstamp, then that pointstamp and any lesser ones do not exist in the system, i.e., are not present in rec (and will remain so). Thus, safety allows workers to learn locally, via glob, something about the system's global state rec, namely that they will never encounter certain pointstamps again. Formally:

definition Safe :: $('w, 'p)\ conf\ stream \Rightarrow bool$ **where**
Safe $s = (\forall w\ p.$ now $(\lambda c.$ vacant (glob $c\ w$) $p)\ s \longrightarrow$ alw (now $(\lambda c.$ vacant (rec c) $p)\ s))$

lemma *safe:* Spec $s \longrightarrow$ alw Safe s

Our extended report [9] provides informal proof sketches for this and other safety properties.

Overall, we have replicated the formalization of Abadi et al.'s clocks protocol and the proof of its safety. Their protocol accurately models the implementation of the progress tracking protocol's distributed component in Timely Dataflow's original implementation Naiad with one subtle exception. The Naiad API (`OnNotify`, `SendBy`) allows an operator to repeatedly send data messages through its output port, which generates pointstamps at the receiver, without requiring that a pointstamp on the output port is decremented. This can result in a perf_op transition that is not upright. Additionally, the modern reference implementation of Timely Dataflow in Rust is more expressive than Naiad, and permits multiple operations that result in non-upright changes. We address and correct this limitation of the clocks protocol in Section 4.

One example of an operator that expresses behavior that results in non-upright changes is the input operator a in the WCC example. This operator may be reading data from an external source, and as soon as it receives new edges, it can forward them with the current pointstamp (*a.1*, (t_1,0)). This operator may be invoked multiple times, and perform this action repeatedly, until it determines from the external source that it should mark a certain timestamp as complete by dropping the pointstamp. All of these intermediate actions that send data at $(t_1, 0)$ are not upright, as sending messages creates new pointstamps on the message targets, without dropping a smaller pointstamp that can support the postive change.

4 Exchanging Progress

As outlined in the previous section, the clocks protocol is not flexible enough to capture executions with non-upright changes, which are desired and supported by concrete implementations of Timely Dataflow. At the same time, the protocol captures behaviors that are not reasonable in practice. Specifically, the clocks protocol does not separate the worker-local state from the system's global state. The perf_op transition, which is meant to be executed by a single worker, uses the global state to check whether the transition is enabled and simultaneously updates the global state rec as part of the transition. In particular, a single perf_op transition allows a worker to drop a pointstamp that in the real system "belongs" to a different worker w and simultaneously consistently updates w's state. In concrete implementations of Timely Dataflow, workers execute perf_op's asynchronously, and thus can only base the transition on information that is locally available to them.

Our modified model of the protocol, called *exchange*, resolves both issues. As the first step, we split the rec field into worker-local signed multisets caps of pointstamps, which we call *capabilities* as they indicate the possibility for the respective worker to emit these pointstamps. Workers may transfer capabilities to other workers. To do so, they asynchronously send capabilities as data messages to a central multiset data of pairs of workers (receivers) and pointstamps. We arrive at the following updated type of configurations:

record $('w :: \mathit{finite}, 'p :: \mathit{order})\ \mathit{conf} =$
  $\mathsf{caps} :: 'w \Rightarrow 'p\ \mathit{zmset}$
  $\mathsf{data} :: ('w \times 'p)\ \mathit{mset}$
  $\mathsf{msg} :: 'w \Rightarrow 'w \Rightarrow 'p\ \mathit{zmset}\ \mathit{list}$
  $\mathsf{temp} :: 'w \Rightarrow 'p\ \mathit{zmset}$
  $\mathsf{glob} :: 'w \Rightarrow 'p\ \mathit{zmset}$

Including this fine-grained view on pointstamps will allow workers to make transitions based on worker-local information. The entirety of the system's pointstamps, rec, which was previously part of the configuration and which the protocol aims to track, can be computed as the sum of all the workers' capabilities and data's in-flight pointstamps.

definition $\mathsf{rec} :: ('w, 'p)\ \mathit{conf} \Rightarrow 'p\ \mathit{zmset}$ **where** $\mathsf{rec}\ c = (\sum_w \mathsf{caps}\ c\ w) + \mathsf{snd}\ `\#\ \mathsf{data}\ c$

Here, the infix operator `# denotes the image of a function over a multiset with resulting counts given by $(f\ `\#\ M)\ \#\ x = \sum_{y \in \{y \in\# M \mid f\ y = x\}} M\ \#\ y$.

The exchange protocol's initial state allows workers to start with some positive capabilities. Each worker's glob must correctly reflect all initially present capabilities.

definition $\mathsf{Init} :: ('w, 'p)\ \mathit{conf} \Rightarrow \mathit{bool}$ **where**
  $\mathsf{Init}\ c = (\forall w\ p.\ \mathsf{caps}\ c\ w\ \#_z\ p \geqslant 0) \wedge \mathsf{data}\ c = \{\#\} \wedge$
    $(\forall w\ w'.\ \mathsf{msg}\ c\ w\ w' = [\,]) \wedge (\forall w.\ \mathsf{temp}\ c\ w = \{\#\}_z) \wedge (\forall w.\ \mathsf{glob}\ c\ w = \mathsf{rec}\ c)$

The transition relation of the exchange protocol, shown in Figure 4, is similar to that of the clocks protocol. We focus on the differences between the two protocols. First, the exchange protocol has an additional transition recv_cap to receive a previously sent capability. The transition removes a pointstamp from data and adds it to the receiving worker's capabilities.

definition $\mathsf{recv\_cap} :: {'w} \Rightarrow {'p} \Rightarrow ({'w}, {'p})\ \mathit{conf} \Rightarrow ({'w}, {'p})\ \mathit{conf} \Rightarrow \mathit{bool}$ **where**
$\quad \mathsf{recv\_cap}\ w\ p\ c\ c' = (w, p) \in\!\#\ \mathsf{data}\ c\ \wedge$
$\quad\quad c' = c(\!|\mathsf{caps} = (\mathsf{caps}\ c)(w := \mathsf{caps}\ c\ w + \{\#p\#\}_z),\ \mathsf{data} = \mathsf{data}\ c - \{\#(w, p)\#\}|\!)$

definition $\mathsf{perf\_op} :: {'w} \Rightarrow {'p}\ \mathit{mset} \Rightarrow ({'w} \times {'p})\ \mathit{mset} \Rightarrow {'p}\ \mathit{mset} \Rightarrow$
$\quad\quad ({'w}, {'p})\ \mathit{conf} \Rightarrow ({'w}, {'p})\ \mathit{conf} \Rightarrow \mathit{bool}$ **where**
$\quad \mathsf{perf\_op}\ w\ \Delta_{neg}\ \Delta_{data}\ \Delta_{self}\ c\ c' =$
$\quad\quad (\Delta_{data} \neq \{\#\} \vee \Delta_{self} - \Delta_{neg} \neq \{\#\}_z) \wedge (\forall p.\ \Delta_{neg}\ \#\ p \leqslant \mathsf{caps}\ c\ w\ \#_z\ p) \wedge$
$\quad\quad (\forall (w', p) \in\!\#\ \Delta_{data}.\ \exists p' < p.\ \mathsf{caps}\ c\ w\ \#_z\ p' > 0) \wedge$
$\quad\quad (\forall p \in\!\#\ \Delta_{self}.\ \exists p' \leqslant p.\ \mathsf{caps}\ c\ w\ \#_z\ p' > 0) \wedge$
$\quad\quad c' = c(\!|\mathsf{caps} = (\mathsf{caps}\ c)(w := \mathsf{caps}\ c\ w + \Delta_{self} - \Delta_{neg}),\ \mathsf{data} = \mathsf{data}\ c + \Delta_{data},$
$\quad\quad\quad\quad \mathsf{temp} = (\mathsf{temp}\ c)(w := \mathsf{temp}\ c\ w + (\mathsf{snd}\ `\#\ \Delta_{data} + \Delta_{self} - \Delta_{neg}))|\!)$

definition $\mathsf{send\_upd} :: {'w} \Rightarrow {'p}\ \mathit{set} \Rightarrow ({'w}, {'p})\ \mathit{conf} \Rightarrow ({'w}, {'p})\ \mathit{conf} \Rightarrow \mathit{bool}$ **where**
$\quad \mathsf{send\_upd}\ w\ P\ c\ c' = \underline{\mathsf{let}}\ \gamma = \{\#p \in\!\#_z\ \mathsf{temp}\ c\ w.\ p \in P\#\}\ \underline{\mathsf{in}}$
$\quad\quad \gamma \neq \{\#\}_z \wedge \mathsf{justified}\ (\mathsf{caps}\ c\ w)\ (\mathsf{temp}\ c\ w - \gamma) \wedge$
$\quad\quad c' = c(\!|\mathsf{msg} = (\mathsf{msg}\ c)(w := \lambda w'.\ \mathsf{msg}\ c\ w\ w' \cdot [\gamma]),\ \mathsf{temp} = (\mathsf{temp}\ c)(w := \mathsf{temp}\ c\ w - \gamma)|\!)$

definition $\mathsf{recv\_upd} :: {'w} \Rightarrow {'w} \Rightarrow ({'w}, {'p})\ \mathit{conf} \Rightarrow ({'w}, {'p})\ \mathit{conf} \Rightarrow \mathit{bool}$ **where**
$\quad \mathsf{recv\_upd}\ w\ w'\ c\ c' = \mathsf{msg}\ c\ w\ w' \neq [\,] \wedge$
$\quad\quad c' = c(\!|\mathsf{msg} = (\mathsf{msg}\ c)(w := (\mathsf{msg}\ c\ w)(w' := \mathsf{tl}\ (\mathsf{msg}\ c\ w\ w'))),$
$\quad\quad\quad\quad \mathsf{glob} = (\mathsf{glob}\ c)(w' := \mathsf{glob}\ c\ w' + \mathsf{hd}\ (\mathsf{msg}\ c\ w\ w'))|\!)$

definition $\mathsf{Next} :: ({'w}, {'p})\ \mathit{conf} \Rightarrow ({'w}, {'p})\ \mathit{conf} \Rightarrow \mathit{bool}$ **where**
$\quad \mathsf{Next}\ c\ c' = (c = c') \vee (\exists w\ p.\ \mathsf{recv\_cap}\ w\ p\ c\ c') \vee$
$\quad\quad (\exists w\ \Delta_{neg}\ \Delta_{data}\ \Delta_{self}.\ \mathsf{perf\_op}\ w\ \Delta_{neg}\ \Delta_{data}\ \Delta_{self}\ c\ c') \vee$
$\quad\quad (\exists w\ P.\ \mathsf{send\_upd}\ w\ P\ c\ c') \vee (\exists w\ w'.\ \mathsf{recv\_upd}\ w\ w'\ c\ c')$

Figure 4 Transition relation of the exchange protocol.

The perf_op transition resembles its homonymous counterpart from the clocks protocol. Yet, the information flow is more fine grained. In particular, the transition is parameterized by a worker w and three multisets of pointstamps. As in the clocks protocol, the multiset Δ_{neg} represents negative changes to pointstamps. Only pointstamps for which w owns a capability in caps may be dropped in this way. The other two multisets Δ_{data} and Δ_{self} represent positive changes. The multiset Δ_{data} represents positive changes to other workers' capabilities – the receiving worker is stored in Δ_{data}. These changes are not immediately applied to the other worker's caps, but are sent via the data field. The multiset Δ_{self} represents positive changes to w's capabilities, which are applied immediately applied to w's caps. The separation between Δ_{data} and Δ_{self} is motivated by different requirements on these positive changes to pointstamps that we prove to be sufficient for safety. To send a positive capability to another worker, w is required to hold a positive capability for a strictly smaller pointstamp. In contrast, w can create a new capability for itself, if it is already holding a capability for the very same (or a smaller) pointstamp. In other words, w can arbitrarily increase the multiset counts of its own capabilities. Note that, unlike in the clocks protocol, there is no requirement of uprightness and, in fact, workers are not required to perform negative changes at all. Of course, it is useful for workers to perform negative changes every now and then so that the overall system can make progress.

The first condition in perf_op, namely $\Delta_{data} \neq \{\#\} \vee \Delta_{self} - \Delta_{neg} \neq \{\#\}_z$, ensures that the transition changes the configuration. In the exchange protocol, we also include explicit stutter steps in the Next relation ($c = c'$) but avoid them in the individual transitions.

locale graph =
  fixes weights :: $('vtx :: finite) \Rightarrow 'vtx \Rightarrow ('lbl :: \{order, monoid\_add\})\ antichain$
  assumes $(l :: 'lbl) \geqslant 0$ **and** $(l_1 :: 'lbl) \leqslant l_3 \longrightarrow l_2 \leqslant l_4 \longrightarrow l_1 + l_2 \leqslant l_3 + l_4$
  and weights $l\ l = \{\}$
locale dataflow = graph summary
  for summary :: $('l :: finite) \Rightarrow 'l \Rightarrow ('sum :: \{order, monoid\_add\})\ antichain$ +
  fixes $\bigoplus :: ('t :: order) \Rightarrow 'sum \Rightarrow 't$
  assumes $t \bigoplus 0 = t$ **and** $(t \bigoplus s) \bigoplus s' = t \bigoplus (s + s')$ **and** $t \leqslant t' \longrightarrow s \leqslant s' \longrightarrow t \bigoplus s \leqslant t' \bigoplus s'$
  and path $l\ l\ xs \longrightarrow xs \neq [\,] \longrightarrow t < t \bigoplus (\sum xs)$

Figure 5 Locales for graphs and dataflows.

Sending (send_upd) and receiving (recv_upd) progress updates works precisely as in the clocks protocol except for the condition on what remains in the sender's temp highlighted in gray in Figure 4. Because we allowed perf_op to perform non-upright changes, we can no longer expect the contents of temp to be upright. Instead, we use the predicate justified, which offers three possible justifications for positive entries in the signed multiset M (in contrast to upright's sole justification of being supported in M):

definition justified :: $'p\ zmset \Rightarrow 'p\ zmset \Rightarrow bool$ **where**
  justified $C\ M = (\forall p.\ M \#_z p > 0 \longrightarrow \text{supp}\ M\ p \vee (\exists p' < p.\ C \#_z p' > 0) \vee M \#_z p < C \#_z p)$

Thus, a positive count for pointstamp p in M may be either

- supported in M, i.e., in particular every upright change is justified, or
- justified by a smaller pointstamp in C, which we think of as the sender's capabilities, or
- justified by p in C, with the requirement that p's count in M is smaller than p's count in C.

The definitions of valid executions Spec and the safety predicate Safe are unchanged compared to the clocks protocol. Also, we prove precisely the same safety property *safe* following a similar proof structure.

We also derive the following additional property of glob, which shows that any in-flight progress updates to a pointstamp p, positive or negative, have a corresponding positive count for some pointstamp less or equal than p in the receiver's glob.

lemma *glob:* Spec $s \longrightarrow$ alw (now ($\lambda c.\ \forall w\ w'\ p.$
  $(\exists M \in \text{set}\ (\text{msg}\ c\ w\ w').\ p \in\#_z M) \longrightarrow (\exists p' \leqslant p.\ \text{glob}\ c\ w'\ \#_z\ p' > 0)))\ s$

5 Locally Propagating Progress

The previous sections focused on the progress-relevant communication between workers and abstracted over the actual dataflow that is evaluated by each worker. Next, we refine this abstraction: we model the actual dataflow graph as a weighted directed graph with vertices representing operator input and output ports, termed *locations*. We do not distinguish between source and target locations and thus also not between internal and dataflow edges. Each weight denotes a minimum increment that is performed to a timestamp when it conceptually travels along the corresponding edge from one location to another. On a single worker, progress updates can be communicated locally, so that every operator learns which timestamps it may still receive in the future. We formalize Timely Dataflow's approach for this local communication: the algorithm gradually propagates learned pointstamp changes along dataflow edges to update downstream frontiers.

Figure 5 details our modeling of graphs and dataflows, which uses locales [5] to capture our abstract assumptions on dataflows and timestamps. A locale lets us fix parameters (types and constants) and assume properties about them. In our setting, a weighted directed graph is given by a finite (class *finite*) type *'vtx* of vertices and a weights function that assigns each pair of vertices a weight. To express weights, we fix a type of labels *'lbl*, which we assume to be partially ordered (class *order*) and to form a monoid (class *monoid_add*) with the monoid operation + and the neutral element 0. We assume that labels are non-negative and that + on labels is monotone with respect to the partial order $\leqslant$. A weight is then an antichain of labels, that is a set of incomparable (with respect to $\leqslant$) labels, which we model as follows:

typedef (*'t* :: *order*) *antichain* = $\{A$ :: *'t set*. finite $A \wedge (\forall a \in A.\ \forall b \in A.\ a \nless b \wedge b \nless a)\}$

We use standard set notation for antichains and omit type conversions from antichains to (signed) multisets. The empty antichain {} is a valid weight, too, in which case we think of the involved vertices as not being connected to each other. Thus, the graph locale's final assumption expresses the non-existence of self-edges in a graph.

Within the graph locale, we can define the predicate path :: *'vtx* ⇒ *'vtx* ⇒ *'lbl list* ⇒ *bool*. Intuitively, path v w xs expresses that the list of labels xs is a valid path from v to w (the empty list being a valid path only if $v = w$ and any weight $l \in$ weights u v can extend a valid path from v to w to a path from u to w). We omit path's formal straightforward inductive definition. Note that even though self-edges are disallowed, cycles in graphs are possible (and desired). In other words, path v v xs can be true for a non-empty list xs.

The second locale, dataflow, has two purposes. First, it refines the generic graph terminology from vertices and labels to locations (*'l*) and summaries (*'sum*), which is the corresponding terminology used in Timely Dataflow. Second, it introduces the type for timestamps *'t*, which is partially ordered (class *order*) and an operation $\oplus$ (read as "results in") that applies a summary to a timestamp to obtain a new timestamp. We chose the asymmetric symbol for the operation to remind the reader that its two arguments have different types, timestamps and summaries. The locale requires the operation $\oplus$ to be well-behaved with respect to the available vocabulary on summaries (0, +, and $\leqslant$). Moreover, it requires that proper cycles xs have a path summary $\sum xs$ (defined by iterating +) that strictly increments any timestamp t.

Now, consider a function P :: *'l* ⇒ *'t zmset* that assigns each location a set of timestamps that it currently holds. We are interested in computing a lower bound of timestamps (with respect to the order $\leqslant$) that may arrive at any location for a given P. Timely Dataflow calls antichains that constitute such a lower bound frontiers. Formally, a frontier is the set of minimal incomparable elements that have a positive count in a signed multiset of timestamps.

definition antichain_of :: *'t set* ⇒ *'t set* **where** antichain_of $A = \{x \in A.\ \neg\exists y \in A.\ y < x\}$
lift_definition frontier :: *'t zmset* ⇒ *'t antichain* **is** $\lambda M.$ antichain_of $\{t.\ M \#_z t > 0\}$

Our frontier of interest, called the implied frontier, at location l can be computed directly for a given function P by adding, for every location l', every (minimal) possible path summary between l' and l, denoted by the antichain path_summary l' l, to every timestamp present at l' and computing the frontier of the result. Formally, we first lift $\oplus$ to signed multisets and antichains. Then, we use the lifted operator $\bigoplus$ to define the implied frontier.

definition $\mathsf{change\_multiplicity} :: {'l} \Rightarrow {'t} \Rightarrow \mathit{int} \Rightarrow ({'l}, {'t})\ \mathit{conf} \Rightarrow ({'l}, {'t})\ \mathit{conf} \Rightarrow \mathit{bool}$ **where**
$\quad \mathsf{change\_multiplicity}\ l\ t\ n\ c\ c' = n \neq 0 \wedge (\exists t' \in \mathsf{frontier}\ (\mathsf{implications}\ c\ l).\ t' \leqslant t) \wedge$
$\quad\quad c' = c(\!|\mathsf{pts} = (\mathsf{pts}\ c)(l := \mathsf{pts}\ c\ l + \mathsf{replicate}\ n\ t),$
$\quad\quad\quad\quad \mathsf{work} = (\mathsf{work}\ c)(l := \mathsf{work}\ c\ l + \mathsf{frontier}\ (\mathsf{pts}\ c'\ l) - \mathsf{frontier}\ (\mathsf{pts}\ c\ l))|\!)$

definition $\mathsf{propagate} :: {'l} \Rightarrow {'t} \Rightarrow ({'l}, {'t})\ \mathit{conf} \Rightarrow ({'l}, {'t})\ \mathit{conf} \Rightarrow \mathit{bool}$ **where**
$\quad \mathsf{propagate}\ l\ t\ c\ c' = t \in\#_z \mathsf{work}\ c\ l \wedge (\forall l'.\ \forall t' \in\#_z \mathsf{work}\ c\ l'.\ \neg t' < t) \wedge$
$\quad\quad c' = c(\!|\mathsf{imp} = (\mathsf{imp}\ c)(l := \mathsf{imp}\ c\ l + \mathsf{replicate}\ (\mathsf{work}\ c\ l\ \#_z\ t)\ t,$
$\quad\quad\quad\quad \mathsf{work} = \lambda l'.\ \underline{\mathsf{if}}\ l = l'\ \underline{\mathsf{then}}\ \{\#t' \in\#_z \mathsf{work}\ c\ l.\ t' \neq t\#\}$
$\quad\quad\quad\quad\quad \underline{\mathsf{else}}\ \mathsf{work}\ c\ l' + ((\mathsf{frontier}\ (\mathsf{imp}\ c'\ l) - \mathsf{frontier}\ (\mathsf{imp}\ c\ l)) \bigoplus \mathsf{summary}\ l\ l')|\!)$

definition $\mathsf{Next} :: ({'l}, {'t})\ \mathit{conf} \Rightarrow ({'l}, {'t})\ \mathit{conf} \Rightarrow \mathit{bool}$ **where**
$\quad \mathsf{Next}\ c\ c' = (c = c') \vee (\exists l\ t\ n.\ \mathsf{change\_multiplicity}\ l\ t\ n\ c\ c') \vee (\exists l\ t.\ \mathsf{propagate}\ l\ t\ c\ c')$

Figure 6 Transition relation of the local progress propagation.

definition $\bigoplus :: {'t}\ \mathit{zmset} \Rightarrow {'sum}\ \mathit{antichain} \Rightarrow {'t}\ \mathit{zmset}$ **where**
$\quad M \bigoplus A = \sum_{s \in A} (\lambda t.\ t \oplus s)\ `\#_z\ M$
definition $\mathsf{implied\_frontier} :: ({'l} \Rightarrow {'t}\ \mathit{zmset}) \Rightarrow {'l} \Rightarrow {'t}\ \mathit{antichain}$ **where**
$\quad \mathsf{implied\_frontier}\ P\ l = \mathsf{frontier}\ (\sum_{l'} (\mathsf{pos}_z\ (P\ l') \bigoplus \mathsf{path\_summary}\ l'\ l))$

Above and elsewhere, given a signed multiset M, we write $f\ `\#_z\ M$ for the image (as a signed multiset) of f over M and $\mathsf{pos}_z\ M$ for the signed multiset of M's positive entries.

Computing the implied frontier for each location in this way (quadratic in the number of locations) would be too inefficient, especially because we want to frequently supply operators with up-to-date progress information. Instead, we follow the optimized approach implemented in Timely Dataflow: after performing some work and making some progress, operators start pushing relevant updates only to their immediate successors in the dataflow graph. The information gradually propagates and eventually converges to the implied frontier. Despite this local propagation not being a distributed protocol as such, we formalize it for a fixed dataflow in a similar state-machine style as the earlier exchange protocol.

Local propagation uses a configuration consisting of three signed multiset components.

record $({'l} :: \mathit{finite}, {'t} :: \{\mathit{monoid\_add}, \mathit{order}\})\ \mathit{conf} =$
$\quad \mathsf{pts} :: {'l} \Rightarrow {'t}\ \mathit{zmset}$
$\quad \mathsf{imp} :: {'l} \Rightarrow {'t}\ \mathit{zmset}$
$\quad \mathsf{work} :: {'l} \Rightarrow {'t}\ \mathit{zmset}$

Following Timely Dataflow terminology, pointstamps pts are the present timestamps grouped by location (the P function from above). The implications imp are the output of the local propagation and contain an over-approximation of the implied frontier (as we will show). Finally, the worklist work is an auxiliary data structure to store not-yet propagated timestamps.

Initially, all implications are empty and worklists consist of the frontiers of the pointstamps.

definition $\mathsf{Init} :: ({'l}, {'t})\ \mathit{conf} \Rightarrow \mathit{bool}$ **where**
$\quad \mathsf{Init}\ c = (\forall l.\ \mathsf{imp}\ c\ l = \{\#\}_z \wedge \mathsf{work}\ c\ l = \mathsf{frontier}\ (\mathsf{pts}\ c\ l))$

The propagation proceeds by executing one of two actions shown in Figure 6. The action change_multiplicity constitutes the algorithm's information input: The system may have changed the multiplicity of some timestamp t at location l and can use this action to notify the propagation algorithm of the change. The change value n is required to be non-zero and the affected timestamp t must be witnessed by some timestamp in the implications. Note that the latter requirement prohibits executing this action in the initial state. The action updates the pointstamps according to the declared change. It also updates the worklist, but only if the update of the pointstamps affects the frontier of the pointstamps at l and moreover the worklists are updated merely by the change to the frontier.

The second action, propagate, applies the information for the timestamp t stored in the worklist at a given location l, to the location's implications (thus potentially enabling the first action). It also updates the worklists at the location's immediate successors in the dataflow graph. Again the worklist updates are filtered by whether they affect the frontier (of the implications) and are adjusted by the summary between l and each successor. Importantly, only minimal timestamps (with respect to timestamps in worklists at all locations) may be propagated, which ensures that any timestamp will eventually disappear from all worklists.

The overall transition relation Next allows us to choose between these two actions and a stutter step. Together with Init, it gives rise to the predicate describing valid executions in the standard way: Spec s = now Init $s \wedge$ alw (relates Next) s.

We show that valid executions satisfy a safety invariant. Ideally, we would like to show that for any t with a positive count in pts at location l and for any path summary s between l and some location l', there is a timestamp in the (frontier of the) implications at l' that is less than or equal to $t \oplus s$. In other words, the location l' is aware that it may still encounter timestamp $t \oplus s$. Stated as above, the invariant does not hold, due to the not-yet-propagated progress information stored in the worklists. If some timestamp, however, does not occur in any worklist (formalized by the below work_vacant predicate), we obtain our desired invariant Safe.

definition work_vacant :: $('l, 't)\ conf \Rightarrow {'t} \Rightarrow bool$ **where**
work_vacant $c\ t = (\forall l\ l'\ s\ t'.\ t' \in\#_z$ work $c\ l \longrightarrow s \in$ path_summary $l\ l' \longrightarrow t' \oplus s \not\leq t)$
definition Safe :: $('l, 't)\ conf\ stream \Rightarrow bool$ **where**
Safe $c = (\forall l\ l'\ t\ s.$ pts $c\ l\ \#_z\ t > 0 \wedge s \in$ path_summary $l\ l' \wedge$ work_vacant $c\ (t \oplus s) \longrightarrow$
$(\exists t' \in$ frontier (imp $c\ l'$). $t' \leq t \oplus s))$
lemma *safe:* Spec $s \longrightarrow$ alw (now Safe) s

In our running WCC example, Safe is for example necessary to determine once operator b has received all incoming updates for a certain round of label propagation, which is encoded as a timestamp (t_1, t_2). If a pointstamp at port *b.3* was not correctly reflected in the frontier at *b.1* the operator may incorrectly determine that it has seen all incoming labels for a certain graph node and proceed to the next round of propagation. Safe states, that this cannot happen and all pointstamps are correctly reflected in relevant downstream frontiers.

The safety proof relies on two auxiliary invariants. First, implications have only positive entries. Second, the sum of the implication and the worklist at a given location l is equal to the sum of the frontier of the pointstamps at l and the sum of all frontiers of the implications of all immediate predecessor locations l' (adjusted by the corresponding summary summary $l'\ l$).

While the above safety property is sufficient to prove safety of the combination of the local propagation and the exchange protocol in the next section, we also establish that the computed frontier of the implications converges to the implied frontier. Specifically, the two frontiers coincide for timestamps which are not contained in any of the worklists.

lemma *implied_frontier:* Spec $s \longrightarrow$ alw (now ($\lambda c.$ work_vacant $c\ t \longrightarrow$
$(\forall l.\ t \in$ frontier (imp $c\ l) \longleftrightarrow t \in$ implied_frontier (pts c) l))) s

6 Progress Tracking

We are now ready to combine the two parts presented so far: the between-worker exchange of progress updates (Section 4) and the worker-local progress propagation (Section 5). The combined protocol takes pointstamp changes and determines per-location frontiers at each operator on each worker. It operates on configurations consisting of a single exchange protocol configuration (referred to with the prefix E) and for each worker a local propagation configuration (prefix P) and a Boolean flag indicating whether the worker has been properly initialized.

record $('w :: \mathit{finite}, 'l :: \mathit{finite}, 't :: \{\mathit{monoid\_add}, \mathit{order}\})\ \mathit{conf} =$
  exch :: $('w, 'l \times 't)\ E.\mathit{conf}$
  prop :: $'w \Rightarrow ('l, 't)\ P.\mathit{conf}$
  init :: $'w \Rightarrow \mathit{bool}$

As pointstamps in the exchange protocol, we use pairs of locations and timestamps. To order pointstamps, we use the following *could-result-in* relation, inspired by Abadi and Isard [3].

definition $\leqslant_{cri}$ **where** $(l,t) \leqslant_{cri} (l',t') = (\exists s \in \mathsf{path\_summary}\ l\ l'.\ t \oplus s \leqslant t')$

As required by the exchange protocol, this definition yields a partial order. In particular, antisymmetry follows from the assumption that proper cycles have a non-zero summary and transitivity relies on the operation $\oplus$ being monotone. Intuitively, $\leqslant_{cri}$ captures a notion of reachability in the dataflow graph: as timestamp t traverses the graph starting at location l, it could arrive at location l', being incremented to timestamp t'. (In Timely Dataflow, an edge's summary represents the minimal increment to a timestamp when it traverses that edge.)

In an initial combined configuration, all workers are not initialized and all involved configurations are initial. Moreover, the local propagation's pointstamps coincide with exchange protocol's glob, which is kept invariant in the combined protocol.

definition Init :: $('w, 'l, 't)\ \mathit{conf} \Rightarrow \mathit{bool}$ **where**
  $\mathsf{Init}\ c = (\forall w.\ \mathsf{init}\ c\ w = \mathsf{False}) \wedge \mathsf{E.Init}\ (\mathsf{exch}\ c) \wedge (\forall w.\ \mathsf{P.Init}\ (\mathsf{prop}\ c\ w)) \wedge$
    $(\forall w\ l\ t.\ \mathsf{P.pts}\ (\mathsf{prop}\ c\ w)\ l\ \#_z\ t = \mathsf{E.glob}\ (\mathsf{exch}\ c)\ w\ \#_z\ (l,t))$

Figure 7 shows the combined protocol's transition relation Next. Most actions have identical names as the exchange protocol's actions and they mostly perform the corresponding actions on the exchange part of the configuration. In addition, the recv_upd action also performs several change_multiplicity local propagation actions: the receiver updates the state of its local propagation configuration for all received timestamp updates. The action propagate does not have a counterpart in the exchange protocol. It iterates, using the while_option combinator from Isabelle's library, propagation on a single worker until all worklists are empty. The term while_option $b\ c\ s$ repeatedly applies c starting from the initial state s, until the predicate b is satisfied. Overall, it evaluates to Some s' satisfying $\neg b\ s'$ and $s' = c\ (\cdots (c\ s))$ with the least possible number of repetitions of c and to None if no such state exists. Thus, it is only possible to take the propagate action, if the repeated propagation terminates for the considered configuration. We believe that repeated propagation terminates for any configuration, but we do not prove this non-obvious[1] fact formally. Timely Dataflow

[1] Because propagation must operate on a globally minimal timestamp and because loops in the dataflow graph have a non-zero summary, repeated propagation will eventually forever remove any timestamp from any worklist. However, it is not as obvious why it eventually will stop introducing larger and larger timestamps in worklists. The termination argument must rely on the fact that only timestamps that modify the frontier of the implications are ever added to worklists.

definition recv_cap :: $'w \Rightarrow\ 'l \times\ 't \Rightarrow (\ 'w,\ 'l,\ 't)\ conf \Rightarrow (\ 'w,\ 'l,\ 't)\ conf \Rightarrow bool$ **where**
  recv_cap $w\ p\ c\ c'$ = E.recv_cap $w\ p$ (exch c) (exch c') $\wedge$ prop c' = prop c $\wedge$ init c' = init c

definition perf_op :: $'w \Rightarrow (\ 'l \times\ 't)\ mset \Rightarrow (\ 'w \times (\ 'l \times\ 't))\ mset \Rightarrow (\ 'l \times\ 't)\ mset \Rightarrow$
  $(\ 'w,\ 'l,\ 't)\ conf \Rightarrow (\ 'w,\ 'l,\ 't)\ conf \Rightarrow bool$ **where**
  perf_op $w\ \Delta_{neg}\ \Delta_{data}\ \Delta_{self}\ c\ c'$ = E.perf_op $w\ \Delta_{neg}\ \Delta_{data}\ \Delta_{self}$ (exch c) (exch c') $\wedge$
    prop c' = prop c $\wedge$ init c' = init c

definition send_upd :: $'w \Rightarrow (\ 'l \times\ 't)\ set \Rightarrow (\ 'w,\ 'l,\ 't)\ conf \Rightarrow (\ 'w,\ 'l,\ 't)\ conf \Rightarrow bool$ **where**
  send_upd $w\ P\ c\ c'$ = E.send_upd (exch c) (exch c') $w\ P$ $\wedge$ prop c' = prop c $\wedge$ init c' = init c

definition cm_all :: $(\ 'l,\ 't)\ P.conf \Rightarrow (\ 'l \times\ 't)\ zmset \Rightarrow (\ 'l,\ 't)\ P.conf$ **where**
  cm_all $c\ \Delta$ = Set.fold ($\lambda(l, t)\ c$. SOME c'. P.change_multiplicity $c\ c'\ l\ t\ (\Delta\ \#_z\ (l, t))$) c
    $\{(l, t).\ (l, t) \in\#_z \Delta\}$

definition recv_upd :: $'w \Rightarrow\ 'w \Rightarrow (\ 'w,\ 'l,\ 't)\ conf \Rightarrow (\ 'w,\ 'l,\ 't)\ conf \Rightarrow bool$ **where**
  recv_upd $w\ w'\ c\ c'$ = init $c\ w'$ $\wedge$ E.recv_upd $w\ t$ (exch c) (exch c') $\wedge$
    prop c' = (prop c)(w' := cm_all (prop $c\ w'$) (hd (E.msg (exch c)))) $\wedge$ init c' = init c

definition propagate :: $'w \Rightarrow (\ 'w,\ 'l,\ 't)\ conf \Rightarrow (\ 'w,\ 'l,\ 't)\ conf \Rightarrow bool$ **where**
  propagate $w\ c\ c'$ = exch c' = exch c $\wedge$ init c' = (init c)(w := True) $\wedge$
    (Some $\circ$ prop c') = (Some $\circ$ prop c)(w := while_option
      ($\lambda c.\ \exists l$. P.work $c\ l \neq \{\#\}_z$) (λc. SOME c'. $\exists l\ t$. P.propagate $l\ t\ c\ c'$) (prop $c\ w$))

definition Next :: $(\ 'w,\ 'l,\ 't)\ conf \Rightarrow (\ 'w,\ 'l,\ 't)\ conf \Rightarrow bool$ **where**
  Next $c\ c'$ = $(c = c')$ $\vee$ ($\exists w\ p$. recv_cap $w\ p\ c\ c'$) $\vee$
    ($\exists w\ \Delta_{neg}\ \Delta_{data}\ \Delta_{self}$. perf_op $w\ \Delta_{neg}\ \Delta_{data}\ \Delta_{self}\ c\ c'$) $\vee$
    ($\exists w\ P$. send_upd $w\ P\ c\ c'$) $\vee$ ($\exists w\ w'$. recv_upd $w\ w'\ c\ c'$) $\vee$ ($\exists w$. propagate $w\ c\ c'$)

Figure 7 Transition relation of the combined progress tracker.

also iterates propagation until all worklists of a worker become empty. This gives us additional empirical evidence that the iteration terminates on practical dataflows. Moreover, even if the iteration were to not terminate for some worker on some dataflow (both in Timely Dataflow and in our model), our combined protocol can faithfully capture this behavior by not executing the propagate action, but also not any other action involving the looping worker, thus retaining safety for the rest of the workers. Finally, any worker that has completed at least one propagation action is considered to be initialized (by setting its init flag to True).

The Init predicate and the Next relation give rise to the familiar specification of valid executions Spec s = now Init s $\wedge$ alw (relates Next) s. Safety of the combined protocol can be described informally as follows: Every initialized worker w has some evidence for the existence of a timestamp t at location l at *any* worker w' in the frontier of its (i.e., w's) implications at all locations l' reachable from l. Formally, E.rec contains the timestamps that exist in the system:

definition Safe :: $(\ 'w,\ 'l,\ 't)\ conf\ stream \Rightarrow bool$ **where**
  Safe c = ($\forall w\ l\ l'\ t\ s$. init $c\ w$ $\wedge$ E.rec (exch c) $\#_z\ (l, t) > 0 \wedge s \in$ path_summary $l\ l'$ $\longrightarrow$
    ($\exists t' \in$ frontier (P.imp (prop $c\ w$) l'). $t' \leqslant t \oplus s$)

Our main formalized result is the statement that the above predicate is an invariant.

lemma *safe:* Spec s $\longrightarrow$ alw (now Safe) s

In the combined progress tracking protocol, safety guarantees that if a pointstamp is present at an operator's port, it is correctly reflected at every downstream port. In the WCC example, when deployed on two workers, each operator is instantiated twice, once on each worker. If a pointstamp (*b.3*, `(3,0)`) is present on port *b.3* of one of the instances of operator *b*, the frontier at *c.1* on all workers must contain a t such that $t \preceq (3, 0)$. Due to the summary between *c.1* and *c.2*, frontiers at *c.2* and *b.1* must contain a t such that $t \preceq (3, 1)$. As an example, this ensures that operator *b* waits for each of its instances to complete the first round propagation of all labels before it chooses the lowest label for the next round.

7 Discussion

We have presented an Isabelle/HOL formalization of Timely Dataflow's progress tracking protocol, including the verification of its safety. Compared to an earlier formalization by Abadi et al. [4], our protocol is both more general, which allows it to capture behaviors present in the implementations of Timely Dataflow and absent in Abadi et al.'s model, and more detailed in that it explicitly models the local propagation of progress information.

Our formalization spans about 7 000 lines of Isabelle definitions and proofs. These are roughly distributed as follows over the components we presented: basic properties of graphs and signed multisets (1 000), exchange protocol (3 100), local propagation (1 700), combined protocol (1 200). This is comparable in size to the TLA$^+$ Proof System formalization by Abadi et al., even though we formalized a significantly more detailed, complex, and realistic variant of the progress tracking protocol. Ground to this claim is the fact that we had actually started our formalization by porting significant parts of the TLA$^+$ Proof System formalization to Isabelle. We completed the proofs of their two main safety statement within one person-week in about 1 000 lines of Isabelle (not included above). Our use of Isabelle's library for linear temporal logic on streams (in particular, the coinductive predicate alw) allowed us to copy directly a vast majority of the TLA$^+$ definitions. Additionally, Isabelle's mature proof automation allowed us to apply a fairly mechanical porting process to many of the proofs. Most ported lemmas could be proved either directly by Sledgehammer [29] or by sketching an Isar [32] proof skeleton of the main proof steps and discharging most of the resulting subgoals with Sledgehammer.

In the subsequent development of the combined protocol, Isabelle's locales [5] were an important asset. By confining the exchange protocol and the local propagation each to their own local assumptions, we were able to develop them in parallel and in their full generality. Thus, we obtain formal models not only of the combined protocol itself but also of these two subsystems in a generality that goes beyond what is needed for the concrete combined instance. For example, although the combined protocol uses the could-result-in order, the exchange protocol works for any partial order on pointstamps. Moreover, the combined protocol always propagates until all worklists are empty, even though the local propagation's safety supports small-step propagation, resulting in a more fine-grained safety property via work_vacant.

In our formalization, we make extensive use of signed multisets [8]. The alternative (used in the TLA$^+$ Proof System formalization), would be to use integer-valued functions instead. The signed multiset type additionally captures a finite domain assumption, which it was convenient not to carry around explicitly and in particular simplified reasoning about summations. The expected downside of having separate types for function-like (*mset*) and set-like (*antichain*) objects was the need to insert explicit type conversions and to transfer properties across these conversions. Both complications were to some extent alleviated by Lifting and Transfer [19].

Progress tracking is only a small, albeit arguably the most intricate part of Timely Dataflow. Verifying its safety is an important first step towards our long-term goal of developing a verified, executable variant of Timely Dataflow and using it as a framework for the verification of efficient and scalable stream processing algorithms. More modest next steps are to prove the local propagation algorithm's termination and to make our formalization executable. We have made first steps towards the latter goal, by creating a functional, executable variant of the local propagation's transition relation [12]. This allowed us to compare our formalized propagation algorithm to the one implemented in Rust. We found that their input–output behavior coincides on all example dataflows accompanying the Rust implementation, confirming our model's faithfulness. We are working on including the exchange protocol in this comparative testing, which poses a challenge because of the protocol's distributed nature.

References

1 Github: Timely dataflow. URL: `https://github.com/TimelyDataflow/timely-dataflow/`.

2 Materialize: Incrementally-updated materialized views. URL: `https://materialize.com`.

3 Martín Abadi and Michael Isard. Timely dataflow: A model. In Susanne Graf and Mahesh Viswanathan, editors, *FORTE 2015*, volume 9039 of *LNCS*, pages 131–145. Springer, 2015. `doi:10.1007/978-3-319-19195-9_9`.

4 Martín Abadi, Frank McSherry, Derek Gordon Murray, and Thomas L. Rodeheffer. Formal analysis of a distributed algorithm for tracking progress. In Dirk Beyer and Michele Boreale, editors, *FMOODS/FORTE 2013*, volume 7892 of *LNCS*, pages 5–19. Springer, 2013. `doi:10.1007/978-3-642-38592-6_2`.

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

6 Véronique Benzaken and Evelyne Contejean. A Coq mechanised formal semantics for realistic SQL queries: formally reconciling SQL and bag relational algebra. In Assia Mahboubi and Magnus O. Myreen, editors, *CPP 2019*, pages 249–261. ACM, 2019. `doi:10.1145/3293880.3294107`.

7 Véronique Benzaken, Evelyne Contejean, Chantal Keller, and E. Martins. A Coq formalisation of SQL's execution engines. In Jeremy Avigad and Assia Mahboubi, editors, *ITP 2018*, volume 10895 of *LNCS*, pages 88–107. Springer, 2018. `doi:10.1007/978-3-319-94821-8_6`.

8 Jasmin Christian Blanchette, Mathias Fleury, and Dmitriy Traytel. Nested multisets, hereditary multisets, and syntactic ordinals in Isabelle/HOL. In Dale Miller, editor, *FSCD 2017*, volume 84 of *LIPIcs*, pages 11:1–11:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPIcs.FSCD.2017.11`.

9 Matthias Brun, Sára Decova, Andrea Lattuada, and Dmitriy Traytel. Verified progress tracking for timely dataflow (extended report), 2021. URL: `https://www.github.com/matthias-brun/progress-tracking-formalization/raw/main/rep.pdf`.

10 Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. Verifying safety properties with the TLA+ proof system. In Jürgen Giesl and Reiner Hähnle, editors, *IJCAR 2010*, volume 6173 of *LNCS*, pages 142–148. Springer, 2010. `doi:10.1007/978-3-642-14203-1_12`.

11 Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. Cosette: An automated prover for SQL. In *CIDR 2017*. www.cidrdb.org, 2017. URL: `http://cidrdb.org/cidr2017/papers/p51-chu-cidr17.pdf`.

12 Sára Decova. Modelling and verification of the Timely Dataflow progress tracking protocol. Master's thesis, ETH Zurich, Zurich, 2020. `doi:10.3929/ethz-b-000444762`.

13 Tomás Díaz, Federico Olmedo, and Éric Tanter. A mechanized formalization of GraphQL. In Jasmin Blanchette and Catalin Hritcu, editors, *CPP 2020*, pages 201–214. ACM, 2020. `doi:10.1145/3372885.3373822`.

14 Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. Verifying strong eventual consistency in distributed systems. *Proc. ACM Program. Lang.*, 1(OOPSLA):109:1–109:28, 2017. `doi:10.1145/3133933`.

15 Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage systems are distributed systems (so verify them that way!). In *OSDI 2020*, pages 99–115. USENIX Association, 2020. URL: `https://www.usenix.org/conference/osdi20/presentation/hance`.

16 Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In Ethan L. Miller and Steven Hand, editors, *SOSP 2015*, pages 1–17. ACM, 2015. `doi:10.1145/2815400.2815428`.

17 Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris: session-type based reasoning in separation logic. *Proc. ACM Program. Lang.*, 4(POPL):6:1–6:30, 2020. `doi:10.1145/3371074`.

18 C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. `doi:10.1145/359576.359585`.

19 Brian Huffman and Ondrej Kuncar. Lifting and Transfer: A modular design for quotients in Isabelle/HOL. In Georges Gonthier and Michael Norrish, editors, *CPP 2013*, volume 8307 of *LNCS*, pages 131–146. Springer, 2013. `doi:10.1007/978-3-319-03545-1_9`.

20 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`.

21 Leslie Lamport. Paxos made simple, fast, and Byzantine. In Alain Bui and Hacène Fouchal, editors, *OPODIS 2002*, volume 3 of *Studia Informatica Universalis*, pages 7–9. Suger, Saint-Denis, rue Catulienne, France, 2002.

22 Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: certified causally consistent distributed key-value stores. In Rastislav Bodík and Rupak Majumdar, editors, *POPL 2016*, pages 357–370. ACM, 2016. `doi:10.1145/2837614.2837622`.

23 J. Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In Manuel V. Hermenegildo and Jens Palsberg, editors, *POPL 2010*, pages 237–248. ACM, 2010. `doi:10.1145/1706299.1706329`.

24 Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. Shared arrangements: practical inter-query sharing for streaming dataflows. *Proc. VLDB Endow.*, 13(10):1793–1806, 2020. URL: `http://www.vldb.org/pvldb/vol13/p1793-mcsherry.pdf`.

25 Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *CIDR 2013*. www.cidrdb.org, 2013. URL: `http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf`.

26 Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In Michael Kaminsky and Mike Dahlin, editors, *SOSP 2013*, pages 439–455. ACM, 2013. `doi:10.1145/2517349.2522738`.

27 Derek Gordon Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martín Abadi. Incremental, iterative data processing with timely dataflow. *Commun. ACM*, 59(10):75–83, 2016. `doi:10.1145/2983551`.

28 Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In Garth Gibson and Nickolai Zeldovich, editors, *USENIX ATC 2014*, pages 305–319. USENIX Association, 2014. URL: `https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro`.

29 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 Series in Computing*, pages 1–11. EasyChair, 2010. URL: `https://easychair.org/publications/paper/wV`.

30 Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *Proc. ACM Program. Lang.*, 2(POPL):28:1–28:30, 2018. `doi:10.1145/3158116`.

31 Christoph Sprenger, Tobias Klenze, Marco Eilers, Felix A. Wolf, Peter Müller, Martin Clochard, and David A. Basin. Igloo: soundly linking compositional refinement and separation logic for distributed system verification. *Proc. ACM Program. Lang.*, 4(OOPSLA):152:1–152:31, 2020. doi:10.1145/3428220.

32 Makarius Wenzel. Isabelle/Isar – A generic framework for human-readable proof documents. In Roman Matuszewski and Anna Zalewska, editors, *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar, and Rhetoric*. Uniwersytet w Białymstoku, 2007.

33 James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In David Grove and Steve Blackburn, editors, *PLDI 2015*, pages 357–368. ACM, 2015. doi:10.1145/2737924.2737958.

Syntactic-Semantic Form of Mizar Articles

Czesław Byliński
Computer Networks Section, University of Bialystok, Poland

Artur Korniłowicz[1]
Institute of Computer Science, University of Bialystok, Poland

Adam Naumowicz
Institute of Computer Science, University of Bialystok, Poland

Abstract

Mizar Mathematical Library is most appreciated for the wealth of mathematical knowledge it contains. However, accessing this publicly available huge corpus of formalized data is not straightforward due to the complexity of the underlying Mizar language, which has been designed to resemble informal mathematical papers. For this reason, most systems exploring the library are based on an internal XML representation format used by semantic modules of Mizar. This representation is easily accessible, but it lacks certain syntactic information available only in the original human-readable Mizar source files. In this paper we propose a new XML-based format which combines both syntactic and semantic data. It is intended to facilitate various applications of the Mizar library requiring fullest possible information to be retrieved from the formalization files.

2012 ACM Subject Classification Information systems → Extensible Markup Language (XML)

Keywords and phrases Mizar system, mathematical knowledge representation, XML representation

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.11

Supplementary Material A collection of prototype hyper-linked Mizar articles generated from the new ESM syntactic-semantic format can be found at
Dataset: `http://alioth.uwb.edu.pl/~artur/mmlesx/`

Acknowledgements The Mizar processing has been performed using the infrastructure of the University of Bialystok High Performance Computing Center.

1 Introduction

Since the beginning of the Mizar project [5], the *verifier* has been the main computer program of the whole system. Its function is to check user-provided input files (called "Mizar articles") in terms of syntactic correctness and logical validity of mathematical content encoded in the Mizar language. The application has been designed analogously to building compilers, so it has a lexical and syntactic analysis modules as well as a module for performing the semantic analysis of the source text. But of course the Mizar verifier does not generate any executable code. Instead, the last module confirms that a given article does not contain errors, or otherwise reports a list of errors found in the text. Hence, the absence of errors in the result of text analysis means the correctness of the text in terms of syntactic, semantic and logical content. In the initial Mizar implementations, this was the sole result of running the verifier, i.e. the tool provided only the information about encountered errors in the form of a text listing file.

The development of the Mizar language conducted under constant leadership of Andrzej Trybulec was reinforced by the experience related to the creation of new Mizar texts as well as experiments with various limited versions of Mizar (e.g. Mizar MSE, Mizar HPF) [12]. At

[1] Corresponding author

the same time, the source code of Mizar verifiers evolved as a result of implementations on subsequent computers. Notably, Mizar 2 (the predecessor of current Mizar) was implemented on one of Polish third-generation computers, Odra 1305, using the Pascal compiler available for ICL 1900 machines. This version of the compiler was a port of Pascal P2 from the CDC 6000 machine to ICL 1900. At that time, the Mizar language was designed in such a way, so that the parsing, semantic analysis, and logical correctness procedures could be implemented as a one-pass program on the mainframe computer. The coding made use of the "top-down" approach with main algorithms based on recursive procedures. Then, Mizar 2 code was transferred from Odra 1305 to IBM 360 using Pascal P8000 with small code changes and later formed the base of implementing Mizar on the PDP-11 minicomputer. Due to the memory limitations of that machine, a one-pass implementation was not possible and therefore the verifier was divided into a series of passes implementing the subsequent stages of Mizar text processing and analysis. This change has become a permanent feature of the system since then. Initially there were seven passes which covered tokenization, syntax analysis, identifier analysis, semantic analysis of Mizar's linguistic structures, checking proof structure, inference correctness and schematization. The division into the passes was due to both content-related and technical reasons. The information about subsequent results of the analysis between the passes was transmitted via text files. The syntax of these files was strictly technical, it was only about providing information necessary for further analysis. The communication files were treated as temporary working files, generated only for the duration of the verifier's work. The final result of the text analysis was realized a bit differently than in Mizar 2. Namely, one error file was created, to which specific errors detected by individual passes were added. When the Mizar verifier was ported from PDP-11 to the IBM PC machines using the Turbo Pascal compiler, the number of passes was limited to four. At that time, the Mizar script still contained an axiomatic part describing the mathematical theory necessary to formulate new definitions and prove theorems.

A next significant change in the Mizar implementation was related to the creation of the Mizar Mathematical Library (MML) [4]. The Mizar script took the modern form of current Mizar articles, containing the so-called environment part (MML references) and the actual text. The verifier, or actually additional programs based on the code of the verifier, have been used to create library files describing all the notions, definitions and theorems introduced and proven in a given article. Current MML provides an interrelated library built from axiomatic foundations (Tarski-Grothendieck set theory) in which all derived facts are positively checked by the Mizar verifier and can be included in the MML only under this condition. Further development of Mizar software and the MML are now closely inter-related. Each new version of the Mizar software must be MML compliant. And on the other hand, after any language changes, MML must be refactored to be compatible with the new Mizar system.

At some stage of work on the implementation of the Mizar verifier (around 2004) and, more broadly, the Mizar system, the structure of MML database files and intermediate files transmitting information between the verifier's modules was changed. Instead of specific internal formats available only from within the Pascal (Delphi and Free Pascal) implementation of the system, a new XML-based form was introduced [17]. Initially it was only a technical change, but soon the more easily accessible information about Mizar articles started to be used by several external applications (c.f. [8, 18, 7]). Most of these applications have utilized the XML file carrying the information passed to the Mizar inference checker and schematizer modules. However, the disadvantage of these files is that they contain already pre-processed semantic information needed for the checker, which only partially allows reconstructing the original content of the article needed for some applications. As such they are not always suitable for use independently of the Mizar system, e.g. for the needs of formal systems other than Mizar. In recent years, several attempts have been made to use the MML for other proof systems (c.f. [9, 20, 10]). So there emerged a need to provide readily-available information about the content of the MML without the necessity of analyzing Mizar articles. The idea was to develop a format describing Mizar articles in which the formalizations contained in the MML could become accessible by any semantic mathematics databases or other formal systems. Consequently, the current versions of the Mizar verifier, in addition to the basic user feedback, i.e. the confirmation that a processed article is correct, also generate the description of the article in a syntactic form with the associated semantics as XML files. In 2012, the creation of files describing the syntactic form of Mizar articles was re-organized. The corresponding files are generated by the two initial modules of the Mizar verifier: the parser which creates a description of the syntactic tree of the analyzed article in a file technically named WSM (Weakly Strict Mizar) [7, 14], and an identification module which creates an MSM file (More Strict Mizar). MSM files are descriptions of the syntax of the Mizar article with additional information about the identifiers used: sentence and variable identifiers. However, they do not contain information about the exact constructors used and being defined. Moreover, they do not provide unambiguous information that the Mizar system uses to process the whole MML.

Recently (2020), a new variant of the Mizar verifier's Analyzer module has been implemented. This module is responsible for the semantic analysis of the Mizar language structure. Now it is possible to generate semantic information, while preserving the syntactic information used in the original Mizar source text. In previous versions of the Analyzer module, all expressions were at that point transformed into the form of so called "semantic correlates" used by the Checker module. This resulted in the loss of information about their syntactic form and rendered the exact reconstruction of the original syntax impossible, because the transformation is irreversible [15]. The most recent implementation of the Analyzer preserves enough information to create an article description in the form of a syntactic tree extended with associated semantic data. The new data format was named ESM (Even more Strict Mizar). The corresponding file generated by the Mizar verifier contains an exact description of the syntactic structure of the article in connection with the resolved semantic information. The disambiguation concerns all entered symbols and the semantics of definitions, expressions and theorems introduced and used in the article. Moreover, unlike the previously available data formats storing only the information necessary for proof checking by the Mizar verifier, the ESM representation is enriched with absolute references to database items (within a particular MML version), which can be understood regardless of the local environment of a given article. This information is eventually suitable for direct use by any formal systems independently of the Mizar software.

2 Even more Strict Mizar (ESM)

In this section we present the extension of previously available WSM and MSM data representation formats (generated by the Mizar verifier as intermediate files with corresponding `.wsx` and `.msx` filename extensions). Although the ESM new data format gives access to information currently stored in the XML (`.xml`) files generated by the Mizar analyzer module, it does not replace the old representation completely, since the latter is still used as the main input for the checker pass. The advantage of ESM, however, is that this representation of Mizar texts can be used independently from the dedicated Mizar proof checking software.

Table 1 Constructors (C) and patterns (P): + means introduction of a new constructor/pattern.

	Predicate		Attribute		Mode		Functor	
	C	P	C	P	C	P	C	P
definition	+	+	+	+	+	+	+	+
redefinition of definiens		+		+		+		+
redefinition of result type	n/a	n/a	n/a	n/a	+	+	+	+
redefinition with properties	+	+	n/a	n/a	n/a	n/a	+	+
synonym		+		+		+		+
antonym		+		+	n/a	n/a	n/a	n/a

2.1 Mizar definitions

The Mizar language allows users to define mathematical notions of several basic categories: predicates, adjectives, types, operations and structures. Using the Mizar terminology, various aspects of all these definitions are internally represented either as *formats*, *constructors*, or *patterns*. Formats represent syntactic information about the kind of symbol used in the definition together with the number and position of arguments. Constructors, on the other hand, provide a numbering system for representing the semantics of mathematical objects. The unique numbering scheme is provided independently within each category of defined notions. And finally patterns represent joint information about the used format, constructor, types of all arguments, and positions of visible arguments for a given definition. The distinction between constructors and patterns is necessary to accommodate various linguistic features like synonyms or redefinitions available in the Mizar language which offer different contextual ways of representing semantically equal objects. In principle, there are more patterns than constructors since not all definitions introduce new constructors, see Table 1. In the current Mizar implementation, the information about all available and newly defined notions is scattered across multiple intermediate XML-based resource files. The next section presents basic technical details concerning these files (identified by their respective filename extensions by Mizar utilities) necessary for developing software which makes use of all definition-related information.

2.2 Intermediate file types

First of all, new symbols introduced to denote concepts defined in a Mizar article are collected in the `.dcx` file, which also includes information about all Mizar reserved words used in the processed article. Each symbol is represented by its category (`kind`), number (`nr`), and spelling (`name`). It should be noted that symbols of different categories are numbered separately.

After the article is parsed, the formats are collected in `.frx` files. Each format is represented by `kind`, format number (`nr`), symbol number (`symbolnr`, computable from `.dcx`), number of visible arguments (`argnr`), including the number of left arguments (`leftargnr`). The number of right arguments can obviously be computed as the difference between the values of `argnr` and `leftargnr`. Notably, the numbering is continuous here, i.e. it is common to all kinds of definitions introducing the formats. Moreover, it should be noted that not all definitions introduce new formats – if the notation signatures are repeated, only one copy of the format is collected. Specific binding force of operations (if different from the default value) is also included in the format files, but it is not used to distinguish between formats.

The internal constructor descriptions are scattered across two files. Firstly, constructors imported from the database are collected in `.aco` files which consist of two sections. The first section, marked with the `<SignatureWithCounts>` element, contains incremental sums of constructors added by successive files imported from the database. By performing appropriate calculations on these sums, it is possible to reconstruct the name of the article and the number of the definition introducing a given constructor in later procedures. That section is followed by a list with descriptions of all imported constructors. Such a description contains a number of attributes representing the constructor's category (`kind`), the name of the article (`aid`) in which the constructor was introduced, the number (`nr`) in this article, and the relative number (`relnr`) of a given constructor in relation to all constructors available in a given article, the types of arguments, result types, and the indication whether a given constructor introduces any property. Constructors are numbered in their respective categories, i.e., mode constructors separately from predicate constructors, and so on. Constructors of concepts introduced in a given article are collected in `.xml` files. Their descriptions are consistent with those of imported constructors.

The most extensive description of defined objects is represented by patterns. Patterns imported from the database are stored in `.eno` files and their content is based on the following attributes:

- `kind` – category of the pattern
- `nr` – number within one imported file
- `aid` – imported article file name
- `formatnr` – number of used format
- `constrkind` – kind of used constructor
- `constrnr` – number of used constructor
- `relnr` – relative number w.r.t. a category

With the new ESM language representation, newly defined patterns are stored by the verifier in `.esx` files (in previous releases of the Mizar system, they were stored in `.xml` files). Their extended description is based on the following attributes:

- `spelling` – spelling of the token (retrieved from the format)
- `position` – position of the token
- `formatdes` – description of used format (retrieved from the format)
- `formatnr` – number of used format
- `patternnr` – number of the pattern within a category w.r.t. the environment
- `absolutepatternMMLId` – unique identifier of the pattern w.r.t. the whole MML, but within a pattern category, e.g., "FUNCT_2:4" where FUNCT_2 is an MML article identifier
- `origpatternnr` – number of the original (being redefined) pattern within a category w.r.t. the environment in redefinitions
- `absoluteorigpatternMMLId` – unique identifier of the original (being redefined) pattern w.r.t. the whole MML, but within groups, in redefinitions
- `constr` – category (kind) and number of the constructor w.r.t. the environment, e.g., "V4" where V represents the category of adjectives
- `absoluteconstrMMLId` – unique identifier of the constructor w.r.t. the whole MML, but within a given category
- `origconstrnr` – number of the original (being redefined) constructor w.r.t. the environment in redefinitions
- `absoluteorigconstrMMLId` – unique identifier of the original (being redefined) constructor w.r.t. the whole MML, but within a given category, in redefinitions.

To summarize, Table 2 shows the location of definition-related data in Mizar resource files.

Table 2 Location of definitional data in resource files.

Resource	Imported	Defined
symbol	`.dcx`	`.dcx`
format	`.frx`	`.frx`
constructor	`.aco`	`.xml`
pattern	`.eno`	`.xml` (also `.esx`)

Below we present a series of snippets of Mizar code and corresponding ESM representations (underlined attributes with values in bold face extend the information inherited from WSM and MSM formats). We start with a definition of the first projection (selecting first elements of contained pairs). This operation defined for an arbitrary set in the MML article `XTUPLE_0` is presented in Listing 1.

Listing 1 Definition of the first projection

```
definition
  let X be set;
  func proj1 X -> set means
:: XTUPLE_0:def 12
  x in it iff ex y st [x,y] in X;
  correctness;
end;
```

The underlying ESM representation of the definition is given in Listing 2. The `Standard-Type` element within `Loci-Declaration` corresponds to the type of the argument, i.e. "set" introduced in the MML article `HIDDEN`. Moreover, the `Type-Specification` element indicates that in this general context the result type of the operation is also a set.

Listing 2 ESM representation of the first projection (definiens is elided)

```
<Block kind="Definitional-Block" position="192\10" endposition="202\3">
<Item kind="Loci-Declaration" position="193\5" endposition="193\14">
<Loci-Declaration>
 <Qualified-Segments>
  <Explicitly-Qualified-Segment position="193\7">
   <Variables>
    <Variable idnr="25" spelling="X" position="193\7" kind="Constant"
     serialnr="54" varnr="1"/>
   </Variables>
   <Standard-Type nr="1" formatnr="3" patternnr="2" absolutepatternMMLId="HIDDEN:2"
     spelling="set" sort="Mode" constrnr="2" absoluteconstrMMLId="HIDDEN:2"
     originalnr="0" position="193\14">
    <Arguments/>
   </Standard-Type>
  </Explicitly-Qualified-Segment>
 </Qualified-Segments>
</Loci-Declaration>
</Item>
<Item kind="Functor-Definition" position="194\6" endposition="197\32">
<Functor-Definition MMLId="XTUPLE_0:12">
 <Redefine occurs="false"/>
 <InfixFunctor-Pattern formatdes="O43[0(1)1]" formatnr="37" spelling="proj1"
  position="194\12" patternnr="21" absolutepatternMMLId="XTUPLE_0:12" constr="K18"
  absoluteconstrMMLId="XTUPLE_0:9" origconstrnr="0">
  <Loci/>
  <Loci>
   <Locus idnr="25" varidkind="Identifier" spelling="X" position="193\7" origin="Constant"
    kind="Constant" serialnr="54" varnr="1"/>
  </Loci>
 </InfixFunctor-Pattern>
 <Type-Specification>
  <Standard-Type nr="1" formatnr="3" patternnr="2" absolutepatternMMLId="HIDDEN:2"
   spelling="set" sort="Mode" constrnr="2" absoluteconstrMMLId="HIDDEN:2" originalnr="0"
   position="198\21">
   <Arguments/>
  </Standard-Type>
 </Type-Specification>
 <Definiens>...</Definiens>
</Block>
```

Then, if we consider a relation, then it is more natural to call this projection the domain of the relation. Hence, as in the article `RELAT_1`, we can introduce a convenient synonym for this operation (for the same constructor) under the assumption that the argument is of the relation type:

Listing 3 Domain as a synonym for the first projection

```
notation :: RELAT_1
  let R be Relation;
  synonym dom R for proj1 R;
end;
```

Listing 4 ESM representation of the domain

```
<Block kind="Notation-Block" position="103\8" endposition="106\3">
<Item kind="Loci-Declaration" position="104\5" endposition="104\19">
<Loci-Declaration>
 <Qualified-Segments>
  <Explicitly-Qualified-Segment position="104\7">
   <Variables>
    <Variable idnr="29" spelling="R" position="104\7"
              kind="Constant" serialnr="31" varnr="1"/>
   </Variables>
   <Standard-Type nr="5" formatnr="44" patternnr="6" position="104\19"
    spelling="Relation" sort="Expandable-Type" absolutepatternMMLId="RELAT_1:1">
    <Arguments/>
   </Standard-Type>
  </Explicitly-Qualified-Segment>
 </Qualified-Segments>
</Loci-Declaration>
</Item>
<Item kind="Func-Synonym" position="105\13" endposition="105\27">
<Func-Synonym>
 <InfixFunctor-Pattern formatdes="O12[0(1)1]" formatnr="45" spelling="dom"
```

```
  origpatternnr="16" absoluteorigpatternMMLId="XTUPLE_0:12" patternnr="27"
  absolutepatternMMLId="RELAT_1:1" constr="K32"
  absoluteconstrMMLId="XTUPLE_0:9" origconstrnr="0" position="105\13">
  <Loci/>
  <Loci>
   <Locus idnr="29" varidkind="Identifier" spelling="R" position="104\7" origin="Constant"
    kind="Constant" serialnr="31" varnr="1"/>
  </Loci>
 </InfixFunctor-Pattern>
```

Finally, when the relation happens to be defined on a given set X, we may use this information as in the article `RELSET_1` to redefine the domain with a more specific result type, i.e. a subset of X (this creates a new constructor):

Listing 5 Redefinition of the domain

```
definition :: RELSET_1
  let X be set;
  let R be X-defined Relation;
  redefine func dom R -> Subset of X;
end;
```

Listing 6 Excerpt from ESM representation of the redefinition

```
<Functor-Definition>
 <Redefine occurs="true"/>
 <InfixFunctor-Pattern formatdes="O2[0(1)1]" formatnr="45" spelling="dom" position="113\18"
  origpatternnr="23" absoluteorigpatternMMLId="RELAT_1:1" patternnr="35"
  absolutepatternMMLId="RELSET_1:1" constr="K48"
  absoluteconstrMMLId="RELSET_1:1" origconstrnr="42"
  absoluteorigconstrMMLId="XTUPLE_0:9">
  <Loci/>
  <Loci>
   <Locus idnr="22" varidkind="Identifier" spelling="R" position="112\23" origin="Constant"
    kind="Constant" serialnr="40" varnr="2"/>
  </Loci>
 </InfixFunctor-Pattern>
 <Type-Specification>
  <Standard-Type nr="5" formatnr="34" patternnr="4" position="113\30"
   spelling="Subset" sort="Expandable-Type" absolutepatternMMLId="SUBSET_1:2">
   <Arguments>
    <Simple-Term idnr="11" spelling="X" position="113\35" origin="ReservedVar"
     sort="Constant" serialnr="3" varnr="1"/>
   </Arguments>
  </Standard-Type>
 </Type-Specification>
```

Thanks to the absolute references provided within the ESM format it is possible to easily distinguish the two notions represented by different constructors (`absoluteconstrMMLId = "XTUPLE_0:9"` and `absoluteconstrMMLId = "RELSET_1:1"`) and three different patterns (`absolutepatternMMLId = "XTUPLE_0:12"`, `absolutepatternMMLId = "RELAT_1:1"` and `absolutepatternMMLId = "RELSET_1:1"`) used when applying this operation in various contexts.

2.3 Decoding Mizar definitions

All the available internal formats may still be used for specific tasks related to processing Mizar articles. However, in many cases the implementation may significantly benefit from utilizing the ESM format. The potential transition requires understanding both the formerly used data structures and the new ESM capabilities. For example, in order to demonstrate how the use of ESM facilitates the unique identification of all components of formulas, let us first analyze the formula `A c= A \/ B` using only its XML representation:

`.xml` file	`.aco` file
`<Pred kind="R" nr="3" pid="4">`	`<SignatureWithCounts>`
`<Var nr="1"/>`	`<ConstrCounts name="HIDDEN">`
`<Func kind="K" nr="6" pid="16">`	`<ConstrCount kind="M" nr="2"/>`
`<Var nr="1"/>`	`<ConstrCount kind="R" nr="2"/>`
`<Var nr="2"/>`	`</ConstrCounts>`
`</Func>`	`<ConstrCounts name="TARSKI">`
`</Pred>`	`<ConstrCount kind="M" nr="2"/>`
	`<ConstrCount kind="R" nr="5"/>`
	`<ConstrCount kind="K" nr="4"/>`
	`</ConstrCounts>`
	`<ConstrCounts name="XBOOLE_0">`
	`<ConstrCount kind="M" nr="2"/>`
	`<ConstrCount kind="V" nr="1"/>`
	`<ConstrCount kind="R" nr="8"/>`
	`<ConstrCount kind="K" nr="9"/>`
	`</ConstrCounts>`

To identify which inclusion and which union are used in the formula, the following steps should be done:

1. From values `kind="R"` and constructor number `nr="3"` and content of `.aco` file we can conclude that the inclusion is the first predicate in `TARSKI` article (numeral 3 is bigger than 2 in the line `<ConstrCount kind="R" nr="2"/>` and lower than 5 in the line `<ConstrCount kind="R" nr="5"/>`).
2. From values `kind="R"` and pattern number `pid="4"` and the line:
   ```
   <Pattern kind="R" nr="1" aid="TARSKI" formatnr="7"
     constrkind="R" constrnr="3" relnr="4">
   ```
   of `.eno` file (`pid="4"` = `relnr="4"`) we know the format number `formatnr="7"`.
3. From values `kind="R"` and `formatnr="7"` and the line:
   `<Format kind="R" nr="7" symbolnr="4" argnr="2" leftargnr="1"/>`
   of `.frx` file (`formatnr="7"` = `nr="7"`) we know the number of used symbol `symbolnr="4"`, and we know that the predicate has two arguments (`argnr="2"`) and arguments are placed as one on the left side of the symbol (`leftargnr="1"`) and one on the right side of the symbol.
4. From `kind="R"` and `symbolnr="4"` and the line:
   `<Symbol kind="R" nr="4" name="c="/>` of `.dcx` file (`symbolnr="4"` = `nr="4"`) we conclude that `c=` is used as a symbol of the relation (`name="c="`).
5. Similar reasoning can be done about the operation coded as `kind="K"` and `nr="6"` .

And when we look at the corresponding piece of the `.esx` file:

Table 3 XML elements of basic concepts.

Terms	Types	Formulas
Aggregate-Term	Clustered-Type	Biconditional-Formula
Circumfix-Term	ReservedDscr-Type	Conditional-Formula
Forgetful-Functor-Term	Standard-Type	Conjunctive-Formula
Fraenkel-Term	Struct-Type	Disjunctive-Formula
Global-Choice-Term		Existential-Quantifier-Formula
Infix-Term		FlexaryConjunctive-Formula
Internal-Selector-Term		FlexaryDisjunctive-Formula
Numeral-Term		Multi-Attributive-Formula
Placeholder-Term		Multi-Relation-Formula
Private-Functor-Term		Negated-Formula
Qualification-Term		Private-Predicate-Formula
Selector-Term		Qualifying-Formula
Simple-Fraenkel-Term		Relation-Formula
Simple-Term		RightSideOf-Relation-Formula
it-Term		Universal-Quantifier-Formula

Listing 7

```
<Relation—Formula nr="4" formatnr="7" patternnr="4" absolutepatternMMLId="TARSKI:1"
     leftargscount="1" spelling="c=" sort="Relation—Formula" constrnr="3"
     absoluteconstrMMLId="TARSKI:1" originalnr="0" position="11\4">
 <Arguments>
  <Simple—Term idnr="4" spelling="A" position="11\1" origin="ReservedVar"
       sort="BoundVar" serialnr="1" varnr="1"/>
 <Infix—Term nr="17" formatnr="37" patternnr="16" absolutepatternMMLId="XBOOLE_0:2"
       leftargscount="1" spelling="\/" sort="Functor—Term" constrnr="6"
       absoluteconstrMMLId="XBOOLE_0:2" originalnr="0" position="11\9">
   <Arguments>
    <Simple—Term idnr="4" spelling="A" position="11\6" origin="ReservedVar"
         sort="BoundVar" serialnr="1" varnr="1"/>
    <Simple—Term idnr="5" spelling="B" position="11\11" origin="ReservedVar"
         sort="BoundVar" serialnr="2" varnr="2"/>
   </Arguments>
  </Infix—Term>
 </Arguments>
</Relation—Formula>
```

we can see that all this information is accessible in one place and no extra computations are required to identify used notions.

3 Main ESM grammar items

In this section we show a selection of ESM grammar items and examples of corresponding Mizar code. Presented data types are specifically related to the extension with respect to earlier WSM and MSM representations. Table 3 lists the names of all possible basic formula, type and term kinds now shared by all strict Mizar formats.

All various Mizar term categories are presented in Table 4 with examples of their applicability. Four possible basic Mizar type variants are represented as elements listed in Table 5, while Table 6 presents available formula categories, respectively.

Table 4 XML elements of terms with examples.

Term	Description	Example
Aggregate-Term	tuple structure	`ZeroStr(#A,a#)`
Circumfix-Term	bracketed term	`[:A,B:]`
Forgetful-Functor-Term	sub-tuple structure	`the ZeroStr of Gr`
Fraenkel-Term	Fraenkel operator	`{n where n is Nat: n is odd}`
Global-Choice-Term	global-choice operator	`the set`
Infix-Term	standard term	`A \/ B`
Internal-Selector-Term	structure selector (inside its definition)	`the carrier`
Numeral-Term	numeral	`1`
Placeholder-Term	argument of private definitions	`$1`
Private-Functor-Term	private functor (`deffunc`)	`F(1)`
Qualification-Term	type-cast operator	`1 qua Element of REAL`
Selector-Term	structure selector (with argument)	`the carrier of Gr`
Simple-Fraenkel-Term	Fraenkel operator	`the set of all n where n is Nat`
Simple-Term	constant	`a`
it-Term	definiendum representation	`it`

Table 5 XML elements of types with examples.

Types	Description	Example
Clustered-Type	type with adjectives	`finite set`
ReservedDscr-Type	dependent type in reservations	`Element of V`
Standard-Type	standard type	`object`
Struct-Type	structural type	`1-sorted`

3.1 Structures

Comparing to the information previously stored in WSM, the description of defined structures is significantly extended. Let us consider the following structure definition as an example.

```
reserve S1,S2 for 1-sorted;

definition
  let S1;
  let S2 be 1-sorted;
  struct (ModuleStr over S1, RightModStr over S2) BiModStr over S1,S2
  (#
    carrier -> set,
    addF, multF -> BinOp of the carrier,
    ZeroF, OneF -> Element of the carrier,
    lmult -> Function of [:the carrier of S1, the carrier:], the carrier,
    rmult -> Function of [:the carrier, the carrier of S2:], the carrier
  #);
end;
```

The same definition with annotations from corresponding pieces of `.esx` file may look like this:

Table 6 XML elements of formulas with examples.

Formulas	Description	Example
Biconditional-Formula	equivalence	`iff`
Conditional-Formula	implication	`implies`
Conjunctive-Formula	conjunction	`&`
Disjunctive-Formula	disjunction	`or`
Existential-Quantifier-Formula	existentially quantified formula	`ex x st P[x]`
FlexaryConjunctive-Formula	flexary conjunctive	`& ... &`
FlexaryDisjunctive-Formula	flexary disjunction	`or ... or`
Multi-Attributive-Formula	attributive formula	`{} is empty`
Multi-Relation-Formula	chain formula	`A c= B c= C`
Negated-Formula	negation	`not`
Private-Predicate-Formula	private predicate (`defpred`)	`P[set,Nat]`
Qualifying-Formula	explicit type qualification	`1 is object`
Relation-Formula	standard formula	`1 <= 2`
RightSideOf-Relation-Formula	tail of chain formula	`c= C`
Universal-Quantifier-Formula	universally quantified formula	`for x holds P[x]`

```
definition
  let S1;
     :: <Loci-Declaration> / <Qualified-Segments> / <Implicitly-Qualified-Segment>
  let S2 be 1-sorted;
     :: <Loci-Declaration> / <Qualified-Segments> / <Explicitly-Qualified-Segment>
  struct
  (ModuleStr over S1, RightModStr over S2) :: <Structure-Definition> / <Ancestors>
  BiModStr :: <Structure-Pattern>
  over S1,S2 :: <Structure-Pattern> / <Loci>
  (#  :: <Field-Segments>
  carrier :: <Field-Segment> / <Selectors>
  ->
  set, :: <Field-Segment> / <Standard-Type>
  addF, multF :: <Field-Segment> / <Selectors>
  ->
  BinOp of the carrier, :: <Field-Segment> / <Standard-Type>
  ZeroF, OneF -> Element of the carrier,
  lmult -> Function of [:the carrier of S1, the carrier:], the carrier,
  rmult -> Function of [:the carrier, the carrier of S2:], the carrier
  #);
end;
```

Apart from the representation of visible syntactic elements, the XML data structure of ESM now contains the following extra elements enclosed within the `<Structure-Patterns-Rendering>` container.

1. `<AggregateFunctor-Pattern>` representing the patterns for tuple terms encoded as `<Aggregate-Term>`. Aggregates represent concrete full structures. For example rings of integers with addition and multiplication as ring operations.
2. `<ForgetfulFunctor-Pattern>` representing the patterns for substructure terms `<Forgetful-Functor-Term>`. Forgetful terms can represent full structures or substructures which are ancestors (direct or indirect) of the structure, for example `the 1-sorted of B` is an indirect and `the ModuleStr of B` is a direct ancestor of some `B` of type `BiModStr`.

3. `<Strict-Pattern>` defining a special adjective `strict`. For example a ring is not a strict group, because it contains more selectors than those of a group. `strict` is generated automatically when a definition of a structure occurs. Regular adjectives, like `empty`, `finite`, etc. must be defined within regular definitional blocks.
4. `<Selectors-List> / <SelectorFunctor-Pattern>` representing the patterns of terms of the category encoded as `<Selector-Term>` Selector terms represent one field of a given structure, for example `the carrier of B`.

4 Applications

The proposed representation format is intended to facilitate various applications based on Mizar formalizations. Its validity and completeness has been initially tested and demonstrated with a collection of HTML files generated from Mizar source files allowing precise browsing through the library and exploring semantic links between notions. Another natural application of the extended ESM representation is in the system used by the journal Formalized Mathematics (FM)[2].

4.1 Linked Mizar Mathematical Library

As Mizar articles in the MML library had been continuously revised while Formalized Mathematics published their state at the time of the publication, an electronic counterpart of FM, called Journal of Formalized Mathematics (JFM), was developed in 1995 with the intention of representing the updated MML. The project initially funded by the ONR Grant N00014-95-1-1336 *Automated hyper-linking in an electronic mathematical proof-check journal* [3] continued till 2004. A central part of the project was a collection of HTML files generated from Mizar source files which offered users intuitive browsing through the library and exploring the inter-linked notions. After introducing in 2004 the internal XML-based format (`.xml` files) representing the result of the Mizar verifier's analyser pass, J. Urban used it to re-implement the HTML[4] linking part of JFM using XSL and extending the original format with useful extra functionality, e.g. rendering of complete proofs. Since then, the technology was frequently used by various external systems based on the MML semantic connections and linking (c.f. [3, 19, 16, 1, 11, 13]).

It should be noted that Urban's HTML rendering of Mizar articles is enriched with various elements invisible in the content of the original Mizar texts, and generated during the verification, such as: definitional theorems (the internal representation of definitions in the form of equivalence theorems) or expanded attribute clusters (sets of attributes appearing in the text extended with automatically calculated consequences based on registrations available in the environment). The representation thus realized is therefore richer than the content of the article seen at the level of the Mizar text. However, generating HTML documents on the basis of `.xml` files involves the necessity of multiple "recalculations" of numerical representations of objects (formats, patterns or constructors). It also loses the original structure of logical formulas as a result of translating these formulas into semantic correlates needed by the checker or expanding local variables introduced with the `set` construct).

[2] ISSN 1426-2630 (Print), eISSN 1898-9934 (Online), `http://fm.mizar.org`
[3] `https://apps.dtic.mil/dtic/tr/fulltext/u2/a322951.pdf`
[4] Available on-line at `http://mizar.uwb.edu.pl/version/current/html/`

Table 7 Examples of link anchor names.

kind	name
attribute	`#articlename_V1_P1`
predicate	`#articlename_R1_P1`
functor	`#articlename_K1_P1`
expandable mode	`#articlename_ME1_P1`
regular mode	`#articlename_M1_P1`
selector	`#articlename_U1_P1`
aggregate	`#articlename_G1_P1`
structure	`#articlename_L1_P1`
theorem	`#ARTICLENAME:1`
definition	`#ARTICLENAME:def1`
scheme	`#ARTICLENAME:sch1`
local reference	`#Lab_S1_L1`
local predicate	`#PrP_S1_V1`
local functor	`#PrF_S1_V1`

The potential utility of our new ESM syntactic-semantic format (`.esx` files) can also be demonstrated by a similar collection of hyper-linked Mizar articles (see Supplementary Material). In comparison to the representation generated and partially reconstructed from the `.xml` files, it is now possible to render the semantic connections between linked notions and at the same preserve the original syntactic structure of the underlying Mizar text.

The ready-available information was used to implement a system of links (anchor names) for the definitions of:

- all basic concepts, i.e. predicates, adjectives, types, operations and structures
- local predicates and local functors,
- local labels and references to external statements, definitions, and schemas.

The links from concepts to their definitions consist of: the ID of the article in which the term was defined, concept type (`V, R, M, K, O, L, G, U`), constructor number of a given concept and the pattern number of a given concept preceded by the letter `P`. Table 7 presents the details of the link naming convention used.

As an example, let us look at the representation of the adjective `odd` defined as the antonym of `even` in the article `abian.miz`. `odd` is represented as `abian_V1_P2`. We can see the difference between the constructor number (1) and the pattern number (2). It results from the fact that antonyms generate a new pattern, and do not generate a new constructor (antonyms inherit the constructor numbers of the originals).

Links to local predicates consist of the `PrP` prefix, a serial number preceded by `S` and the current predicate number available in the given reasoning block preceded by `V`. Links to local functors have the same structure but with the `PrF` prefix. Finally, links to local labels use the prefix `Lab`.

References to theorems, definitions, and schemes consist of the article identifier from which the item is derived and the corresponding number preceded by `def` for a definition or `sch` for a scheme.

Considering the redefinition of a relation's domain described in Section 2.1, we may see its rendering at `http://alioth.uwb.edu.pl/~artur/mmlesx/relset_1.html#relset_1_K1_P1` with a link to the synonymous notion being redefined at `http://alioth.uwb.`

edu.pl/~artur/mmlesx/relat_1.html#xtuple_0_K9_P1. The internal XML-based HTML representation at http://mizar.uwb.edu.pl/version/current/html/relset_1.html#K1 lacks this syntactic information and offers only a link to the original constructor (http://mizar.uwb.edu.pl/version/current/html/xtuple_0.html#K9).

The system of links can be further extended e.g. with references to the origin of constants, quantified variables, scheme variables etc. Please note that although the aim of the project was to reproduce the text of the Mizar articles as faithfully as possible, the differences in the use of spaces, line breaks or brackets are currently disregarded. Accurate reproduction of these aspects would require more purely textual information generated by the Mizar parser to be recorded and stored in `.wsx` files.

4.2 Formalized Mathematics

Formalized Mathematics (FM) publishes papers based on regular Mizar formalizations accepted for inclusion into the Mizar Mathematical Library following a round of human peer-review. After acceptance, the underlying Mizar scripts are automatically translated into a LaTeX format [2] and the resulting generated natural language (English) texts become available as traditional mathematical papers downloadable as PDF files. The current implementation the of software responsible for the translation is based on a number of `XSLT` style-sheets which convert `.wsx` files representing parse-trees of given Mizar articles into a series of XML files containing human readable meta-text with increasing level of semantic detail [6]. As the original Mizar scripts are plain text files encoded with standard ASCII, traditional mathematical symbols like $\cup$, Σ, or $\int$ cannot be directly used in Mizar texts. To render them in the LaTeX documents, authors of Mizar formalizations can propose their preferred *translation patterns* to FM editors. These patterns allow changing formal and often technical-looking Mizar statements into more natural representation resembling standard mathematical notation using traditional symbols or a fixed placement and order of arguments. Sometimes these patterns become more complex, e.g. in the case of matrices, where the translation needs a special language construct rather than a simple symbolic replacement.

The current FM translation method based primarily on the `.wsx` files has the downside that some different notions introduced in the Mizar script are indistinguishable without special processing by more advanced modules of the Mizar verifier. For example, the multiplication of complex numbers and the multiplication of elements of a ring, when they both are written as infix operations utilizing the same symbol. Such shortcomings are overcome by the richer information present in the new proposed ESM format. The corresponding new `.esx` representations look like this:

Listing 8

```
<Functor-Definition MMLId="E1:1">
 <InfixFunctor-Pattern formatdes="O43[1(2)1]" formatnr="80" spelling="*"
    position="24\9" patternnr="224" absolutepatternMMLId="E1:2" constr="K465"
    absoluteconstrMMLId="E1:1" origconstrnr="0">
```

Listing 9

```
<Functor-Definition MMLId="E1:2">
 <InfixFunctor-Pattern formatdes="O43[1(2)1]" formatnr="80" spelling="*"
    position="33\9" patternnr="225" absolutepatternMMLId="E1:3" constr="K466"
    absoluteconstrMMLId="E1:2" origconstrnr="0">
```

We can observe that both operations have the same `formatnr="80"`, but they can now be uniquely identified by values of the `patternnr` and `constr` attributes. Consequently, the future FM representations of Mizar articles can be improved accordingly once the translation is based on patterns rather than formats.

5 Conclusions

In this paper we announced the existence of an extended XML-based data format simplifying the access to mathematical notions formalized in Mizar and available as part of the Mizar Mathematical Library, called Even more Strict Mizar (ESM). It is designed to combine and provide an easy access to both syntactic and semantic data of the underlying Mizar scripts. The extra information should allow creating various applications of the Mizar library requiring fullest possible information to be retrieved from the formalization files, especially using external general-purpose XML processing libraries (e.g. `dom4j`[5] or RapidXml[6]). Additionally, the work on the new language helped to analyze and improve the structure of already existing Mizar XML file formats (WSM and MSM).

References

1 Jesse Alama, Tom Heskes, Daniel Kühlwein, Evgeni Tsivtsivadze, and Josef Urban. Premise selection for mathematics by corpus analysis and kernel methods. *J. Autom. Reasoning*, 52(2):191–213, 2014. doi:10.1007/s10817-013-9286-5.

2 Grzegorz Bancerek. Automatic translation in Formalized Mathematics. *Mechanized Mathematics and Its Applications*, 5(2):19–31, December 2006.

3 Grzegorz Bancerek. Information retrieval and rendering with MML query. In Jonathan Borwein and William Farmer, editors, *Mathematical Knowledge Management*, volume 4108 of *Lecture Notes in Computer Science*, pages 266–279. Springer Berlin Heidelberg, 2006. doi:10.1007/11812289_21.

4 Grzegorz Bancerek, Czesław Byliński, Adam Grabowski, Artur Korniłowicz, Roman Matuszewski, Adam Naumowicz, and Karol Pąk. The role of the Mizar Mathematical Library for interactive proof development in Mizar. *Journal of Automated Reasoning*, 61(1):9–32, June 2018. doi:10.1007/s10817-017-9440-6.

5 Grzegorz Bancerek, Czesław Byliński, Adam Grabowski, Artur Korniłowicz, Roman Matuszewski, Adam Naumowicz, Karol Pąk, and Josef Urban. Mizar: State-of-the-art and beyond. In Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge, editors, *Intelligent Computer Mathematics – International Conference, CICM 2015, Washington, DC, USA, July 13–17, 2015, Proceedings*, volume 9150 of *Lecture Notes in Computer Science*, pages 261–279. Springer, 2015. doi:10.1007/978-3-319-20615-8_17.

6 Grzegorz Bancerek, Adam Naumowicz, and Josef Urban. System description: XSL-based translator of Mizar to LaTeX. In Florian Rabe, William M. Farmer, Grant O. Passmore, and Abdou Youssef, editors, *Intelligent Computer Mathematics – 11th International Conference, CICM 2018, Hagenberg, Austria, August 13–17, 2018, Proceedings*, volume 11006 of *Lecture Notes in Computer Science*, pages 1–6. Springer, 2018. doi:10.1007/978-3-319-96812-4_1.

7 Czesław Byliński and Jesse Alama. New developments in parsing Mizar. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors, *Intelligent Computer Mathematics 11th International Conference, AISC 2012, 19th Symposium, Calculemus 2012, 5th International Workshop, DML 2012, 11th*

[5] https://dom4j.github.io/
[6] http://rapidxml.sourceforge.net/

International Conference, MKM 2012, Systems and Projects, volume 7362 of *Lecture Notes in Artificial Intelligence*, pages 427–431. Springer-Verlag, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-31374-5_30.

8 Ingo Dahn. Interpretation of a Mizar-like logic in first-order logic. In Ricardo Caferra and Gernot Salzer, editors, *FTP (LNCS Selection)*, volume 1761 of *Lecture Notes in Computer Science*, pages 137–151. Springer, 1998. doi:10.1007/3-540-46508-1_9.

9 Mihnea Iancu, Michael Kohlhase, Florian Rabe, and Josef Urban. The Mizar Mathematical Library in OMDoc: Translation and applications. *Journal of Automated Reasoning*, 50(2):191–202, February 2013. doi:10.1007/s10817-012-9271-4.

10 Cezary Kaliszyk and Karol Pąk. Isabelle import infrastructure for the Mizar Mathematical Library. In Florian Rabe, William M. Farmer, Grant O. Passmore, and Abdou Youssef, editors, *Intelligent Computer Mathematics – 11th International Conference, CICM 2018, Hagenberg, Austria, August 13–17, 2018, Proceedings*, volume 11006 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2018. doi:10.1007/978-3-319-96812-4_13.

11 Cezary Kaliszyk and Josef Urban. MizAR 40 for Mizar 40. *CoRR*, abs/1310.2805, 2013. URL: http://arxiv.org/abs/1310.2805.

12 Roman Matuszewski and Piotr Rudnicki. Mizar: The first 30 years. *Mechanized Mathematics and Its Applications, Special Issue on 30 Years of Mizar*, 4(1):3–24, March 2005.

13 Kazuhisa Nakasho. Development of a flexible Mizar tokenizer and parser for information retrieval system. In Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki, editors, *Proceedings of the 2019 Federated Conference on Computer Science and Information Systems, FedCSIS 2019, Leipzig, Germany, September 1–4, 2019*, volume 18 of *Annals of Computer Science and Information Systems*, pages 77–80, 2019. doi:10.15439/2019F151.

14 Adam Naumowicz and Radosław Piliszek. Accessing the Mizar library with a weakly strict Mizar parser. In Michael Kohlhase, Moa Johansson, Bruce R. Miller, Leonardo de Moura, and Frank Wm. Tompa, editors, *Intelligent Computer Mathematics – 9th International Conference, CICM 2016, Bialystok, Poland, July 25–29, 2016, Proceedings*, volume 9791 of *Lecture Notes in Computer Science*, pages 77–82. Springer, 2016. doi:10.1007/978-3-319-42547-4_6.

15 Karol Pąk. Combining the Syntactic and Semantic Representations of Mizar Proofs. In Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki, editors, *Proceedings of the 2018 Federated Conference on Computer Science and Information Systems, FedCSIS 2018, Poznan, Poland, September 9–12, 2018*, volume 15 of *Annals of Computer Science and Information Systems*, pages 145–153. IEEE, 2018. doi:10.15439/2018F248.

16 J. Urban, G. Sutcliffe, S. Trac, and Y. Puzis. Combining Mizar and TPTP Semantic Presentation and Verification Tools. *Studies in Logic, Grammar and Rhetoric*, 18(31):121–136, 2009.

17 Josef Urban. XML-izing Mizar: Making semantic processing and presentation of MML easy. In Michael Kohlhase, editor, *Mathematical Knowledge Management, 4th International Conference, MKM 2005, Bremen, Germany, July 15–17, 2005, Revised Selected Papers*, volume 3863 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 2005. doi:10.1007/11618027_23.

18 Josef Urban. MizarMode – an integrated proof assistance tool for the Mizar way of formalizing mathematics. *Journal of Applied Logic*, 4(4):414–427, 2006.

19 Josef Urban. MoMM – fast interreduction and retrieval in large libraries of formalized mathematics. *Int. J. on Artificial Intelligence Tools*, 15(1):109–130, 2006. URL: http://ktiml.mff.cuni.cz/~urban/MoMM/momm.ps.

20 Josef Urban, Piotr Rudnicki, and Geoff Sutcliffe. ATP and presentation service for Mizar formalizations. *Journal of Automated Reasoning*, 50(2):229–241, February 2013. doi:10.1007/s10817-012-9269-y.

Homotopy Type Theory in Isabelle

Joshua Chen
University of Nottingham, UK

Abstract

This paper introduces Isabelle/HoTT, the first development of homotopy type theory in the Isabelle proof assistant. Building on earlier work by Paulson, I use Isabelle's existing logical framework infrastructure to implement essential automation, such as type checking and term elaboration, that is usually handled on the source code level of dependently typed systems. I also integrate the propositions-as-types paradigm with the declarative Isar proof language, providing an alternative to the tactic-based proofs of Coq and the proof terms of Agda. The infrastructure developed is then used to formalize foundational results from the Homotopy Type Theory book.

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

Keywords and phrases Proof assistants, Logical frameworks, Dependent type theory, Homotopy type theory

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.12

Supplementary Material *Software (Source)*: `https://github.com/jaycech3n/Isabelle-HoTT/tree/ITP2021`; archived at `swh:1:dir:8c8943216a189d6b9144942ee0d37ab75b52e5fc`

Funding Partially funded by ERC project SMART (starting grant no. 714034) and the Royal Society (grant URF\R1\191055).

1 Introduction

Isabelle [15] is a simply typed proof assistant and logical framework. Of its multiple object logics Isabelle/HOL [12] is arguably the best known, however many other logics have been created since Isabelle's inception and are still bundled along with its distribution. Among these early logics is Isabelle/CTT (*constructive type theory*) [13], which is based on extensional Martin-Löf type theory but which has not been further developed. In light of considerable recent progress in the field of dependent type theory, it seems appropriate to revive support for this in Isabelle. This paper aims to do this by introducing *Isabelle/HoTT*, the first development of homotopy type theory in Isabelle.

Widely accepted folklore in the theorem proving community holds that a sufficiently strong logical framework can in principle be used to encode and work in any foundational theory of equal or lesser strength. However, a drawback to this approach is that one then has to implement the foundation-specific infrastructure on one's own, while working within the additional constraints imposed by the framework. This becomes particularly clear when the formalism of the framework logic is sufficiently different from that of the object logic, as in our current case.

The Isabelle/HoTT project may thus be viewed in three distinct but related ways:

- As the beginnings of an Isabelle formalization of homotopy type theory.
- As a dependently typed Isabelle object logic which improves on Isabelle/CTT with necessary supporting infrastructure for type checking, term elaboration, proof term abstraction and tactics.
- As a practical case study on the implementation issues discussed in the previous paragraph.

This is a short paper on ongoing work. Isabelle/HoTT currently lacks automation for function definitions, datatypes, and advanced features like higher inductive types (although these may be manually defined or postulated). Despite this, it is already able to formalize nontrivial results from the Homotopy Type Theory book [14]. In addition, although the logic presented here is formulated in the axiomatic style of the HoTT book, one could use the same approach to develop two-level type theory [1, 3] and cubical type theory [2, 8] in Isabelle.

Isabelle/HoTT is implemented as a library of Standard ML and Isabelle theory files. References to specific files are given as footnotes throughout this paper, and the source code is available online at `https://github.com/jaycech3n/Isabelle-HoTT/tree/ITP2021`.

Related Work

One of the earliest object logics for Isabelle was Paulson's Isabelle/CTT [13] for constructive type theory with extensional equality. Indeed, the fundamental ideas of using resolution to perform type checking and inference, and of discharging subgoals in order of increasing flexibility, already appear here. Isabelle/HoTT improves on this work by implementing universes, an intensional equality type, as well as better integration of type inference and implicit elaboration into the proof process.

Another recent study in developing homotopy type theory in a logical framework appears in work by Barras and Maestracci [5], where they present a partial embedding of de Morgan cubical type theory [8] using rewrite rules in the $\lambda\Pi$-calculus modulo logic of Dedukti [4]. Our encoding of axiomatic HoTT in simple type theory is more straightforward, allowing us to focus instead on issues arising from integrating the simply and dependently typed paradigms of the meta and object logics.

The largest computer developments of homotopy type theory are well known and use the Coq and Agda proof assistants [6, 7]. In these settings the theory is developed synthetically, and in the case of Coq the source code was directly modified in order to implement new features required by the theory. In our case the trusted prover code is untouched, and we simply extend Isabelle/Pure with new features using its existing logical framework facilities.

2 Logical Foundations

Judgments. We begin[1], as usual, by declaring a meta type o of terms of the object logic, and a constructor `has_type` $:: o \to o \to prop$ (written as usual with an infix colon) to encode the typing assertion. Since we implement Russell-style universes, types are themselves terms and must have the same meta type, and in this way we will effectively have a set of untyped terms in higher order abstract syntax. Working with Tarski-style universes would allow us to maintain the syntactic type/term distinction with separate meta types, at the cost of having to introduce interpretation operators everywhere.

Judgmental equality of the type theory is shallowly embedded using the Isabelle/Pure equality $\equiv$. This forgets type information, but allows us to more easily reuse the simplifier to compute terms.

Universes. We postulate a set of levels isomorphic to the standard natural numbers with their usual order, by declaring a meta type lvl and constants `0`, `S` and $<$. Universes are formed by a constructor `U` $:: lvl \to o$, and we axiomatize rules governing the ordering of levels, as well as the hierarchy and cumulativity of universes.

[1] `mltt/core/MLTT.thy`

Types and Terms. The constants for formers, constructors and eliminators for the Π, Σ and identity types are postulated using Church-style typing. Type families as well as function arguments to dependent eliminators are encoded using meta instead of object lambda terms. For example, in theoretical presentations the Σ-eliminator might be given by a term $\mathtt{SigInd}(A, B, C, f)$ whose third and fourth arguments are, respectively, a type family $C\colon (\Sigma\, A\, B) \to U$ and a function $f\colon \Pi(x\colon A)(y\colon B(x)).\, C(x, y)$ defining the value of C on all pairs $(a, b)\colon \Sigma\, A\, B$. In the encoding, these are instead given as the simply typed meta functions $C :: o \to o$ and $f :: o \to o \to o$. However, after the Π-type has been encoded, Isabelle's implicit coercion mechanism (Section 12.3 of [17]) is used to coerce object functions into meta functions, which allows users to ignore this distinction most of the time.

Inference Rules. Following Jacobs and Melham [11], we define an encoding ε from the judgments of dependent type theory into Isabelle/Pure by sending $x_1\colon A_1, \ldots, x_n\colon A_n \vdash \mathcal{I}$ to the universally-quantified implication

$$\bigwedge x_1, \ldots, x_n.\; x_1\colon A_1 \Longrightarrow \cdots \Longrightarrow x_n\colon A_n \Longrightarrow \varepsilon(\mathcal{I}),$$

where $\varepsilon(t\colon T) := t\colon T$ and $\varepsilon(a \equiv b\colon T) := a \equiv b$ are the encodings of typing and equality discussed above. This encoding is recursively extended to inference rules by defining

$$\varepsilon\left(\frac{\mathcal{J}_1 \quad \cdots \quad \mathcal{J}_k}{\mathcal{J}}\right) := \big(\varepsilon(\mathcal{J}_1) \Longrightarrow \cdots \Longrightarrow \varepsilon(\mathcal{J}_k) \Longrightarrow \varepsilon(\mathcal{J})\big).$$

Note that entailment and derivability are both translated to Pure implication. The usual rules for formation, introduction, elimination, computation and congruence of Π, Σ and equality types (see e.g. [14]) can then be axiomatized.

More generally, a statement

$$\bigwedge \vec{x}.\; P_1(\vec{x}) \Longrightarrow \cdots \Longrightarrow P_k(\vec{x}) \Longrightarrow Q(\vec{x}) \tag{1}$$

in Isabelle/HoTT may be viewed as an extended form of type-theoretic judgment

$$\Gamma \vdash t\colon T \quad \text{or} \quad \Gamma \vdash t \equiv s\colon T \text{ (where } T \text{ is implicit)},$$

where contexts $\Gamma = \big(P_1(\vec{x}), \ldots, P_k(\vec{x})\big)$ are also allowed to contain equality judgments, and where the metavariables $\vec{x} = \{x_1, \ldots, x_m\}$ may appear throughout. In particular, the x_i need not be explicitly typed by the context Γ. Such occurrences typically appear as unification variables in type checking and elaboration problems.

3 Proof Infrastructure

Implicits and Elaboration. Implicit arguments and term elaboration are crucial to working in a dependently typed system. We declare constants `?` and `{}` representing, respectively, holes and implicit arguments, together with a theorem attribute `implicit` and an Isabelle syntax phase operation `make_holes`.[2] We can then use the usual definitional facilities together with the `implicit` attribute in the usual manner, e.g.

```
definition Id_i (infix "=" 110) where [implicit]: "x = y ≡ x ={} y"
```

[2] `mltt/core/implicits.ML`

where `x =A y` is the fully explicit notation for the equality type. The implicits `{}` in such definitions will be parsed by `make_holes` into holes `?`, which are then further converted into schematic variables (i.e. Isabelle/Pure metavariables) in goal statements.

The implementation of implicit arguments as schematic variables means that a general goal statement in Isabelle/HoTT is schematic. Such goals are not very well supported by the existing Isar commands, so we define new goal keywords `Lemma`, `Theorem`, etc. (replacing `lemma`, `theorem` etc.)[3] as well as a command `assuming` (replacing `assume`).[4] These call the type checker on assumptions to infer their implicit arguments and thus instantiate all metavariables before passing them to the regular context assumption mechanism.[5]

Proof Terms. Consider the task of automatically abstracting proof terms into definitions. A theorem stated in a dependently typed system is given by a single type à la Curry-Howard. In contrast, in the LCF-style setting of Isabelle the assumptions of a theorem statement are typically available as facts in an Isabelle/Isar proof context, which are lifted to premises after the conclusion has been proved. In particular, these premises are *not* bound by the type of the theorem's conclusion. This distinction is exactly the isomorphism – given by the Π-introduction rule – between open terms with variables typed by a nonempty (type-theoretic) context, and closed terms of a Π-type (i.e. lambda terms).

Hence, in Isabelle, the proof term in a theorem's conclusion must be abstracted over all variables typed by the premises, in order to form a meta lambda term. This is then wrapped up into a definition. This functionality is available as a modifier `(def)` to the goal statement keywords discussed previously.

Induction/Elimination Rules. In dependent type theory, given a predicate $C\colon A \to U$, the elimination rule for A is used to prove $C(a)$ for all $a\colon A$. Crucially, this requires that C encodes all the assumptions needed to prove its conclusion. As previously noted, in Isabelle these assumptions may instead appear out in the Isar context, and thus in order to be able to apply elimination rules correctly we must ensure that such "free-floating" assumptions are pushed into the type of the goal.

Concretely, this involves checking the conclusion Q of a goal (1) for variables that are typed by premises or Isar context facts P_i, and then using Π-formation to push these assumptions into the object logic predicate C. Furthermore, since the Isar context is unordered and there may be typing dependencies among these assumptions, we first topologically sort them by $\lesssim$, where $t_i\colon T_i \lesssim t_j\colon T_j$ if t_i is a subterm of T_j. This process is automated by infrastructure introducing the `elim` attribute and proof method.[6]

Propositional Equality and Calculational Reasoning. The identity type $x =_A y$ is presented as an inductive family over its endpoints x, y. Its induction principle is subject to the same general considerations for elimination rules discussed above, and the proof method `eq` for reasoning with path induction is essentially a special case of the `elim` method.

Rewriting (aka *transport*) along propositional equalities is given by a method `rewr`.[7] We additionally adapt Isar's calculational reasoning (Sections 1.2 and 2.2.4 of [17]) to so-called *calculational* types, which are types $T\colon A \to A \to U$ that have a notion of composition

[3] `mltt/core/goals.ML`
[4] `mltt/core/elaborated_statement.ML`
[5] `mltt/core/elaboration.ML`
[6] `mltt/core/elimination.ML`, `mltt/core/tactics.ML`
[7] `hott/Identity.thy`

$\diamond\colon \Pi\{x, y, z\colon A\}.\ T(x, y) \to T(y, z) \to T(x, z)$ expressing transitivity of T. After declaring a calculational type and its transitivity rule with the `calc` keyword[8] and `trans` attribute we can then use the familiar idiom "`have...also have...finally show`" to construct transitive chains in proofs. By design, this technology is also general enough to support reasoning with chains of homotopies $f \sim g$.

4 Type Checking

The type checker[9] is a key component integrated throughout the infrastructure described above. It is used by goal commands to perform implicit elaboration, hooked in to proof methods to automatically discharge ancillary typing conditions that arise throughout the course of a proof, and installed as an Isabelle simp-solver[10] to enable typed term reduction. It is also available as a standalone method `typechk`.

At its core is a tactic that recursively resolves goals against the type inference (i.e. formation, introduction and elimination) rules, suitable facts from the local Isar context, any additional rules declared with the `type` attribute, and the conversion rule. It is restricted to judgments $t\colon T$ where t is rigid (i.e. where the head of t is a constant) and T may be schematic. Combined with unification this yields a bidirectional type checking/inference algorithm, which is syntax-directed on the collection of type inference rules since every rule in this collection types a term with unique head. Nondeterminism is introduced when resolving against context facts and rules from the user-modifiable `type` theorem collection, and here backtracking allows the checker to try all possible options. If it fails to completely solve an inference problem, the type checker will return the goals on which it failed to the user for further refinement.

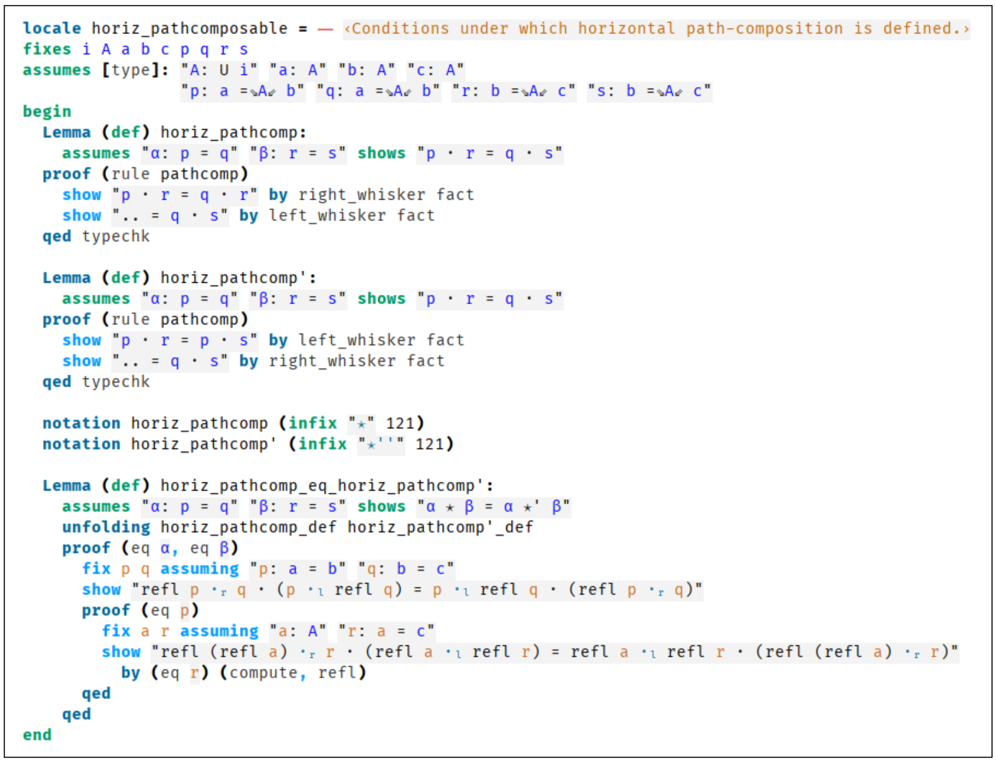
The conversion rule $\bigwedge a\, A\, A'.\ a\colon A \Longrightarrow A \equiv A' \Longrightarrow a\colon A'$ introduces normalization into the type checker; the simplifier is used to solve the second proof obligation. This is currently somewhat rudimentary since definitional unfolding is not yet implemented, but this is expected to be relatively straightforward to add.

As already noted by Paulson, the order in which subgoals are tackled in a type inference problem matters greatly, as the large number of metavariables – especially with implicit arguments – creates potentially many unification candidates and too large a search space if not resolved against the correct rule. He mitigates this by using a filter-and-repeat technique to attempt the subgoals with the fewest metavariables first; we achieve a similar effect by carefully ordering the premises of inference rules according to the criteria for bidirectional type systems set out by Dunfield and Krishnaswami [9].

5 Formalization

The object logic developed is used to formalize material from the first chapters of the Homotopy Type Theory book in Isabelle2020 [15], including results on equality, homotopies and equivalences, and more.[11] Figure 1 shows an example proof that the two ways one can define horizontal composition of equalities on a type A are equal[12], which is an intermediate result en route to the proof of the Eckmann-Hilton argument for $\Omega^2(A)$ (Theorem 2.1.6

[8] `mltt/core/calc.ML`
[9] `mltt/core/types.ML`
[10] An Isabelle simplifier component that solves subgoals arising from conditional simplification rules.
[11] `hott/*.thy`
[12] `hott/Identity.thy`.



```
locale horiz_pathcomposable = — ‹Conditions under which horizontal path-composition is defined.›
fixes i A a b c p q r s
assumes [type]: "A: U i" "a: A" "b: A" "c: A"
                "p: a =⇘A⇙ b" "q: a =⇘A⇙ b" "r: b =⇘A⇙ c" "s: b =⇘A⇙ c"
begin
  Lemma (def) horiz_pathcomp:
    assumes "α: p = q" "β: r = s" shows "p · r = q · s"
  proof (rule pathcomp)
    show "p · r = q · r" by right_whisker fact
    show ".. = q · s" by left_whisker fact
  qed typechk

  Lemma (def) horiz_pathcomp':
    assumes "α: p = q" "β: r = s" shows "p · r = q · s"
  proof (rule pathcomp)
    show "p · r = p · s" by left_whisker fact
    show ".. = q · s" by right_whisker fact
  qed typechk

  notation horiz_pathcomp (infix "⋆" 121)
  notation horiz_pathcomp' (infix "⋆''" 121)

  Lemma (def) horiz_pathcomp_eq_horiz_pathcomp':
    assumes "α: p = q" "β: r = s" shows "α ⋆ β = α ⋆' β"
    unfolding horiz_pathcomp_def horiz_pathcomp'_def
    proof (eq α, eq β)
      fix p q assuming "p: a = b" "q: b = c"
      show "refl p ·ᵣ q · (p ·ₗ refl q) = p ·ₗ refl q · (refl p ·ᵣ q)"
      proof (eq p)
        fix a r assuming "a: A" "r: a = c"
        show "refl (refl a) ·ᵣ r · (refl a ·ₗ refl r) = refl a ·ₗ refl r · (refl (refl a) ·ᵣ r)"
          by (eq r) (compute, refl)
      qed
    qed
end
```

Figure 1 Example: Horizontal composition.

of [14], also formalized in this work). This example demonstrates all the functionality described above in action: implicit elaboration of assumptions throughout all goal and proof statements, automatic definitions for the terms `horiz_pathcomp` and `horiz_pathcomp'` from their constructions via proofs, as well as path induction and calculational reasoning on equalities in the proof of the final lemma.

6 Discussion and Future Work

Isabelle/HoTT and its accompanying formalization show that Isabelle's simply typed logical framework infrastructure is feasibly able to provide strong support for modern-day developments of dependent type theory. However, many improvements are still possible, and future work aims to implement inductive and higher inductive types, as well as to explore how the techniques presented in this paper may be used to implement cubical type theory [2, 8] and two-level type theory [3, 16].

It would be productive to attempt to formalize the notion of a semisimplicial type [10] in Isabelle/HoTT. Internalizing the full definition of such an object in homotopy type theory is a well known open problem, with the current state-of-the-art requiring a two-level type theory in order to have a strict equality and natural number type on the outer level [1]. In principle, Isabelle's logical framework can easily provide these. The main hurdles would again be in implementing enough features on the object logic level, for example to support mutually inductive datatypes. In this way, the goal of formalizing semisimplicial types could provide further impetus to the development of homotopy type theory in Isabelle.

References

1 Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. Extending Homotopy Type Theory with Strict Equality. In Jean-Marc Talbot and Laurent Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*, volume 62 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:17, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.CSL.2016.21`.

2 Carlo Angiuli, Kuen-Bang Hou (Favonia), and Robert Harper. Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities. In Dan Ghica and Achim Jung, editors, *27th EACSL Annual Conference on Computer Science Logic (CSL 2018)*, volume 119 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:17, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.CSL.2018.6`.

3 Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. Two-level type theory and applications, 2019. `arXiv:1705.03307`.

4 Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Dedukti: a logical framework based on the $\lambda\pi$-calculus modulo theory, 2016. URL: `http://www.lsv.fr/~dowek/Publi/expressing.pdf`.

5 Bruno Barras and Valentin Maestracci. Implementation of two layers type theory in Dedukti and application to cubical type theory. *Electronic Proceedings in Theoretical Computer Science*, 332:54–67, January 2021. `doi:10.4204/eptcs.332.4`.

6 Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. The HoTT library: A formalization of homotopy type theory in Coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, page 164–172, New York, NY, USA, 2017. Association for Computing Machinery. `doi:10.1145/3018610.3018615`.

7 Guillaume Brunerie, Kuen-Bang Hou (Favonia), Evan Cavallo, Tim Baumann, Eric Finster, Jesper Cockx, Christian Sattler, Chris Jeris, Michael Shulman, et al. Homotopy type theory in Agda, 2021. URL: `https://github.com/HoTT/HoTT-Agda`.

8 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:34, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.TYPES.2015.5`.

9 Jana Dunfield and Neel Krishnaswami. Bidirectional typing, 2020. `arXiv:1908.05839`.

10 Hugo Herbelin. A dependently-typed construction of semi-simplicial types. *Mathematical Structures in Computer Science*, 25(5):1116–1131, 2015. `doi:10.1017/S0960129514000528`.

11 Bart Jacobs and Thomas F. Melham. Translating dependent type theory into higher order logic. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, TLCA '93, page 209–229, Berlin, Heidelberg, 1993. Springer-Verlag.

12 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, April 2020. URL: `https://isabelle.in.tum.de/website-Isabelle2020/dist/Isabelle2020/doc/tutorial.pdf`.

13 Lawrence C. Paulson. *Constructive Type Theory*, April 2020. URL: `https://isabelle.in.tum.de/website-Isabelle2020/dist/Isabelle2020/doc/logics.pdf`.

14 The Univalent Foundations Program and Institute for Advanced Study. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, 1st edition, 2013.

15 University of Cambridge, Technische Universität München, and Contributors. Isabelle2020, April 2020. URL: `https://isabelle.in.tum.de/website-Isabelle2020`.

16 Vladimir Voevodsky. A simple type system with two identity types, February 2013. Unpublished note, available online at `https://www.math.ias.edu/vladimir/Lectures`. URL: `https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/HTS.pdf`.

17 Makarius Wenzel. *The Isabelle/Isar Reference Manual*, April 2020. URL: `https://isabelle.in.tum.de/website-Isabelle2020/dist/Isabelle2020/doc/isar-ref.pdf`.

Flexible Coinduction in Agda

Luca Ciccone
University of Torino, Italy

Francesco Dagnino
DIBRIS, University of Genova, Italy

Elena Zucca
DIBRIS, University of Genova, Italy

Abstract

We provide an Agda library for *inference systems*, also supporting their recent generalization allowing *flexible coinduction*, that is, interpretations which are neither inductive, nor purely coinductive. A specific inference system can be obtained as an instance by writing a set of meta-rules, in an Agda format which closely resembles the usual one. In this way, the user gets for free the related properties, notably the inductive and coinductive intepretation and the corresponding proof principles. Moreover, a significant modularity is achieved. Indeed, rather than being defined from scratch and with a built-in interpretation, an inference system can also be obtained by composition operators, such as union and restriction to a smaller universe, and its semantics can be modularly chosen as well. In particular, flexible coinduction is obtained by composing in a certain way the interpretations of two inference systems. We illustrate the use of the library by several examples. The most significant one is a big-step semantics for the λ-calculus, where flexible coinduction allows to obtain a special result (∞) for all and only the diverging computations, and the proof of equivalence with small-step semantics is carried out by relying on the proof principles offered by the library.

2012 ACM Subject Classification Theory of computation → Semantics and reasoning

Keywords and phrases inference systems, induction, coinduction

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.13

Supplementary Material The complete code of the library with all the examples discussed in the paper is available at
Software (Source Code): `https://github.com/LcicC/inference-systems-agda`
archived at `swh:1:dir:955d11416c46a1ee1b0b913f0bc5f30bf3de9e97`

Acknowledgements We warmly thank the anonymous ITP reviewers and Andreas Abel for the suggestions which greatly improved the paper, in particular for pointing out the relation with indexed containers.

1 Introduction

An *inference system* [5, 19, 21], that is, a set of (meta-)rules stating that a consequence can be derived from a set of premises, is a simple, general and widely-used way to express and reason about a recursive definition. In most cases such recursive definition is seen as inductive, that is, the denoted set consists of the elements with a finite derivation. This enables *inductive reasoning*, that is, to prove that the elements an inductively defined set satisfy a property, it is enough to show that, for each (meta-)rule, the property holds for the consequence assuming that it holds for the premises. In other cases, the recursive definition is seen as coinductive, that is, the denoted set consists of the elements with a possibly infinite derivation. This enables *coinductive reasoning*, that is, to prove that all the elements satisfying a property belong to the coinductively defined set, it is enough to show that, when the property holds for an element, it can be derived from premises for which the property holds as well. Recently, a generalization of inference systems has been proposed [8, 13, 15] which handles cases where neither the inductive, nor the purely coinductive intepretation provides the desired meaning.

This approach is called *flexible coinduction*, and, correspondingly, coinductive reasoning is generalized as well by a principle which is called *bounded coinduction*.

The Agda proof assistant [23] offers language constructs to inductively/coinductively define predicates, and correspondingly built-in proof principles. However, in this way the recursive definition is monolithic, and hard-wired with its chosen interpretation. Our aim, instead, is to provide an Agda library allowing the user to express a recursive definition as an *instance of a parametric type* of inference systems. In this way, the user is not committed from the beginning to a given interpretation but, rather, gets for free a bunch of properties which have been proved once and for all, including the inductive and coinductive intepretation and the corresponding proof principles. Moreover, it is possible to define composition operators on inference systems, for instance union and restriction. Finally, flexible coinduction is modularly obtained as well, by composing in a certain way the interpretations of two inference systems.

Indexed containers [6] provide a way to specify possibly recursive definitions of predicates independently from their interpretation and are supported in the Agda standard library. An Agda implementation of inference systems can be provided by seeing them as indexed containers. However, this approach requires to structure definitions in an unusual way. Indeed, inference systems are usually presented through a (finite) set of *meta-rules*, denoting all the rules which can be obtained by instantiating meta-variables with values satisfying the side condition. Hence, we provide a different implementation following this schema, to allow users to write their own inference system in an Agda format which closely resembles that "on paper". We then prove that the two implementations are equivalent, showing that every indexed container can be encoded in terms of meta-rules and viceversa.

In Sect. 2 we recall basic notions on inference systems, and in Sect. 3 the generalization supporting flexible coinduction. In Sect. 4 we describe how to implement (generalized) inference systems in Agda. Notably, in Sect. 4.1 we present the approach mimicking meta-rules, showing step-by-step the correspondence with the previous definitions. In Sect. 4.2, instead, we explain the view of inference systems as indexed containers, and prove the equivalence. Then, we illustrate the use of the library by several examples. In Sect. 5 we consider three different predicates on possibily infinite lists (*colists* in Agda terminology) defined by induction, coinduction, and flexible coinduction, respectively. In Sect. 6 we provide a more significant and elaborated example: a big-step semantics for the λ-calculus where flexible coinduction allows to obtain a special result (∞) for all and only the diverging computations, and the proof of equivalence with small-step semantics is carried out by relying on the proof principles offered by the library. Finally, we summarize the contribution and outline further work in Sect. 7.

2 Inference systems

We recall basic definitions on inference systems [5, 19, 21, 15]. Throughout this section and the following we assume a set $\mathcal{U}$, named *universe*, whose elements j are called *judgments*. An *inference system* $\mathcal{I}$ is a set of *rules*, which are pairs $\langle pr, j \rangle$, with $pr \subseteq \mathcal{U}$ the set of *premises*, and $j \in \mathcal{U}$ the *conclusion* (a.k.a. *consequence*). A rule with an empty set of premises is an *axiom*. A rule $\langle pr, j \rangle$ is often written as a fraction $\dfrac{pr}{j}$.

In practice, inference systems are described by a (finite) set of *meta-rules*, written in some meta-language. For instance, taking as universe the set $\mathbb{N}^\infty$ of finite and infinite lists of natural numbers, and denoting Λ the empty list, and $x{:}u$ the list with head x and tail u, the following two sets of meta-rules describe inference systems for the predicates holding when an element belongs to the list, and all elements are positive, respectively.

$$\text{(mem-h)}\ \frac{}{\mathsf{member}(x, x{:}xs)} \qquad \text{(mem-t)}\ \frac{\mathsf{member}(x, xs)}{\mathsf{member}(x, y{:}xs)}$$

$$\text{(allP-}\Lambda\text{)}\ \frac{}{\mathsf{allPos}(\Lambda)} \qquad \text{(allP-t)}\ \frac{\mathsf{allPos}(xs)}{\mathsf{allPos}(x{:}xs)}\ \ x > 0$$

The aim of an inference system is to define, in a way which provides canonical techniques to prove properties, a subset of the universe. There are several ways to choose this set, depending on the *interpretation* given to the inference system.

To define an interpretation in a model-theoretic way, the basis is the *inference operator* associated with $\mathcal{I}$, which is the function $F_\mathcal{I} : \wp(\mathcal{U}) \to \wp(\mathcal{U})$ defined by

$$F_\mathcal{I}(X) = \{j \in \mathcal{U} \mid pr \subseteq X, \langle pr, j\rangle \in \mathcal{I},\ \text{for some } pr \in \wp(\mathcal{U})\}.$$

A subset X of the universe is $\mathcal{I}$*-closed* if, for all rules $\langle pr, j\rangle \in \mathcal{I}$, if $pr \subseteq X$ then $j \in X$, it is $\mathcal{I}$*-consistent* if, for all $j \in X$, there is a rule $\langle pr, j\rangle \in \mathcal{I}$ and $pr \subseteq X$.

The *inductive interpretation* $Ind[\![\mathcal{I}]\!]$ is the least fixed point of $F_\mathcal{I}$, which, by the Knaster-Tarski theorem, coincides with the least pre-fixed point of $F_\mathcal{I}$ and so with the least $\mathcal{I}$-closed set. As an immediate consequence, when we define a set inductively, that is, as $Ind[\![\mathcal{I}]\!]$ for some $\mathcal{I}$, we can prove that such definition is *sound* with respect to a given specification, namely, a subset $\mathcal{S} \subseteq \mathcal{U}$, by the *induction principle*:

(ind) If a set $\mathcal{S} \subseteq \mathcal{U}$ is $\mathcal{I}$-closed, then $Ind[\![\mathcal{I}]\!] \subseteq \mathcal{S}$.

Proving that $\mathcal{S}$ is $\mathcal{I}$-closed amounts to show that, for each (meta-)rule, if the premises satisfy $\mathcal{S}$ then the consequence satisfies $\mathcal{S}$ as well.

The *coinductive interpretation* $CoInd[\![\mathcal{I}]\!]$ is the greatest fixed point of $F_\mathcal{I}$, which, by the Knaster-Tarski theorem, coincides with the largest post-fixed point of $F_\mathcal{I}$ and so with the largest $\mathcal{I}$-consistent set. As an immediate consequence, when we define a set coinductively, that is, as $CoInd[\![\mathcal{I}]\!]$ for some $\mathcal{I}$, we can prove that such definition is *complete* with respect to a given specification $\mathcal{S} \subseteq \mathcal{U}$ by the *coinduction principle*:

(coind) If a subset $\mathcal{S} \subseteq \mathcal{U}$ is $\mathcal{I}$-consistent, then $\mathcal{S} \subseteq CoInd[\![\mathcal{I}]\!]$.

Proving that $\mathcal{S}$ is $\mathcal{I}$-consistent amounts to show that, for each j satisfying $\mathcal{S}$, there is a rule with consequence j and premises satisfying $\mathcal{S}$ as well.

To prove completeness of the inductive interpretation, and soundness of the coinductive interpretation, instead, there is no canonical technique, so some ad-hoc proof is needed.

Alternatively, the interpretation can also be specified proof-theoretically, that is, through the notion of *proof tree*. For the aim of this paper a semi-formal definition is enough, we refer to [13, 15] for a rigorous treatment. Set $\mathcal{T}$ the set of trees with nodes (labeled by) judgments. Given $\tau \in \mathcal{T}$, $\mathsf{r}(\tau)$ is the (label of the) root, $\mathsf{dst}(\tau)$ the set of direct subtrees, and $\mathsf{chl}(\tau)$ the set of (the labels of) their roots. The inference operator can be naturally extended to a function $T_\mathcal{I} : \wp(\mathcal{T}) \to \wp(\mathcal{T})$ as follows:

$$T_\mathcal{I}(Y) = \{\tau \in \mathcal{T} \mid \mathsf{dst}(\tau) \subseteq Y, \langle \mathsf{chl}(\tau), \mathsf{r}(\tau)\rangle \in \mathcal{I}\}$$

Then, a *proof tree* (a.k.a. *derivation*) is a tree such that, for each subtree τ, $\langle \mathsf{chl}(\tau), \mathsf{r}(\tau)\rangle \in \mathcal{I}$, that is, there is a node (labelled by) j with set of children (labelled by) pr only if the rule $\langle pr, j\rangle$ belongs to $\mathcal{I}$. A *proof tree for* j is a proof tree τ such that $\mathsf{r}(\tau) = j$.

Then, $\mathit{Ind}[\![\mathcal{I}]\!]$ and $\mathit{CoInd}[\![\mathcal{I}]\!]$ can be equivalently defined as the sets of judgments with respectively a finite and a possibly infinite proof tree. Moreover, the sets of finite and possibly infinite proof trees turn out to be the least fixed point and the greatest fixed point, respectively, of the inference operator extended to trees. See [13, 14] for detailed proofs carried out with a rigorous definition of trees.

Coming back to the two examples above, it is easy to see that, in order to obtain the desired meaning, the inference system for member should be interpreted inductively, while that for allPos coinductively. Indeed, the fact that an element belongs to the list can be shown by a finite proof tree, even for an infinite list, whereas, for such a list, to show that all elements are positive an infinite proof tree is needed.

3 Corules

We recall the notion of inference systems with corules, which mixes induction and coinduction in a flexible way [8, 13, 15]. For $X \subseteq \mathcal{U}$, we write $\mathcal{I}_{|X}$ for the *restriction* of $\mathcal{I}$ to X, that is, the inference system $\{\langle pr, j\rangle \in \mathcal{I} \mid j \in X\}$.

▶ **Definition 1.** *A* generalized inference system*, or* inference system with corules*, is a pair $\langle \mathcal{I}, \mathcal{I}_{\mathsf{co}}\rangle$ where $\mathcal{I}$ and $\mathcal{I}_{\mathsf{co}}$ are inference systems. Elements in $\mathcal{I}$ and $\mathcal{I}_{\mathsf{co}}$ are called rules and corules, respectively. The interpretation of $\langle \mathcal{I}, \mathcal{I}_{\mathsf{co}}\rangle$ is defined by $\mathit{FCoInd}[\![\mathcal{I}, \mathcal{I}_{\mathsf{co}}]\!] = \mathit{CoInd}[\![\mathcal{I}_{|\mathit{Ind}[\![\mathcal{I}\cup\mathcal{I}_{\mathsf{co}}]\!]}]\!]$.*

Thus, the interpretation $\mathit{FCoInd}[\![\mathcal{I}, \mathcal{I}_{\mathsf{co}}]\!]$ is basically *coinductive*, but restricted to a universe of judgements which is *inductively defined* by the (potentially) larger system $\mathcal{I} \cup \mathcal{I}_{\mathsf{co}}$.

In [8, 13, 15] the following results are proved:

- $\mathit{FCoInd}[\![\mathcal{I}, \mathcal{I}_{\mathsf{co}}]\!]$ is the largest post-fixed point of $F_{\mathcal{I}}$ included in $\mathit{Ind}[\![\mathcal{I} \cup \mathcal{I}_{\mathsf{co}}]\!]$
- in proof-theoretic terms, $\mathit{FCoInd}[\![\mathcal{I}, \mathcal{I}_{\mathsf{co}}]\!]$ is the set of judgments which have a possibly infinite proof tree in $\mathcal{I}$ whose nodes all have a finite proof tree in $\mathcal{I} \cup \mathcal{I}_{\mathsf{co}}$, that is, the (standard) inference system consisting of rules and corules.

As an immediate consequence, when we define a set by flexible coinduction, that is, as $\mathit{FCoInd}[\![\mathcal{I}, \mathcal{I}_{\mathsf{co}}]\!]$ for some $\langle \mathcal{I}, \mathcal{I}_{\mathsf{co}}\rangle$, we can prove that such definition is *complete* with respect to a given specification $\mathcal{S} \subseteq \mathcal{U}$ by the *bounded coinduction principle*, which generalizes the coinduction principle:

(b-coind) If a subset $\mathcal{S} \subseteq \mathcal{U}$ is bounded, that is, $\mathcal{S} \subseteq \mathit{Ind}[\![\mathcal{I} \cup \mathcal{I}_{\mathsf{co}}]\!]$, and $\mathcal{I}$-consistent, then $\mathcal{S} \subseteq \mathit{FCoInd}[\![\mathcal{I}, \mathcal{I}_{\mathsf{co}}]\!]$.

Proving that $\mathcal{S}$ is bounded means proving completeness of the inference system extended by corules, interpreted inductively, with respect to $\mathcal{S}$. Hence, there is no canonical technique, and for each concrete case we must find an ad-hoc proof. Proving that $\mathcal{S}$ is $\mathcal{I}$-consistent, as for the standard coinduction principle, amounts to show that, for each j satisfying $\mathcal{S}$, there is a rule with consequence j and premises satisfying $\mathcal{S}$ as well. As for the purely coinductive interpretation, an ad-hoc proof is also needed for soundness. However, as shown in the examples in Sect. 5 and Sect. 6, in many cases we can take advantage of the fact that $\mathit{FCoInd}[\![\mathcal{I}, \mathcal{I}_{\mathsf{co}}]\!]$ is a subset of $\mathit{Ind}[\![\mathcal{I} \cup \mathcal{I}_{\mathsf{co}}]\!]$, and reason by induction on the latter.

We illustrate the role of corules by a simple example: defining the maximal element of a list. A (meta-)corule is written as a fraction with a thicker line.

$$\text{(max-}\Lambda\text{)}\ \frac{}{\mathsf{maxElem}(x{:}\Lambda, x)} \qquad \text{(max-t)}\ \frac{\mathsf{maxElem}(u, y)}{\mathsf{maxElem}(x{:}u, z)}\ z = \max(x, y) \qquad \text{(max-co)}\ \frac{}{\mathsf{maxElem}(x{:}u, x)}$$

Considering the standard inference system consisting of the two rules, its inductive interpretation only provides the desired meaning on finite lists, since for infinite lists an infinite proof is needed. However, the coinductive interpretation allows also wrong judgements. For instance, let $L = 1{:}2{:}1{:}2{:}1{:}2{:}\ldots$. Then any judgment $\mathsf{maxElem}(L, x)$ with $x \geq 2$ can be derived, as illustrated by the following examples.

$$\dfrac{\dfrac{\dfrac{\ldots}{\mathsf{maxElem}(L,2)}}{\mathsf{maxElem}(2{:}L,2)}}{\mathsf{maxElem}(1{:}2{:}L,2)} \qquad \dfrac{\dfrac{\dfrac{\ldots}{\mathsf{maxElem}(L,5)}}{\mathsf{maxElem}(2{:}L,5)}}{\mathsf{maxElem}(1{:}2{:}L,5)}$$

By adding the coaxiom, we force the element to belong to the list, so that wrong results are "filtered out". Indeed, the judgment $\mathsf{maxElem}(1{:}2{:}L, 2)$ has the infinite proof tree shown above, and each node has a finite proof tree in the inference system extended by the corule:

$$\dfrac{\dfrac{\dfrac{\ldots}{\mathsf{maxElem}(L,2)}}{\mathsf{maxElem}(2{:}L,2)}}{\mathsf{maxElem}(1{:}2{:}L,2)} \qquad \dfrac{\genfrac{}{}{2pt}{}{}{\mathsf{maxElem}(2{:}L,2)}}{\mathsf{maxElem}(1{:}2{:}L,2)}$$

On the other hand, the judgment $\mathsf{maxElem}(1{:}2{:}L, 5)$ has the infinite proof tree shown above, but has *no finite proof tree* in the inference system extended by the corule. Indeed, since 5 does not belong to the list, the corule can never be applied. Hence, this judgment cannot be derived in the inference system with corules. We refer to [8, 13, 15] for other examples.

Note that the inductive and coinductive interpretation of $\mathcal{I}$ are special cases, notably:

- the inductive interpretation of $\mathcal{I}$ is the interpretation of $\langle \mathcal{I}, \emptyset \rangle$
- the coinductive interpretation of $\mathcal{I}$ is the interpretation of $\langle \mathcal{I}, \{\langle \emptyset, j \rangle \mid j \in \mathcal{U}\} \rangle$.

4 Generalized inference systems in Agda

We describe how to implement (generalized) inference systems in Agda. Notably, in Section 4.1 we present an approach mimicking meta-rules, showing step-by-step the correspondence with the definitions of the previous sections. In Sect. 4.2, instead, we explain the view of inference systems as indexed containers, and prove the equivalence.

4.1 An Agda library for writing meta-rules

In this section and the following we report the most interesting parts of the Agda code.

As anticipated, the aim of the Agda library is to allow a user to write meta-rules as "on paper". To illustrate this format, let us consider, e.g., the previous example:

$$\text{(allP-t)}\ \dfrac{\mathsf{allPos}(xs)}{\mathsf{allPos}(x{:}xs)} \quad x > 0$$

In a meta-rule, we have *meta-variables*, which range over certain sets, in a way possibly restricted by a *side condition*. We call *context* the set of the instantiations of meta-variables which satisfy the side-condition, hence produce a rule of the inference system. In the example, there are two meta-variables, x and xs, which range over $\mathbb{N}$ and $\mathbb{N}^\infty$, respectively, with the restriction that x should be positive. Hence the context is $\{\langle x, l \rangle \in \mathbb{N} \times \mathbb{N}^\infty \mid x > 0\}$, see Sect. 5 for the Agda version of this meta-rule.

Correspondingly, the following Agda declaration defines a meta-rule as a record, parametric on the universe U. The first two components are the context and a set of positions for premises. For each element of the context (instantiation of meta-variables satisfying the side condition), the last two components produce the premises, one for each position, and the conclusion of the rule obtained by this instantiation.

```
record MetaRule {ℓc ℓp : Level} (U : Set ℓu) : Set _ where
  field
    Ctx : Set ℓc
    Pos : Set ℓp
    prems : Ctx → Pos → U
    conclu : Ctx → U

  RF[_] : ∀{ℓ} → (U → Set ℓ) → (U → Set _)
  RF[_] P u = Σ[ c ∈ Ctx ] (u ≡ conclu c × (∀ p → P (prems c p)))

  RClosed : ∀{ℓ} → (U → Set ℓ) → Set _
  RClosed P = ∀ c → (∀ p → P (prems c p)) → P (conclu c)
```

Recall that in Agda the declaration U : Set introduces the type (set) U, and P : U → Set the dependent type (predicate on U) P. For each element u of U, P u is the type of the proofs that u satifies P, hence P u inhabited means that u satisfies P. To avoid paradoxes, not every Agda type is in Set; there is an infinite sequence Set 0, Set 1, . . . , Set ℓ, . . . such that Set ℓ : Set (suc ℓ), where ℓ is a *level*, and Set is an abbreviation for Set 0. The programmer can write a wildcard for a level which can be inferred; to make the Agda code reported in the paper lighter, we sometimes use a wildcard even for a level which is explicit in the real code.

In the Agda code in this section, predicates P : U → Set encode subsets of the universe as in Sect. 2 and Sect. 3, so we speak of subsets and membership, rather than of predicates and satisfaction, to closely follow the previous formulation.

The function RF[_] encodes the inference operator associated with the meta-rule. Given a subset P of the universe, u belongs to the resulting subset if we can find an instantiation c of meta-variables satisfying the side condition, producing u as conclusion, and, for each position, a premise in P. Note the use of existential quantification Σ[x ∈ A] B where B depends on x.

The predicate RClosed encodes the property of being closed with respect to the meta-rule. A subset P of the universe is closed if, for each instantiation c of the meta-variables satisfying the side-condition, if all the premises are in P then the conclusion is in P as well. Note the use of universal quantification ∀ (x : A) → B, where B depends on x.

Since in practical cases meta-rules are very often *finitary*, that is, premises are a finite set, the library also offers an interface to write a (finitary) meta-rule, by providing, besides the context, two components which are the *vector* of premises, with fixed length n, and the conclusion. The injection from transforms this more concrete format in the generic one for meta-rules, by specifying that the set of positions is Fin n (the set of indexes from 0 to $n-1$).

```
record FinMetaRule {ℓc n} (U : Set ℓu) : Set _ where
  field
    Ctx : Set ℓc
    comp : C → Vec U n × U

  from : MetaRule {ℓc} {zero} U
  from .MetaRule.Ctx = Ctx
  from .MetaRule.Pos = Fin n
  from .MetaRule.prems c i = get (proj₁ (comp c)) i
  from .MetaRule.conclu c = proj₂ (comp c)
```

An inference system is defined as a record, parametric on the universe U, consisting of a set of meta-rule names and a family of meta-rules. The function ISF [_] and the predicate ISClosed are defined composing those given for a single meta-rule.

```
record IS {ℓc ℓp ℓn : Level} (U : Set ℓu) : Set _ where
  field
    Names : Set ℓn
    rules : Names → MetaRule {ℓc} {ℓp} U

  ISF[_] : ∀{ℓ} → (U → Set ℓ) → (U → Set _)
  ISF[_] P u = Σ[ rn ∈ Names ] RF[ rules rn ] P u

  ISClosed : ∀{ℓ} → (U → Set ℓ) → Set _
  ISClosed P = ∀ rn → RClosed (rules rn) P
```

Recall that the inductive interpretation $Ind⟦\mathcal{I}⟧$ of an inference system $\mathcal{I}$ is the set of elements of the universe which have a finite proof tree, and finite proof trees are, in turn, inductively defined, that is, by a least fixed point operator. In Agda, inductive structures are encoded as *datatypes*, which specify their constructors.

```
data Ind⟦_⟧ {ℓc ℓp ℓn : Level}
(is : IS {ℓc} {ℓp} {ℓn} U) : U → Set _ where
  fold : ∀ {u} → ISF[ is ] Ind⟦ is ⟧ u → Ind⟦ is ⟧ u
```

For each u, Ind ⟦ is ⟧ u is the type of the proofs that u satisfies Ind ⟦ is ⟧, which are essentially the finite proof trees[1] for u. Indeed, the fold constructor, given a proof that u can be derived by applying a rule from premises belonging to Ind ⟦ is ⟧, which essentially consists of a rule with conclusion u and finite proof trees for its premises, builds a finite proof tree for u.

The coinductive interpretation $CoInd⟦\mathcal{I}⟧$, instead, is the set of elements of the universe which have a possibly infinite proof tree, and possibly infinite proof trees are, in turn, coinductively defined, that is, by a greatest fixed point operator. In Agda, coinductive structures can be encoded in two different ways: either as *coinductive records* [3], or as datatypes by using the mechanism of *thunks* (suspended computations) together with *sized types* [1, 2, 4] to ensure termination. To allow compatibility with existing code implemented in either way, both versions are supported by the library.

```
record CoInd⟦_⟧ {ℓc ℓp ℓn : Level}
 (is : IS {ℓc} {ℓp} {ℓn} U) (u : U) : Set _ where
  coinductive
  constructor cofold_
  field
    unfold : ISF[ is ] CoInd⟦ is ⟧ u

  data SCoInd⟦_⟧ {ℓc ℓp ℓn : Level}
   (is : IS {ℓc} {ℓp} {ℓn} U) : U → Size → Set _ where
    sfold : ∀ {u i} → ISF[ is ] (λ u → Thunk (SCoInd⟦ is ⟧ u) i) u
      → SCoInd⟦ is ⟧ u i
```

For each u, CoInd ⟦ u ⟧ is the type of the proofs that u satisfies CoInd ⟦ is ⟧, which are essentially the possibly infinite proof trees for u, and analogously for SCoInd ⟦ is ⟧.

In the first version, a possibly infinite proof tree for u is a record with only one field unfold containing an element of ISF [is] CoInd ⟦ is ⟧ u, that is, a proof that u can be derived by applying a rule from premises belonging to CoInd ⟦ is ⟧, which essentially consists of a rule with conclusion u and possibly infinite proof trees for its premises.

[1] With some more structure, since the Agda proofs keep trace of the applied meta-rules.

In the second version, a possibly infinite proof tree is obtained by a **data** constructor, analogously to a finite one in the inductive interpretation; however, since proof trees are encoded as thunks, hence evaluated lazily, this encoding represents infinite trees as well. In other words, coinduction is "hidden" in the library type Thunk, which is a coinductive record with only one field force, intuitively representing the suspended computation.

The interpretation of a generalized inference system can then be encoded following exactly the definition in Sect. 3: it is the coinductive interpretation of I, restricted to rules whose conclusion is in the inductive interpretation of the (standard) inference system consisting of both rules I and corules C.

```
FCoInd⟦_,_⟧ : ∀{ℓc ℓp ℓn ℓn'} → (I : IS {ℓc} {ℓp} {ℓn} U)
  → (C : IS {ℓc} {ℓp} {ℓn'} U) → U → Set _
FCoInd⟦ I , C ⟧ = CoInd⟦ I ⊓ Ind⟦ I ∪ C ⟧ ⟧

SFCoInd⟦_,_⟧ : ∀{ℓc ℓp ℓn ℓn'} → (I : IS {ℓc} {ℓp} {ℓn} U)
  → (C : IS {ℓc} {ℓp} {ℓn'} U) → U → Size → Set _
SFCoInd⟦ I , C ⟧ = SCoInd⟦ I ⊓ Ind⟦ I ∪ C ⟧ ⟧
```

The definition is provided in two flavours where the coinductive interpretation is encoded by coinductive records and thunks, respectively, and uses two operators on inference systems, restriction ⊓ and union ∪. We report the former, which adds to each rule the side condition that the conclusion should satisfy P, as specified by the function addSideCond, omitted.

```
_⊓_ : ∀ {ℓc ℓp ℓn ℓ}{U : Set ℓu} → IS {ℓc} {ℓp} {ℓn} U
  → (U → Set ℓ) → IS {ℓc ⊔ ℓ} {_} {_} U
(is ⊓ P) .Names = is .Names
(is ⊓ P) .rules rn = addSideCond (is .rules rn) P
```

The library also provides the proofs of relevant properties, e.g., that closed sets coincide with pre-fixed points, and consistent sets coincide with post-fixed points. Moreover, it is shown that the two versions of encoding of the coinductive interpretation (by coinductive records and thunks) are equivalent. Finally, the library provides the induction, coinduction, and bounded coinduction principles. We only report here the statements.

```
ind[_] : ∀{ℓc ℓp ℓn ℓ}
    → (is : IS {ℓc} {ℓp} {ℓn} U)          -- IS
    → (S : U → Set ℓ)                     -- specification
    → ISClosed is S                       -- S is closed
    → Ind⟦ is ⟧ ⊆ S
```

If S is closed, then each element of the inductively defined set Ind ⟦ is ⟧ satisfies S.

```
coind[_] : ∀{ℓc ℓp ℓn ℓ}
    → (is : IS {ℓc} {ℓp} {ℓn} U)
    → (S : U → Set ℓ)
    → (S ⊆ ISF[ is ] S)                   -- S is consistent
    → S ⊆ CoInd⟦ is ⟧
```

If S is consistent, then each element satisfying S is in the coinductively defined set CoInd ⟦ is ⟧.

```
bounded-coind[_,_] : ∀{ℓc ℓp ℓn ℓn' ℓ}
    → (I : IS {ℓc} {ℓp} {ℓn} U)
    → (C : IS {ℓc} {ℓp} {ℓn'} U)
    → (S : U → Set ℓ)
    → S ⊆ Ind⟦ I ∪ C ⟧                    -- S is bounded w.r.t. I ∪ C
    → S ⊆ ISF[ I ] S                      -- S is consistent w.r.t. I
    → S ⊆ FCoInd⟦ I , C ⟧
```

If S is bounded, and consistent with respect to I, then each element which satisfies S belongs to the set FCoInd ⟦ I , C ⟧ defined by flexible coinduction.

Another easy theorem useful in proofs is that $\mathit{FCoInd}[\![\mathcal{I},\mathcal{I}_{\mathsf{co}}]\!] \subseteq \mathit{Ind}[\![\mathcal{I}\cup\mathcal{I}_{\mathsf{co}}]\!]$:

```
fcoind-to-ind : ∀{ℓc ℓp ℓn ℓn'}
    {is : IS {ℓc} {ℓp} {ℓn} U}{cois : IS {ℓc} {ℓp} {ℓn'} U}
    → FCoInd⟦ is , cois ⟧ ⊆ Ind⟦ is ∪ cois ⟧
```

4.2 Inference systems as indexed containers

Indexed containers [6] are a rather general notion, meant to capture families of datatypes with some form of indexing. They are part of the Agda standard library. We report below the definition, simplified and adapted a little for presentation purpose. Notably, we use ad-hoc field names, chosen to reflect the explanation provided below.

```
record Container {ℓi ℓo}
 (I : Set ℓi) (O : Set ℓo) (ℓc ℓp : Level) : Set _ where
  constructor _ ◁ _/_
  field
    Cons  : (o : O) → Set ℓc
    Pos : ∀ {o} → Cons o → Set ℓp
    input : ∀ {o} (c : Cons o) → Pos c → I

⟦_⟧: ∀ {ℓi ℓo ℓc ℓp ℓ} {I : Set ℓi} {O : Set ℓo} → Container I O ℓc ℓp →
      (I → Set ℓ) → (O → Set _)
⟦ C ◁ P / inp ⟧ X o = Σ[ c ∈ C o ] ((p : P c) → X (inp c p))
```

To explain the view of an inference system as an indexed container, we can think of the latter as describing a family of datatype constructors where I and O are input and output sorts, respectively. Then, Cons specifies, for each output sort o, the set of its constructors; for each constructor for o, Pos specifies a set of positions to store inputs to the constructor; finally, input specifies the input sort for each position in a constructor.

The function ⟦_⟧ models the “semantics” of an indexed container, that is, given a family of inputs X indexed by I, it returns the family of outputs indexed by O which can be constructed by providing to some constructor inputs from P of correct sorts.

Then, inference systems can be defined as indexed containers where input and output sorts coincide, and are the elements of the universe, as follows.

```
  ISCont : {ℓc ℓp : Level} → (U : Set ℓu) → Set _
  ISCont {ℓc} {ℓp} U = Container U U ℓc ℓp
```

In this way, for each u : U:

- Cons u is the set of (indexes for) all the rules which have consequence u
- Pos c is the set of (indexes for) the premises of the c-th rule
- input c p is the p-th premise of the c-th rule.

This view comes out quite naturally observing that an inference system is an element of $\wp(\wp(\mathcal{U})\times\mathcal{U})$; equivalently, a function which, for each $j\in\mathcal{U}$, returns the set of the sets of premises of all the rules with consequence j. In a constructive setting such as Agda, the powerset construction is not available, hence we have to use functions. So, for each element u, we need a type to index all rules with consequence u, and, for each rule, a type to index its premises, which are exactly the data of an indexed container. In other words, this view of inference systems as indexed containers explicitly interprets rules as constructors for proofs.

Moreover, definitions in Sect. 2 can be easily obtained as instances of analogous definitions for indexed containers, building on the fact that the inference operator associated with an inference system turns out to be the semantics ⟦_⟧ of the corresponding container.

Whereas this encoding allows reuse of notions and code, a drawback is that information is structured in a rather different way from that "on paper"; notably, we group together rules with the same consequence, rather than those obtained as instances of the same "schema", that is, meta-rule. For instance, the inference system for allPos would be as follows:

```
allPosCont : ISCont (Colist ℕ ∞)
allPosCont .Cons [] = ⊤
allPosCont .Cons (x :: xs) = x > 0
allPosCont .Pos {[]} c = ⊥
allPosCont .Pos {x :: xs} c = Fin 1
allPosCont .input {x :: xs} c zero = xs .force
```

For this reason we developed the Agda library mimicking meta-rules described in Sect. 4.1, and we use this library for the examples in the following sections.

However, we can prove that the two notions are equivalent, as shown below. To this end, we define a translation C[_] from inference systems as in Sect. 4.1 to indexed containers, and a converse translation IS [_]. Note that in the translation C[_] each meta-rule is transformed in all its instantiations; more precisely, for each u : C, Cons u gives all the instantiations of meta-rules having u as consequence. Conversely, in the translation IS [_], each rule is transformed in a meta-rule with trivial context.

```
C[_] : ∀{ℓc ℓp ℓn} → IS {ℓc} {ℓp} {ℓn} U → Container U U _ ℓp
C[ is ] .Cons u = Σ[ rn ∈ is .Names ] Σ[ c ∈ is .rules rn .Ctx ]
    u ≡ is .rules rn .conclu c
C[ is ] .Pos (rn , _ , refl) = is .rules rn .Pos
C[ is ] .input (rn , c , refl) p = is .rules rn .prems c p

IS[_] : ∀{ℓc ℓp} → Container U U ℓc ℓp → IS {zero} {ℓp} {l ⊔ ℓc} U
IS[ C ] .Names = Σ[ u ∈ U ] C .Cons u
IS[ C ] .rules (u , c) =
  record {
    Ctx = ⊤ ;
    Pos = C .Pos c ;
    prems = λ _ r → C .input c r ;
    conclu = λ _ → u }

isf−to−c : ∀{ℓc ℓp ℓn ℓp} {is : IS {ℓc} {ℓp} {ℓn} U}{P : U → Set ℓp}
    → ISF[ is ] P ⊆ ⟦ C[ is ] ⟧ P
  isf−to−c (rn , c , refl , pr) = (rn , c , refl) , pr

c−to−isf : ∀{l' ℓp ℓp} {C : Container U U l' ℓp}{P : U → Set ℓp}
    → ⟦ C ⟧ P ⊆ ISF[ IS[ C ] ] P
c−to−isf (c , pr) = (_ , c) , tt , refl , pr
```

5 Using the library

We show how to use the library to define specific inference systems and prove their properties. Consider the examples in Sect. 2 and Sect. 3. For member, the universe are pairs of elements and possibly infinite lists, implemented by the Agda library Colist which uses thunks:

```
U = A × Colist A ∞
data memberRN : Set where mem-h mem-t : memberRN
```

```
mem-h-r : FinMetaRule U
mem-h-r .Ctx = A × Thunk (Colist A) ∞
mem-h-r .comp (x , xs) =
  [] ,
  ──────────────
  (x , x :: xs)

mem-t-r : FinMetaRule U
mem-t-r .Ctx = A × A × Thunk (Colist A) ∞
mem-t-r .comp (x , y , xs) =
  ((x , xs .force) :: []) ,
  ──────────────
  (x , y :: xs)

memberIS : IS U
memberIS .Names = memberRN
memberIS .rules mem-h = from mem-h-r
memberIS .rules mem-t = from mem-t-r
```

Here memberRN are the rule names, and each rule name has an associated element of FinMetaRule U, which exactly encodes the meta-rule in Sect. 2. Note, in mem-t-r, the use of the force field of Thunk to actually obtain the tail colist.

This inference system is expected to define exactly the pairs (x , xs) such that x belongs to xs, that is, those satisfying the following specification

```
memSpec : U → Set
memSpec (x , xs) = Σ[ i ∈ ℕ ] (Colist.lookup i xs ≡ just x)
```

where the library function lookup : ℕ → Colist A ∞ → Maybe A returns the i-th element of xs, if any.

As said in Sect. 2, to obtain the desired meaning this inference system has to be interpreted inductively, and soundness can be proved by the induction principle, that is, by providing a proof that the specification is closed with respect to the two meta-rules, as shown below.

```
_member_ : A → Colist A ∞ → Set
x member xs = Ind⟦ memberIS ⟧ (x , xs)

memSpecClosed : ISClosed memberIS memSpec
memSpecClosed mem-h _ _ = zero , refl
memSpecClosed mem-t _ pr =
  let (i , proof) = pr Fin.zero in (suc i) , proof

memberSound : ∀ {x xs} → x member xs → memSpec (x , xs)
memberSound = ind[memberIS] memSpec memSpecClosed
```

For completeness there is no canonical technique; in this example, it can be proved by induction on the position (the index i in the specification).

For allPos , the universe are possibly infinite lists.

```
U : Set
U = Colist ℕ ∞
data allPosRN : Set where allP-Λ allP-t : allPosRN

allP-Λ-r : FinMetaRule U
allP-Λ-r .Ctx = ⊤
allP-Λ-r .comp c =
  [] ,
  ──────────────
  []
```

```
allP-t-r : FinMetaRule U
allP-t-r .Ctx = Σ[ (x , _) ∈ ℕ × Thunk (Colist ℕ) ∞ ] x > 0
allP-t-r .comp ((x , xs) , _) =
  ((xs .force) :: []) ,
  ──────────────────
  x :: xs)

allPosIS : IS U
allPosIS .Names = allPosRN
allPosIS .rules allP-Λ = from allP-Λ-r
allPosIS .rules allP-t = from allP-t-r
```

This inference system is expected to define exactly the lists such that all elements are positive, that is, those satisfying the following specification (where for simplicity, we use the predicate ∈, omitted, directly defined inductively).

```
allPosSpec : U → Set
allPosSpec xs = ∀ {x} → x ∈ xs → x > 0
```

As said in Sect. 2, to obtain the desired meaning this inference system has to be interpreted coinductively, and completeness can be proved by the coinduction principle, that is, by providing a proof that the specification is consistent with respect to the inference system, as shown below.

```
allPos : U → Set
allPos = CoInd⟦ allPosIS ⟧

allPosSpecCons : ∀ {xs} → allPosSpec xs → ISF[ allPosIS ] allPosSpec xs
allPosSpecCons {[]} _ = allP-Λ , (tt , (refl , tt , λ ()))
allPosSpecCons {(x :: xs)} Sxs =
  allP-t ,
  ((x , xs) , (refl , (Sxs here , λ {Fin.zero → λ mem → Sxs (there mem)})))

allPosComplete : allPosSpec ⊆ allPos
allPosComplete = coind[ allPosIS ] allPosSpec allPosSpecCons
```

For soundness there is no canonical technique; in this example, when the colist is empty the proof that the specification holds is trivial. If the colist is not empty, then the proof proceeds by induction on the position of the element to be proved to be positive.

Finally, for maxElem, the universe are pairs of natural numbers and possibly infinite lists.

```
U : Set
U = ℕ × Colist ℕ ∞
data maxElemRN : Set where max-h max-t : maxElemRN
data maxElemCoRN : Set where co-max-h : maxElemCoRN

max-h-r : FinMetaRule U
max-h-r .Ctx = Σ[ (_ , xs) ∈ ℕ × Thunk (Colist ℕ) ∞ ] xs .force ≡ []
max-h-r .comp ((x , xs) , _) =
  [] ,
  ──────────────
  x , x :: xs

max-t-r : FinMetaRule U
max-t-r .Ctx =
  Σ[ (x , y , z , _) ∈ ℕ × ℕ × ℕ × Thunk (Colist ℕ) ∞ ] z ≡ max x y
max-t-r .comp ((x , y , z , xs) , _) =
  (x , xs .force) :: [] ,
  ──────────────
```

```
   z , y :: xs

co-max-h-r : FinMetaRule U
co-max-h-r .Ctx = ℕ × Thunk (Colist ℕ) ∞
co-max-h-r .comp (x , xs) =
   [] ,
  ─────────────────
   (x , x :: xs)

maxElemIS : IS U
maxElemIS .Names = maxElemRN
maxElemIS .rules max-h = from max-h-r
maxElemIS .rules max-t = from max-t-r

maxElemCoIS : IS U
maxElemCoIS .Names = maxElemCoRN
maxElemCoIS .rules co-max-h = from co-max-h-r
```

Note that in this example we have defined two inference systems, the rules and the corules. This generalized inference system is expected to define exactly the pairs `(x , xs)` such that `x` is the maximal element of `xs`, that is, those satisfying the following specification, where to be the maximal element `x` should belong to `xs`, and be greater or equal than any `n` in `xs`.

```
maxSpec inSpec geqSpec : U → Set
inSpec (x , xs) = x ∈ xs
geqSpec (x , xs) = ∀{n} → n ∈ xs → x ≡ max x n
maxSpec u = inSpec u × geqSpec u
```

As said in Sect. 3, the desired meaning is provided by the interpretation of the generalized inference system.

```
_maxElem_ : ℕ → Colist ℕ ∞ → Set
x maxElem xs = FCoInd⟦ maxElemIS , maxElemCoIS ⟧ (x , xs)
```

and completeness can be proved by the bounded coinduction principle, see (bcoind) at page 4.

```
maxElemComplete : ∀{x xs} → maxSpec (x , xs) → x maxElem xs
maxElemComplete =
   bounded-coind[ maxElemIS , maxElemCoIS ] maxSpec
     (λ{(x , xs) → maxSpecBounded x xs}) λ{(x , xs) → maxSpecCons x xs}
```

Notably, we have to prove that the specification is:

- bounded, that is, contained in `_maxElem_i_`, the inductive interpretation of the standard inference system consisting of both rules and corules, as shown below:

  ```
  _maxElem_i_ : ℕ → Colist ℕ ∞ → Set
  x maxElem_i xs = Ind⟦ maxElemIS ∪ maxElemCoIS ⟧ (x , xs)

  maxSpecBounded : ∀{x xs} → inSpec (x , xs)
    → geqSpec (x , xs) → x maxElem_i xs
  ```

- consistent with respect to the inference system consisting of only rules, as shown below:

  ```
  maxSpecCons : ∀{x xs} → inSpec (x , xs) →
    geqSpec (x , xs) → ISF[ maxElemIS ] maxSpec (x , xs)
  ```

These proofs are omitted.

For soundness there is no canonical technique. The proof can be split for the two components of the specification. It is worth noting that, for soundness with respect to `inSpec`, we first use `fcoind -to-ind` at page 9, and then define `maxElemSound -in-ind`, omitted, by induction on the inference system consisting of rules and corules. The use of fcoind-to-ind in the proof corresponds to the fact that without corules unsound judgments could be derived, see Sect. 3.

```
maxElemSound-in : ∀ {x xs} → x maxElem xs → inSpec (x , xs)
maxElem-sound-in max = maxElemSound-in-ind (fcoind-to-ind max)
```

Soundness with respect to geqSpec is proved by induction on the position, that is, the proof of membership, of the element that must be proved to be less or equal. In this case, soundness would hold even in the purely coinductive case.

6 A worked example

We describe a more significant example of instantiation: an inference system with corules providing a big-step semantics of lambda-calculus including divergence among the possible results [9], reported in Fig. 1. In this example, corules play a key role: indeed , considering, e.g., the divergent term $\Omega = (\lambda x.x)\,(\lambda x.x)$, in the standard inductive big-step semantics no result can be derived (an infinite proof tree is needed), as for a stuck term; in the purely coinductive interpretation, any judgment $\Omega \Downarrow v^\infty$ would be obtained [19]. Since each node of the infinite proof tree for a judgment should also have a finite proof tree using the corules, the coaxiom (coa) forces to obtain only ∞ as result, see [9] for a more detailed explanation.[2]

$$
\begin{array}{rcll}
t & ::= & v \mid x \mid t_1\,t_2 \mid \ldots & \text{term}\\
v & ::= & \lambda x.t \mid \ldots & \text{value}\\
v^\infty & ::= & v \mid \infty & \text{result}
\end{array}
$$

$$
\text{(coa)}\ \frac{}{e \Downarrow \infty} \qquad \text{(val)}\ \frac{}{v \Downarrow v}
$$

$$
\text{(app)}\ \frac{t_1 \Downarrow \lambda x.t \quad t_2 \Downarrow v \quad t[x/v] \Downarrow v^\infty}{t_1\ t_2 \Downarrow v^\infty}
$$

$$
\text{(l-div)}\ \frac{t_1 \Downarrow \infty}{t_1\ t_2 \Downarrow \infty} \qquad \text{(r-div)}\ \frac{t_1 \Downarrow v \quad t_2 \Downarrow \infty}{t_1\ t_2 \Downarrow \infty}
$$

Figure 1 λ-calculus: syntax and big-step semantics.

In rule (app), v^∞ is used for the result, so the rule also covers the case when the evaluation of the body of the lambda abstraction diverges. As usual, $t[x/v]$ denotes capture-avoiding substitution. Rules (l-div) and (r-div) cover the cases when either t_1 or t_2 diverges, assuming a left-to-right evaluation strategy.

Terms, values, and results are inductively defined, hence encoded by Agda datatypes. As customary in implementations of lambda-calculus, we use the De Bruijn notation: notably, Term n is the set of terms with n free variables.

```
data Term (n : ℕ) : Set where
  var : Fin n → Term n
  lambda : Term (suc n) → Term n
  app : Term n → Term n → Term n

data Value : Set where lambda : Term 1 → Value

term : Value → Term 0
term (lambda x) = lambda x

data Value∞ : Set where
```

[2] Other examples of big-step semantic definitions with more sophisticated corules are given in [10, 7].

```
  res : Value → Value∞
  ∞ : Value∞
```

The universe consists of big-step judgments (pairs consisting of a term and a result).

```
U : Set
U = Term 0 × Value∞
```

The two inference systems of rules and corules are encoded below:

```
data BigStepRN : Set where val app l—div r—div : BigStepRN
data BigStepCoRN : Set where COA : BigStepCoRN

BigStepIS : IS U
BigStepIS .Names = BigStepRN
BigStepIS .rules val = from val-r
BigStepIS .rules app = from app-r
BigStepIS .rules L-DIV = from l-div-r
BigStepIS .rules R-DIV = from r-div-r

BigStepCoIS : IS U
BigStepCoIS .Names  = BigStepCoRN
BigStepCoIS .rules COA = from coa-r
```

where BigStepRN are the rule names, and each rule name has an associated element of FinMetaRule U. For instance, app-r is given below. The auxiliary function subst , omitted, implements capture-avoiding substitution.

```
app-r : FinMetaRule U
app-r .Ctx = Term 0 × Term 1 × Term 0 × Value × Value∞
app-r .comp (t1 , t , t2 , v , v∞) =
 (t1 , res (lambda t)) :: (t2 , res v) :: (subst t (term v) , v∞) :: [] ,
    ————————————————————————————
  (app t1 t2 , v∞)
```

The big-step semantics can be obtained as the interpretation of the generalized inference system, as shown below. We use the flavour with thunks.

```
_⇓_ : Term 0 → Value∞ → Size → Set
(t ⇓ v∞) i = SFCoInd⟦ BigStepIS , BigStepCoIS ⟧ (t , v∞) i

_⇓ᵢ_ : Term 0 → Value∞ → Set
t ⇓ᵢ v∞ = Ind⟦ BigStepIS ∪ BigStepCoIS ⟧ (t , v∞)
```

The second predicate (i stands for "inductive") models that a judgment has a finite proof tree in the inference system consisting of rules and coaxiom, and will be used in proofs.

Small-step semantics, reported in Fig. 2, can also be obtained appropriately instantiating

$$(\beta)\ \frac{}{(\lambda x.t)\ v \Rightarrow t[x/v]} \qquad (\text{l-app})\ \frac{t_1 \Rightarrow t_1'}{t_1\ t_2 \Rightarrow t_1'\ t_2} \qquad (\text{r-app})\ \frac{t_2 \Rightarrow t_2'}{v\ t_2 \Rightarrow v\ t_2'}$$

Figure 2 λ-calculus: small-step semantics.

the library. In this case, the universe consists of small-step judgments, which are pairs of terms. There is only one inference system, where SmallStepRN are the rule names, and each rule name has an associated element of FinMetaRule U.

```
U : Set
U = Term 0 × Term 0
```

```
data SmallStepRN : Set where β L-app R-app : SmallStepRN

SmallStepIS : IS U
SmallStepIS .Names = SmallStepRN
SmallStepIS .rules β = from β-r
SmallStepIS .rules L-app = from l-app-r
SmallStepIS .rules R-app = from r-app-r
```

For instance, β-r is given below.

```
β-r : FinMetaRule U
β-r .Ctx = Term 1 × Value
β-r .comp (t , v) =
  [] ,
  ───────────────────────────
  (app (lambda t) (term v) , subst t (term v))
```

The one-step relation $\Rightarrow$ is obtained as the inductive interpretation of the (standard) inference system. Then, finite computations are modeled by its reflexive and transitive closure $\Rightarrow^\star$, defined using Star in the Agda library, as shown below.

```
_⇒_ : Term 0 → Term 0 → Set
t ⇒ t' = Ind⟦ SmallStepIS ⟧ (t , t')

_⇒*_ : Term 0 → Term 0 → Set
_⇒*_ = Star _⇒_
```

Infinite computations, instead, are modeled by the relation $\Rightarrow^\infty$, coinductively defined by the meta-rule $\dfrac{t' \Rightarrow^\infty}{t \Rightarrow^\infty}\; t \Rightarrow t'$, encoded in Agda by thunks.

```
  data _⇒∞ : Term 0 → Size → Set where
    step : ∀ {t t' i} → t ⇒ t' → Thunk (t' ⇒∞) i → t ⇒∞ i
```

The proof of equivalence between big-step and small-step semantics is structured as follows, where $\mathcal{S} = \{\langle t, v\rangle \mid t \Rightarrow^\star v\} \cup \{\langle t, \infty\rangle \mid t \Rightarrow^\infty\}$.

Soundness

$t \Downarrow v$ implies $t \Rightarrow^\star v$ We use fcoind -to-ind at page 9, and then reason by induction on the judgment $t \Downarrow_{\mathsf{i}} v$. That is, we show that $t \Rightarrow^\star v$ is closed w.r.t. the inference system consisting of rules and corules. As already pointed out for the maxElem example, the use of fcoind -to-ind in the proof corresponds to the fact that, without the coaxiom (coa), unsound judgments would be derived, e.g., $\Omega \Downarrow v$ for $v \in \mathsf{Val}$.

$t \Downarrow \infty$ implies $t \Rightarrow^\infty$ This implication, instead, would hold even in the purely coinductive case. It can be proved from *progress* and *subject reduction* properties:

Progress $t \Downarrow \infty$ implies that there exists t' such that $t \Rightarrow t'$.

Subject reduction $t \Downarrow \infty$ and $t \Rightarrow t'$ implies $t' \Downarrow \infty$.

Completeness By bounded coinduction, see (bcoind) at page 4.

Boundedness

$t \Rightarrow^\star v$ implies $t \Downarrow_{\mathsf{i}} v$ By induction on the number of steps.

$t \Rightarrow^\infty$ implies $t \Downarrow_{\mathsf{i}} \infty$ Trivial, since the coaxiom coa can be applied.

Consistency We have to show that, for each $\langle t, v^\infty\rangle \in \mathcal{S}$, $\langle t, v^\infty\rangle$ is the consequence of a big-step rule where the premises are in $\mathcal{S}$ as well. We distinguish two cases.

$t \Rightarrow^\star v$ By induction on the number of steps. If it is 0, then t is a value, hence we can use rule (val). Otherwise, t is an application, and we can use rule (app).

$t \Rightarrow^\infty$ The term t is an application $t_1\, t_2$. We distinguish the following cases:

- t_1 diverges, hence we can use rule (l-div)

- t_1 converges and t_2 diverges, hence we can use rule (r-div)
- both t_1 and t_2 converge, hence we can use rule (app).

Note that in this proof by cases we need to use the *excluded middle* principle, which is defined in the standard library, and postulated in our proof.

7 Conclusion

We have presented an Agda implementation of inference systems which, besides the standard inductive and coinductive interpretations, supports also flexible coinduction and the associated proof principle. The key feature is that the library allows the separation of the definitions from their semantics, thus enabling modular composition and reasoning. This is particularly useful for flexible coinduction, because the interpretation of a generalized inference system is just defined by mixing the inductive and the coinductive interpretations of two inference systems built from rules and corules.

Of course, as Agda supports both inductive and coinductive dependent types, one could directly write Agda code for inductive, coinductive and even flexible coinductive definitions of concrete examples. We have explored this possibility in [12]. However, in this way, the definition is hard-wired with its semantics, and, for flexible coinduction, one has to manually construct the interpretation by combining in the correct way an inductive and a coinductive type and to prove the bounded coinduction principle for each example. For instance, the definition of maxElem will look as follows:

```
data _maxElem_ : ℕ → CoList ℕ ∞ → Size → Set where
  max-h : ∀ {x xs i} →force xs ≡ [] → x maxElem (x :: xs) i
  max-t : ∀ {x y xs i} → Thunk (x maxElem (force xs)) i
                         → z ≡ max x y
                         → z maxElem_i (y :: xs)
                         → z maxElem (y :: xs) i

data _maxElem_i_ : ℕ → CoList ℕ ∞ → Set where
  imax-h : ∀ {x xs} →force xs ≡ [] → x maxElem_i (x :: xs)
  imax-t : ∀ {x y xs} → x maxElem_i (force xs)) → z ≡ max x y
                        → z maxElem_i (y :: xs)
  co-max-h : ∀ {x xs} → x maxElem_i (x :: xs)
```

Clearly, this approach causes duplication of rules and code, as rules of the coinductive type have to be duplicated in the inductive one, making things rather complex. Our library instead hides all these details, exposing interfaces for interpretations and proof principles, so that the user only has to write code describing rules.

For future work we plan to extend the library in several directions. The first one is to support other interpretations of inference systems, such as the *regular* one [14], which is basically coinductive but allows only proof trees with finitely many distinct subtrees.To this end, useful starting points are works on regular terms and streams [22, 24] and on finite sets [17] in dependent type theories. The challenging part is the finiteness constraint, which is not trivial in a type-theoretic setting. A second direction is to implement other proof techniques for (flexible) coinduction, as parametrized coinduction [18] and up-to techniques [20, 16]. Finally, another direction could be the development of a full framework for composition of inference systems, along the lines of seminal work on module systems [11]. On the more practical side, a further development is to transform the methodology in an automatic translation. That is, a user should be allowed to write an inference system (with corules) in a natural syntax, and the corresponding Agda types should be generated automatically, either by an external tool, or, more interestingly, using *reflection*, recently added in Agda.

References

1 Andreas Abel. Type-based termination, inflationary fixed-points, and mixed inductive-coinductive types. In Dale Miller and Zoltán Ésik, editors, *Proceedings 8th Workshop on Fixed Points in Computer Science, FICS 2012*, volume 77 of *EPTCS*, pages 1–11, 2012. doi:10.4204/EPTCS.77.1.

2 Andreas Abel and Brigitte Pientka. Well-founded recursion with copatterns and sized types. *Journal of Functional Programming*, 26:e2, 2016. doi:10.1017/S0956796816000022.

3 Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: programming infinite structures by observations. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM Symposium on Principles of Programming Languages, POPL'13*, pages 27–38. ACM Press, 2013. doi:10.1145/2429069.2429075.

4 Andreas Abel, Andrea Vezzosi, and Théo Winterhalter. Normalization by evaluation for sized dependent types. *Proceedings of ACM on Programming Languages*, 1(ICFP):33:1–33:30, 2017. doi:10.1145/3110277.

5 Peter Aczel. An introduction to inductive definitions. In Jon Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, pages 739–782. Elsevier, 1977.

6 Thorsten Altenkirch, Neil Ghani, Peter G. Hancock, Conor McBride, and Peter Morris. Indexed containers. *Journal of Functional Programming*, 25, 2015. doi:10.1017/S095679681500009X.

7 Davide Ancona, Francesco Dagnino, Jurriaan Rot, and Elena Zucca. A big step from finite to infinite computations. *Science of Computer Programming*, 197:102492, 2020. doi:10.1016/j.scico.2020.102492.

8 Davide Ancona, Francesco Dagnino, and Elena Zucca. Generalizing inference systems by coaxioms. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017*, volume 10201 of *Lecture Notes in Computer Science*, pages 29–55, Berlin, 2017. Springer. doi:10.1007/978-3-662-54434-1_2.

9 Davide Ancona, Francesco Dagnino, and Elena Zucca. Reasoning on divergent computations with coaxioms. *Proceedings of ACM on Programming Languages*, 1(OOPSLA):81:1–81:26, 2017. doi:10.1145/3133905.

10 Davide Ancona, Francesco Dagnino, and Elena Zucca. Modeling infinite behaviour by corules. In Todd D. Millstein, editor, *32nd European Conference on Object-Oriented Programming, ECOOP 2018*, volume 109 of *LIPIcs*, pages 21:1–21:31, Dagstuhl, 2018. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECOOP.2018.21.

11 G. Bracha. *The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Department of Comp. Sci., Univ. of Utah, 1992.

12 Luca Ciccone. Flexible coinduction in Agda. Master's thesis, University of Genova, 2019. URL: https://arxiv.org/abs/2002.06047.

13 Francesco Dagnino. Coaxioms: flexible coinductive definitions by inference systems. *Logical Methods in Computer Science*, 15(1), 2019. doi:10.23638/LMCS-15(1:26)2019.

14 Francesco Dagnino. Foundations of regular coinduction. Technical report, DIBRIS, University of Genova, June 2020. Submitted for journal publication. URL: https://arxiv.org/abs/2006.02887.

15 Francesco Dagnino. *Flexible Coinduction*. PhD thesis, DIBRIS, University of Genova, 2021.

16 Nils Anders Danielsson. Up-to techniques using sized types. *Proceedings of ACM on Programming Languages*, 2(POPL):43:1–43:28, 2018. doi:10.1145/3158131.

17 Denis Firsov and Tarmo Uustalu. Dependently typed programming with finite sets. In Patrick Bahr and Sebastian Erdweg, editors, *Proceedings of the 11th ACM Workshop on Generic Programming, WGP@ICFP 2015*, pages 33–44. ACM, 2015. doi:10.1145/2808098.2808102.

18 Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The power of parameterization in coinductive proof. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM Symposium on Principles of Programming Languages, POPL'13*, pages 193–206. ACM Press, 2013. doi:10.1145/2429069.2429093.

19 Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009. doi:10.1016/j.ic.2007.12.004.

20 Damien Pous. Coinduction all the way up. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS'16*, pages 307–316. ACM Press, 2016. doi:10.1145/2933575.2934564.

21 Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, USA, 2011.

22 Régis Spadotti. *A mechanized theory of regular trees in dependent type theory. (Une théorie mécanisée des arbres réguliers en théorie des types dépendants)*. PhD thesis, Paul Sabatier University, Toulouse, France, 2016. URL: https://tel.archives-ouvertes.fr/tel-01589656.

23 The Agda Team. *The Agda Reference Manual*. URL: http://agda.readthedocs.io/en/latest/index.html.

24 Tarmo Uustalu and Niccolò Veltri. Finiteness and rational sequences, constructively. *Journal of Functional Programming*, 27:e13, 2017. doi:10.1017/S0956796817000041.

A Verified Decision Procedure for Univariate Real Arithmetic with the BKR Algorithm

Katherine Cordwell[1]
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Yong Kiam Tan
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

André Platzer
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

We formalize the univariate fragment of Ben-Or, Kozen, and Reif's (BKR) decision procedure for first-order real arithmetic in Isabelle/HOL. BKR's algorithm has good potential for parallelism and was designed to be used in practice. Its key insight is a clever recursive procedure that computes the set of all consistent sign assignments for an input set of univariate polynomials while carefully managing intermediate steps to avoid exponential blowup from naively enumerating all possible sign assignments (this insight is fundamental for both the univariate case and the general case). Our proof combines ideas from BKR and a follow-up work by Renegar that are well-suited for formalization. The resulting proof outline allows us to build substantially on Isabelle/HOL's libraries for algebra, analysis, and matrices. Our main extensions to existing libraries are also detailed.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases quantifier elimination, matrix, theorem proving, real arithmetic

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.14

Supplementary Material Our formalization is available on the Archive of Formal Proofs (AFP).
Model (formal proof development): `https://www.isa-afp.org/entries/BenOr_Kozen_Reif.html` [8]

Funding This material is based upon work supported by the National Science Foundation under Grant No. CNS-1739629 and by the AFOSR under grant number FA9550-16-1-0288. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or of AFOSR.
Katherine Cordwell: This material is based upon work supported by a National Science Foundation Graduate Research Fellowship under Grants Nos. DGE1252522 and DGE1745016.
Yong Kiam Tan: The second author was supported by A*STAR, Singapore.

Acknowledgements We thank Brandon Bohrer, Fabian Immler, and Wenda Li for useful discussions about Isabelle/HOL and its libraries. We also thank the ITP'21 anonymous reviewers and Magnus Myreen for helpful feedback on earlier drafts of this paper.

1 Introduction

Formally verified arithmetic has important applications in formalized mathematics and rigorous engineering domains. For example, real arithmetic questions (first-order formulas in the *theory of real closed fields*) often arise as part of formal proofs for safety-critical cyber-physical systems (CPS) [29], the formal proof of the Kepler conjecture involves the verification of more than 23, 000 real inequalities [13], and the verification of floating-point

[1] Corresponding author

algorithms also involves real arithmetic reasoning [14]. Some real arithmetic questions involve $\forall$ and $\exists$ quantifiers; these *quantified* real arithmetic questions arise in, e.g., CPS proofs, geometric theorem proving, and stability analysis of models of biological systems [35].

Quantifier elimination (QE) is the process by which a quantified formula is transformed into a logically equivalent quantifier-free formula. Tarski famously proved that the theory of first-order real arithmetic ($\mathrm{FOL}_{\mathbb{R}}$) admits QE; $\mathrm{FOL}_{\mathbb{R}}$ validity and satisfiability are therefore decidable by QE and evaluation [36]. Thus, *in theory,* all it takes to rigorously answer *any* real arithmetic question is to verify a QE procedure for $\mathrm{FOL}_{\mathbb{R}}$. However, *in practice*, QE algorithms for $\mathrm{FOL}_{\mathbb{R}}$ are complicated and the fastest known QE algorithm, *cylindrical algebraic decomposition (CAD)* [6] is, in the worst case, doubly exponential in the number of variables. The multivariate CAD algorithm is highly complicated and has yet to be fully formally verified in a theorem prover [18], although various specialized approaches have been used to successfully tackle restricted subsets of real arithmetic questions in proof assistants, e.g., quantifier elimination for linear real arithmetic [25], sum-of-squares witnesses [15] or real Nullstellensatz witnesses [30] for the universal fragment, and interval arithmetic when the quantified variables range over bounded domains [16, 33].

There are few general-purpose formally verified decision procedures for $\mathrm{FOL}_{\mathbb{R}}$. Mahboubi and Cohen [5] formally verified an algorithm for QE based on Tarski's proof but their formalization is primarily a theoretical decidability result [5, Section 1] owing to the non-elementary complexity of Tarski's algorithm. The proof-producing procedure by McLaughlin and Harrison [22] can solve a number of small multivariate examples but suffers similarly from the complexity of the underlying Cohen-Hörmander procedure. The situation for *univariate real arithmetic* (i.e., problems that involve only a single variable) is better. In Isabelle/HOL, Li, Passmore, and Paulson [18] formalized an efficient univariate decision procedure based on univariate CAD. There are additionally some univariate decision procedures in PVS, including `hutch` [23] (based on CAD) and `tarski` [24] (based on the Sturm-Tarski theorem).

This paper adds to the latter body of work by formalizing the univariate case of Ben-Or, Kozen, and Reif's (BKR) decision procedure [2] in Isabelle/HOL [26, 27]. Our formalization of univariate BKR is ≈7000 lines [8]. Our main contributions are:

- In Section 2, we present an algorithmic blueprint for implementing BKR's procedure that blends insights from Renegar's [32] later variation of BKR. Compared to the original abstract presentations [2, 32], our blueprint is phrased concretely in terms of matrix operations which facilitates its implementation and identifies its correctness properties.
- Our blueprint is designed for formalization by judiciously combining and fleshing out BKR's and Renegar's proofs. In Section 3, we outline key aspects of our proof, its use of existing Isabelle/HOL libraries, and our contributions to those libraries.

It is desirable to have a variety of formally verified decision procedures for arithmetic since different strategies can have different efficiency tradeoffs on different classes of problems [7, 30]. For example, in PVS, `hutch` is usually significantly faster than `tarski` [23] but there are a number of adversarial problems for `hutch` on which `tarski` performs better [7]. BKR has a fundamentally different working principle than CAD; like the Cohen-Hörmander procedure, it represents roots and sign-invariant regions abstractly, instead of via computationally expensive, real algebraic numbers required in CAD. Further, unlike Cohen-Hörmander, BKR was designed to be used in practice: when its inherent parallelism is exploited, an optimized version of univariate BKR is an NC algorithm (that is, it runs in parallel polylogarithmic time). Our formalization is not yet optimized and parallelized, so we do not yet achieve such efficiency. However, we do export our Isabelle/HOL formalization to Standard ML (SML) and are able to solve some examples with the exported code (Section 3.3).

Additionally, our formalization is a significant stepping stone towards the multivariate case, which builds inductively on the univariate case. We give some (informal) mathematical intuition for multivariate BKR in Appendix A – since multivariate BKR seems to rely fairly directly on the univariate version, we hope that it will be significantly easier to formally verify than multivariate CAD, which is highly complicated. However, it is unlikely that multivariate BKR will be as efficient as CAD in the average case. While BKR states that their multivariate algorithm is computable in parallel exponential time (or in NC for fixed dimension), Canny later found an error in BKR's analysis of the multivariate case [3], which highlights the subtlety of the algorithm and the role for formal verification. Notwithstanding this, multivariate BKR is almost certain to outperform methods such as Tarski's algorithm and Cohen-Hörmander and can supplement an eventual formalization of multivariate CAD.

2 Mathematical Underpinnings

This section provides an outline of our decision procedure for univariate real arithmetic and its verification in Isabelle/HOL [26]. The goal is to provide an accessible mathematical blueprint that explains our construction and its blend of ideas from BKR [2] and Renegar [32]; in-depth technical discussion of the formal proofs is largely deferred to Section 3. Our procedure starts with two transformation steps (Sections 2.1 and 2.2) that simplify an input decision problem into a so-called restricted sign determination format. An algorithm for the latter problem is then presented in Section 2.3. Throughout this paper, unless explicitly specified, we are working with *univariate* polynomials, which we assume to have variable x. Our decision procedure works for polynomials with rational coefficients (`rat poly` in Isabelle), though some lemmas are proved more generally for univariate polynomials with real coefficients (`real poly` in Isabelle).

2.1 From Univariate Problems to Sign Determination

Formulas of *univariate real arithmetic* are generated by the following grammar, where p is a univariate polynomial with rational coefficients:

$$\phi, \psi \ ::= \ p > 0 \mid p \geq 0 \mid p = 0 \mid \phi \vee \psi \mid \phi \wedge \psi$$

In Isabelle/HOL, we define this grammar in `fml`, which is our type for formulas.

For formula ϕ, the *universal* decision problem is to decide if ϕ is true for *all* real values of x, i.e., validity of the quantified formula $\forall x\, \phi$. The *existential* decision problem is to decide if ϕ is true for *some* real value of x, i.e., validity of the quantified formula $\exists x\, \phi$. For example, a decision procedure should return false for formula (1) and true for formula (2) below (left).

$$\forall x\, (x^2 - 2 = 0 \wedge 3x > 0) \qquad (1)$$
$$\exists x\, (x^2 - 2 = 0 \wedge 3x > 0) \qquad (2)$$

Formula Structure: $\textcircled{A} = 0 \wedge \textcircled{B} > 0$

Polynomials: $\textcircled{A} : x^2 - 2, \quad \textcircled{B} : 3x$

The first observation is that both univariate decision problems can be transformed to the problem of finding the set of *consistent sign assignments* (also known as realizable sign assignments [1, Definition 2.34]) of the set of polynomials appearing in the formula ϕ.

▶ **Definition 1.** *A* ***sign assignment*** *for a set G of polynomials is a mapping σ that assigns each $g \in G$ to either $+1$, -1, or 0. A sign assignment σ for G is* ***consistent*** *if there exists an $x \in \mathbb{R}$ where, for all $g \in G$, the sign of $g(x)$ matches the sign of $\sigma(g)$.*

For the polynomials $x^2 - 2$ and $3x$ appearing in formulas (1) and (2), the set of all consistent sign assignments (written as ordered pairs) is:

$$\{(+1,-1),\ (0,-1),\ (-1,-1),\ (-1,0),\ (-1,+1),\ (0,+1),\ (+1,+1)\}$$

Formula (1) is not valid because consistency of sign assignment $(0,-1)$ implies there exists a real value $x \in \mathbb{R}$ such that conjunct $x^2 - 2 = 0$ is satisfied but not $3x > 0$. Conversely, formula (2) is valid because the consistent sign assignment $(0,+1)$ demonstrates the existence of an $x \in \mathbb{R}$ satisfying $x^2 - 2 = 0$ and $3x > 0$. The truth-value of formula ϕ at a given sign assignment is computed by evaluating the formula after replacing all of its polynomials by their respective assigned signs. For example, for the sign assignment $(0,-1)$, replacing Ⓐ by 0 and Ⓑ by -1 in the formula structure underlying (1) and (2) shown above (right) yields $0 = 0 \wedge -1 > 0$, which evaluates to false. Validity of $\forall x\, \phi$ is decided by checking that ϕ evaluates to true at *each* of its consistent sign assignments. Similarly, validity of $\exists x\, \phi$ is decided by checking that ϕ evaluates to true at *at least one* consistent sign assignment.

Our top-level formalized algorithms are called `decide_universal` and `decide_existential`, both with type `rat poly fml ⇒ bool`. The definition of `decide_existential` is as follows (the omitted definition of `decide_universal` is similar):

```
definition decide_existential :: "rat poly fml ⇒ bool"
where "decide_existential fml = (
let (fml_struct,polys) = convert fml in
  find (lookup_sem fml_struct) (find_consistent_signs polys) ≠ None)"
```

Here, `convert` extracts the list of constituent polynomials `polys` from the input formula `fml` along with the formula structure `fml_struct`, `find_consistent_signs` returns the list of all consistent sign assignments `conds` for `polys`, and `find` checks that predicate `lookup_sem fml_struct` is true at one of those sign assignments. Given a sign assignment σ, `lookup_sem fml_struct` σ evaluates the truth value of `fml` at σ by recursively evaluating the truth of its subformulas after replacing polynomials by their sign according to σ using the formula structure `fml_struct`. Thus, `decide_existential` returns true iff `fml` evaluates to true for at least one of the consistent sign assignments of its constituent polynomials.

The correctness theorem for `decide_universal` and `decide_existential` is shown below, where `fml_sem fml x` evaluates the truth of formula `fml` at the real value `x`.

```
theorem decision_procedure:
 "(∀x::real. fml_sem fml x) ⟷ decide_universal fml"
 "(∃x::real. fml_sem fml x) ⟷ decide_existential fml"
```

This theorem depends crucially on `find_consistent_signs` correctly finding *all* consistent sign assignments for `polys`, i.e., solving the sign determination problem.

2.2 From Sign Determination to Restricted Sign Determination

The next step restricts the sign determination problem to the following more concrete format: Find all consistent sign assignments σ for a set of polynomials $q_1, \ldots, q_n$ at the roots of a nonzero polynomial p, i.e., the signs of $q_1(x), \ldots, q_n(x)$ that occur at the (finitely many) real values $x \in \mathbb{R}$ with $p(x) = 0$. The key insight of BKR is that this restricted problem can be solved efficiently (in parallel) using purely algebraic tools (Section 2.3). Following BKR's procedure, we also normalize the q_i's to be coprime with (i.e. share no common factors with) p, which simplifies the subsequent construction for the key step and its formal proof.

▶ Remark 2. The normalization of q_i's to be coprime with p can be avoided using a slightly more intricate construction due to Renegar [32]. We have also formalized this construction but omit full details in this paper as the formalization was completed after acceptance for publication. Its overall structure is quite similar to Section 2.3, and it is available in the AFP alongside our formalization of BKR [8].

Consider as input a set of polynomials (with rational coefficients) $G = \{g_1, \ldots, g_k\}$ for which we need to find all consistent sign assignments. The transformation proceeds as follows:

(1) Factorize the input polynomials G into a set of pairwise coprime factors (with rational coefficients) $Q = \{q_1, \ldots, q_n\}$. This also removes redundant/duplicate polynomials.
Each input polynomial $g \in G$ can be expressed in the form $g = c\prod_{i=1}^{n} q_i^{d_i}$ for some rational coefficient c and natural number exponents $d_i \geq 1$ so the sign of g is directly recovered from the signs of the factors $q \in Q$. For example, if $g_1 = q_1 q_2$ and in a consistent sign assignment q_1 is positive while q_2 is negative, then g_1 is negative according to that assignment, and so on. Accordingly, to determine the set of all consistent sign assignments for G it suffices to determine the same for Q.

(2) Because the q_i's are pairwise coprime, there is no consistent sign assignment where two or more q_i's are set to zero. So, in any given sign assignment, there is either *exactly one* q_i set to zero, or the q_i's are *all assigned to nonzero* (i.e., +1, -1) signs.
Now, for each $1 \leq i \leq n$, solve the restricted sign determination problem for all consistent sign assignments of $\{q_1, \ldots, q_n\} \setminus \{q_i\}$ at the roots of q_i. This yields all consistent sign assignments of Q where exactly one q_i is assigned to zero.

(3) This step and the next step focus on finding all consistent sign assignments where all q_i's are nonzero. Compute a polynomial p that satisfies the following properties:
  i) p is pairwise coprime with all of the q_i's,
  ii) p has a root in every interval between any two roots of the q_i's,
  iii) p has a root that is greater than all of the roots of the q_i's, and
  iv) p has a root that is smaller than all of the roots of the q_i's.
An explicit choice of p satisfying these properties when $q_i \in Q$ are squarefree and pairwise
coprime is the roots of
$q_i \in Q$ is presentative
sample p

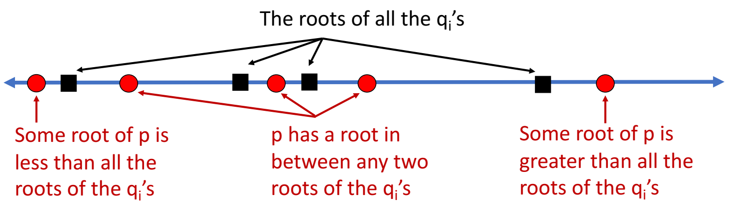

The roots of all the q_i's
Some root of p is less than all the roots of the q_i's
p has a root in between any two roots of the q_i's
Some root of p is greater than all the roots of the q_i's

Figure 1 The relation between the roots of the added polynomial p and the roots of the q_i's.

(4) Solve the restricted sign determination problem for all consistent sign assignments of $\{q_1, \ldots, q_n\}$ at the roots of p.
Returning to Fig. 1, the q_i's are sign-invariant in the intervals between any two roots of the q_i's (black squares) and to the left and right beyond all roots of the q_i's. Intuitively, this is true because moving along the blue real number line in Fig. 1, no q_i can change sign without first passing through a black square. Thus, all consistent sign assignments of q_i that only have nonzero signs must occur in one of these intervals and therefore, by sign-invariance, also at one of the roots of p (red points).

(5) The combined set of sign assignments where some q_i is zero, as found in (2), and where no q_i is zero, as found in (4), solves the sign determination problem for Q, and therefore also for G, as argued in (1).

Our algorithm to solve the restricted sign determination problem using BKR's key insight is called *`find_consistent_signs_at_roots`*; we now turn to the details of this method.

2.3 Restricted Sign Determination

The restricted sign determination problem for polynomials $q_1, \dots, q_n$ at the roots of a polynomial $p \neq 0$, where each $q_1, \dots, q_n$ is coprime with p, can be tackled naively by setting up and solving a *matrix equation.* The idea of using a matrix equation for sign determination dates back to Tarski [36] [1, Section 10.3], and accordingly our formalization shares some similarity to Cohen and Mahboubi's formalization [5] of Tarski's algorithm (see [4, Section 11.2]). BKR's additional insight is to avoid the prohibitive complexity of enumerating exponentially many possible sign assignments for $q_1, \dots, q_n$ by computing the matrix equation recursively and performing a *reduction* that retains only the consistent sign assignments at each recursive step. This reduction keeps intermediate data sizes manageable because the number of consistent sign assignments is bounded by the number of roots of p throughout. We first explain the technical underpinnings of the matrix equation before returning to our implementation of BKR's recursive procedure. *For brevity, references to sign assignments for $q_1, \dots, q_n$ in this section are always at the roots of p.*

2.3.1 Matrix Equation

The inputs to the matrix equation are a set of candidate (i.e., not necessarily consistent) sign assignments $\tilde{\Sigma} = \{\tilde{\sigma}_1, \dots, \tilde{\sigma}_m\}$ for the polynomials $q_1, \dots, q_n$ and a set of subsets $S = \{I_1, \dots, I_l\}$, $I_i \subseteq \{1, \dots, n\}$ of indices selecting among those polynomials. The set of all consistent sign assignments Σ for $q_1, \dots, q_n$ is assumed to be a subset of $\tilde{\Sigma}$, i.e., $\Sigma \subseteq \tilde{\Sigma}$.

For example, consider $p = x^3 - x$ and $q_1 = 3x^3 + 2$. The set of all possible candidate sign assignments $\tilde{\Sigma} = \{(+1), (-1)\}$ must contain the consistent sign assignments for q_1 (sign (0) is impossible as p, q_1 are coprime). The possible subsets of indices are $I_1 = \{\}$ and $I_2 = \{1\}$.

The main algebraic tool underlying the matrix equation is the *Tarski query* which provides semantic information about the number of roots of p with respect to another polynomial q.

▶ **Definition 3.** *Given univariate polynomials p, q with $p \neq 0$, the* Tarski query *$N(p, q)$ is:*

$$N(p, q) = \#\{x \in \mathbb{R} \mid p(x) = 0, q(x) > 0\} - \#\{x \in \mathbb{R} \mid p(x) = 0, q(x) < 0\}.$$

Importantly, the Tarski query $N(p, q)$ can be computed from input polynomials p, q using Euclidean remainder sequences *without* explicitly finding the roots of p. This is a consequence of the Sturm-Tarski theorem which has been formalized in Isabelle/HOL by Li [17]. The theoretical complexity for computing $N(p, q)$ is $O(\deg p\ (\deg p + \deg q))$ [1, Sections 2.2.2 and 8.3]. However, this complexity analysis does not take into account the growth in bitsizes of coefficients in the remainder sequences [1, Section 8.3], so it will not be not achieved by the current Isabelle/HOL formalization of Tarski queries [17] without further optimization.

For the matrix equation, we lift Tarski queries to a *subset* of the input polynomials:

▶ **Definition 4.** *Given a univariate polynomial $p \neq 0$, univariate polynomials $q_1, \dots, q_n$, and a subset $I \subseteq \{1, \dots, n\}$, the* Tarski query *$N(I)$ with respect to p is:*

$$\begin{aligned} N(I) = N(p, \Pi_{i \in I} q_i) = \#\{x \in \mathbb{R} \mid p(x) = 0, \Pi_{i \in I} q_i(x) > 0\} \\ - \#\{x \in \mathbb{R} \mid p(x) = 0, \Pi_{i \in I} q_i(x) < 0\}. \end{aligned}$$

The *matrix equation* is the relationship $M \cdot w = v$ between the following three entities:

- M, the l-by-m matrix with entries $M_{i,j} = \Pi_{k \in I_i} \tilde{\sigma}_j(q_k) \in \{-1, 1\}$ for $I_i \in S$ and $\tilde{\sigma}_j \in \tilde{\Sigma}$,
- w, the length m vector whose entries count the number of roots of p where $q_1, \ldots, q_n$ has sign assignment $\tilde{\sigma}$, i.e., $w_i = \#\{x \in \mathbb{R} \mid p(x) = 0, \operatorname{sgn}(q_j(x)) = \tilde{\sigma}_i(q_j)$ for all $1 \leq j \leq n\}$,
- v, the length l vector consisting of Tarski queries for the subsets, i.e., $v_i = N(I_i)$.

Observe that the vector w is such that the sign assignment $\tilde{\sigma}_i$ is consistent (at a root of p) iff its corresponding entry w_i is nonzero. Thus, the matrix equation can be used to solve the sign determination problem by solving for w. In particular, the matrix M and the vector v are both computable from the input (candidate) sign assignments and subsets. Further, since the subsets will be chosen such that the constructed matrix M is *invertible*, the matrix equation uniquely determines w and the nonzero entries of $w = M^{-1} \cdot v$.

The following Isabelle/HOL theorem summarizes sufficient conditions on the list of sign assignments `signs` and the list of index subsets `subsets` for the matrix equation to hold for polynomial list `qs` at the roots of polynomial `p`. Note the switch from set-based representation to list-based representation in the theorem. This formally provides an ordering to the polynomials, sign assignments, and subsets, which is useful for computations.

```
theorem matrix_equation:
assumes "p≠0"
assumes "⋀q. q ∈ set qs ⟹ coprime p q"
assumes "distinct signs"
assumes "consistent_signs_at_roots p qs ⊆ set signs"
assumes "⋀l i. l ∈ set subsets ⟹ i ∈ set l ⟹ i < length qs"
shows "M_mat signs subsets *ᵥ w_vec p qs signs = v_vec p qs subsets"
```

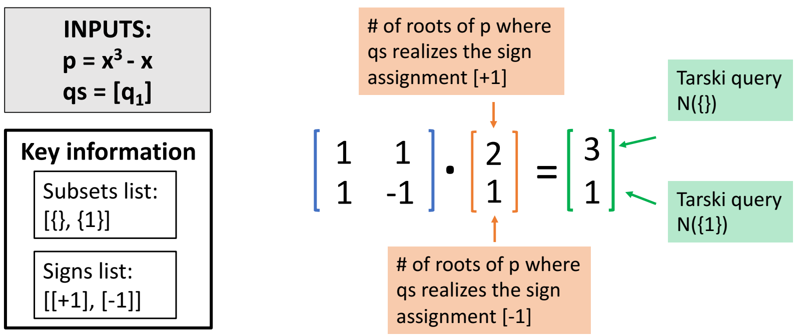
Here, `M_mat`, `w_vec`, and `v_vec` construct the matrix M and vectors w, v respectively; $*_v$ denotes the matrix-vector product in Isabelle/HOL. The switch into list notation necessitates some consistency assumptions, e.g., that the `signs` list contains `distinct` sign assignments and that the index `i` occurring in each list of indices `l` in `subsets` points to a valid element of the list `qs`. The proof of `matrix_equation` uses a counting argument: intuitively, $M_{i,j}$ is the contribution of any real value x that has the sign assignment $\tilde{\sigma}_j$ towards $N(I_i)$, so multiplying these contributions by the actual counts of those real values in w gives $M_i \cdot w = v_i$.

Note that the theorem does *not* ensure that the constructed matrix M is invertible (or even square). This must be ensured separately when solving the matrix equation for w. We now discuss BKR's inductive construction and its usage of the matrix equation.

2.3.2 Base Case

The simplest (base) case of the algorithm is when there is a single polynomial $[q_1]$. Here, it suffices to set up a matrix equation $M \cdot w = v$ from which we can compute all consistent sign assignments. As hinted at earlier, this can be done with the list of index subsets $[\{\}, \{1\}]$ and the candidate sign assignment list $[(+1), (-1)]$.[2] Further, as illustrated in Fig. 2, the matrix M is invertible for these choices of subsets and candidate sign assignments, so the matrix equation can be explicitly solved for w.

[2] In the Isabelle/HOL formalization, we use 0-indexed lists to represent sets and sign assignments, so the subsets list is represented as `[[],[0]]` and the signs list is `[[1],[-1]]`.



$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$$

Figure 2 Matrix equation for $p = x^3 - x$, $q_1 = 3x^3 + 2$.

2.3.3 Inductive Case: Combination Step

The matrix equation can be similarly used to determine the consistent sign assignments for an arbitrary list of polynomials $[q_1, \ldots, q_n]$. The driving idea for BKR is that, given two solutions of the sign determination problem at the roots of p for two input lists of polynomials, say, $\ell_1 = [r_1, \ldots, r_k]$ and $\ell_2 = [r_{k+1}, \ldots, r_{k+l}]$, one can combine them to yield a solution for the list of polynomials $[r_1, \ldots, r_{k+l}]$. This yields a recursive method for solving the sign determination problem by solving the base case at the single polynomials $[q_1], [q_2], \ldots, [q_n]$, and then recursively combining those solutions, i.e., solving $[q_1, q_2], [q_3, q_4], \ldots,$ then $[q_1, q_2, q_3, q_4], \ldots,$ and so on until a solution for $[q_1, \ldots, q_n]$ is obtained. Importantly, BKR performs a reduction (Section 2.3.4) after each combination step to bound the size of the intermediate data.

More precisely, assume for ℓ_1, we have a list of index subsets S_1 and a list of sign assignments $\tilde{\Sigma}_1$ such that $\tilde{\Sigma}_1$ contains all of the consistent sign assignments for ℓ_1 and the matrix M_1 constructed from S_1 and $\tilde{\Sigma}_1$ is invertible. Accordingly, for ℓ_2, we have the list of subsets S_2, list of sign assignments $\tilde{\Sigma}_2$ containing all consistent sign assignments for ℓ_2, and M_2 constructed from S_2, $\tilde{\Sigma}_2$ is invertible. In essence, we are assuming that $S_1, \tilde{\Sigma}_1$ and $S_2, \tilde{\Sigma}_2$ satisfy the hypotheses for the matrix equation to hold, so that they contain all the information needed to solve for the consistent sign assignments of ℓ_1 and ℓ_2 respectively.

Observe that any consistent sign assignment for $\ell = [r_1, \ldots, r_{k+l}]$ must have a prefix that is itself a consistent sign assignment to ℓ_1 and a suffix that is itself a consistent sign assignment to ℓ_2. Thus, the combined list of sign assignments $\tilde{\Sigma}$ obtained by concatenating every entry of $\tilde{\Sigma}_1$ with every entry of $\tilde{\Sigma}_2$ necessarily contains all consistent sign assignments for ℓ. The combined subsets list S is obtained in an analogous way from S_1, S_2 (where concatenation is now set union), with a slight modification: the subset list S_2 indexes polynomials from ℓ_2, but those polynomials now have different indices in ℓ, so everything in S_2 is shifted by the length of ℓ_1 before combination. Once we have the combined subsets list, we can calculate the RHS vector v with Tarski queries as explained in Section 2.3.1.

The matrix M constructed from S, $\tilde{\Sigma}$ is exactly the Kronecker product of M_1 and M_2. Further, the Kronecker product of invertible matrices is invertible, so the matrix equation can be solved for the LHS vector w using M and the vector v computed from the subsets list S. Then the nonzero entries of w correspond to the consistent sign assignments of ℓ. Taking a concrete example, suppose we want to find the list of consistent sign assignments for $\ell = [3x^3 + 2, 2x^2 - 1]$ at the zeros of $p = x^3 - x$. The combination step for $\ell_1 = [3x^3 + 2]$ and $\ell_2 = [2x^2 - 1]$ is visualized in Fig. 3.

p = $x^3 - x$
q_list = [$3x^3 + 2$, $2x^2 - 1$]

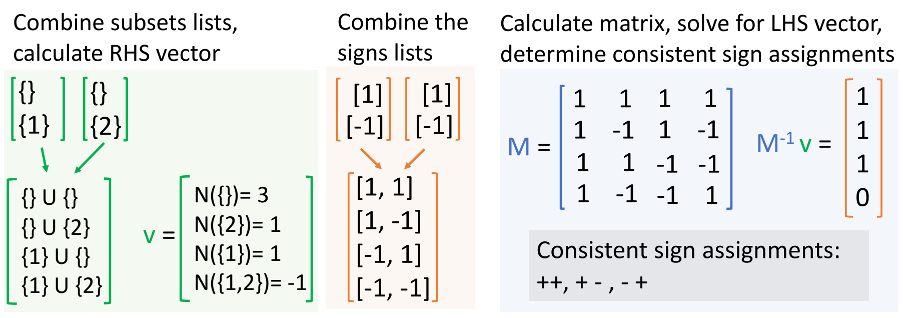


Figure 3 Combining two systems.

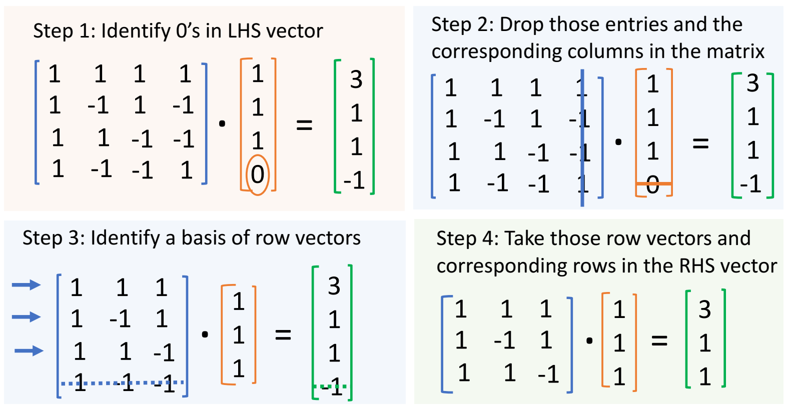

Step 1: Identify 0's in LHS vector
1 1 1 1
1 -1 1 -1
1 1 -1 -1
1 -1 -1 1
1 1 1 0
3 1 1 -1
Step 2: Drop those entries and the corresponding columns in the matrix
1 1 1 1
1 -1 1 -1
1 1 -1 -1
1 -1 -1 1
1 1 1 0
3 1 1 -1
Step 3: Identify a basis of row vectors
1 1 1
1 -1 1
1 1 -1
1 -1 -1
1 1 1
3 1 1 -1
Step 4: Take those row vectors and corresponding rows in the RHS vector
1 1 1
1 -1 1
1 1 -1
1 1 1
3 1 1

Figure 4 Reducing a system.

2.3.4 Reduction Step

The reduction step takes an input list of index subsets S and candidate sign assignments $\tilde{\Sigma}$. It removes the inconsistent sign assignments and then unnecessary index subsets, which keeps the size of the intermediate data tracked for the matrix equation as small as possible.

The reduction step is best explained in terms of the matrix equation $M \cdot w = v$ constructed from the inputs S, $\tilde{\Sigma}$. After solving for w, the reduction starts by deleting all indexes of w_i that are 0 and the corresponding i-th sign assignments in $\tilde{\Sigma}$ which are now known to be inconsistent (recall that w_i counts the number of zeros of p where the i-th sign assignment is realized). This corresponds to deleting the i-th columns of matrix M. If any columns are deleted, the resulting matrix is no longer square (nor invertible). Thus, the next step finds a basis among the remaining rows of the matrix to make it invertible again (deleting any rows that do not belong to the chosen basis). Deleting the j-th row in this matrix corresponds to deleting the j-th index subset in S.

The reduction step for the matrix equation with $p = x^3 - x$ and $\ell = [3x^3 + 2, 2x^2 - 1]$ is visualized in Fig. 4. Naively using the matrix equation for restricted sign determination would require $2^{|\ell|} = 4$ Tarski queries for this example, whereas $2 + 2 + 4 = 8$ queries are required using BKR (2 for each base case, 4 for the combination step). However, for longer lists ℓ, the naive approach requires $2^{|\ell|}$ queries while BKR's reduction step ensures that the number of intermediate consistent sign assignments is bounded by the number of roots of p (and hence $\deg p$) throughout. This difference is shown in Section 3.3 and is also illustrated by Fig. 4, where p has degree 3 and there are 3 consistent sign assignments for ℓ after reduction.

3 Formalization

Now that we have set up the theory behind the BKR algorithm, we turn to some details of our formalization: the proofs, extensions to the existing matrix libraries, and the exported code. Our proof builds significantly on existing proof developments in the Archive of Formal Proofs [17, 38, 39]. Isabelle/HOL's builtin search tool and Sledgehammer [28] provided invaluable automation for discovering existing theorems and for finishing (easy) subgoals in proofs. The most challenging part of the formalization, in our opinion, is the reduction step, in no small part because it involves significant linear algebra (further details in Section 3.2.2).

3.1 Formalizing the Decision Procedure

In this section, we discuss the proofs for our decision procedure in *reverse* order compared to Section 2; that is, we first discuss the formalization of our algorithm for restricted sign determination `find_consistent_signs_at_roots` before discussing the top-level decision procedures for univariate real arithmetic, `decide_{universal|existential}`. The reader may wish to revisit Section 2 for informal intuition behind the procedure while reading this section.

3.1.1 Sign Determination at Roots

We combine BKR's base case (Section 2.3.2), combination step (Section 2.3.3), and reduction step (Section 2.3.4) to form our core algorithm `calc_data` for the restricted sign determination problem at the roots of a polynomial. The `calc_data` algorithm takes a real polynomial `p` and a list of polynomials `qs` and produces a 3-tuple `(M, S, Σ)`, consisting of the matrix `M` from the matrix equation, the list of index subsets `S`, and the list of all consistent sign assignments Σ for `qs` at the roots of `p`. Although `M` can be calculated directly from `S` and Σ, it is returned (as part of the algorithm), to avoid redundantly recomputing it at every recursive call.

```
fun calc_data ::
 "real poly ⇒ real poly list ⇒ (rat mat × (nat list list × rat list list))"
where "calc_data p qs = (let len = length qs in
if len = 0 then
  (λ(a,b,c).(a,b,map (drop 1) c)) (reduce_system p ([1],base_case_info))
else if len ≤ 1 then reduce_system p (qs,base_case_info)
else (let qs1 = take (len div 2) qs; left = calc_data p qs1;
         qs2 = drop (len div 2) qs; right = calc_data p qs2 in
       reduce_system p (combine_systems p (qs1,left) (qs2,right))))"
```

```
definition find_consistent_signs_at_roots ::
 "real poly ⇒ real poly list ⇒ rat list list"
where "find_consistent_signs_at_roots p qs = (let (M,S,Σ) = calc_data p qs in Σ)"
```

The base case where `qs` has length ≤ 1 is handled[3] using the fixed choice of matrix, index subsets, and sign assignments (defined as the constant `base_case_info`) from Section 2.3.2. Otherwise, when `length qs` > 1, the list is partitioned into two sublists `qs1, qs2` and the algorithm recurses on those sublists. The outputs for both sublists are combined using `combine_systems` which takes the Kronecker product of the output matrices and concatenates

[3] The trivial case where `length qs = 0` is also handled for completeness; in this case, the list of consistent sign assignments is empty if p has no real roots, otherwise, it is the singleton list `[[]]`.

the index subsets and sign assignments as explained in Section 2.3.3. Finally, `reduce_system` performs the reduction according to Section 2.3.4, removing inconsistent sign assignments and redundant subsets of indices. The top-level procedure is `find_consistent_signs_at_roots`, which returns only Σ (the third component of `calc_data`). The following Isabelle/HOL snippets show its main correctness theorem and important relevant definitions.

```
definition roots :: "real poly ⇒ real set" where "roots p = {x. poly p x = 0}"

definition consistent_signs_at_roots ::
 "real poly ⇒ real poly list ⇒ rat list set"
where "consistent_signs_at_roots p qs = (sgn_vec qs) ` (roots p)"

theorem find_consistent_signs_at_roots:
assumes "p ≠ 0"
assumes "⋀q. q ∈ set qs ⟹ coprime p q"
shows "set (find_consistent_signs_at_roots p qs) = consistent_signs_at_roots p qs"
```

Here, `roots` defines the set of roots of a polynomial `p` (non-constructively), i.e., real values `x` where the polynomial evaluates to 0 (`poly p x = 0`). Similarly, `consistent_signs_at_roots` returns the set of all sign vectors for the list of polynomials `qs` at the roots of `p`; `sgn_vec` returns the sign vector for input `qs` at a real value and ` is Isabelle/HOL notation for the image of a function on a set. These definitions are not meant to be computational. Rather, they are used to state the correctness theorem that the algorithm `find_consistent_signs_at_roots` (and hence `calc_data`) computes *exactly all* consistent sign assignments for `p` and `qs` for input polynomial `p ≠ 0` and polynomial list `qs`, where every entry in `qs` is `coprime` to `p`.

The proof of `find_consistent_signs_at_roots` is by induction on `calc_data`. Specifically, we prove that the following properties (our inductive invariant) are *satisfied by the base case and maintained by both the combination step and the reduction step*:

1. The signs list is well-defined, i.e., the length of every entry in the signs list is the same as the length of the corresponding `qs`. Additionally, all assumptions on `S` and Σ from the `matrix_equation` theorem from Section 2.3.1 hold. (In particular, the algorithm always maintains a distinct list of sign assignments that, when viewed as a set, is a superset of all consistent sign assignments for `qs`.)
2. The matrix `M` matches the matrix calculated from `S` and Σ. (Since we do not directly compute the matrix from `S` and Σ, as defined in Section 2.3.1, we need to verify that our computations keep track of `M` correctly.)
3. The matrix `M` is invertible (so $M \cdot w = v$ can be uniquely solved for w).

Some of these properties are easier to verify than others. The well-definedness properties, for example, are quite straightforward. In contrast, matrix invertibility is more complicated to verify, especially after the reduction step; we will discuss this in more detail in Section 3.2. The inductive invariant establishes that we have a superset of the consistent sign assignments throughout the construction. This is because the base case and the combination step may include extraneous sign assignments. Only the reduction step is guaranteed to produce exactly the set of consistent sign assignments. Thus the other main ingredient in our formalization, besides the inductive invariant, is a proof that the reduction step deletes all inconsistent sign assignments. As `calc_data` always calls the reduction step before returning output, `calc_data` returns exactly the set of all consistent sign assignments, as desired.

3.1.2 Building the Univariate Decision Procedure

To prove the `decision_procedure` theorem from Section 2.1, we need to establish correctness of `find_consistent_signs`. The most interesting part is formalizing the transformation described in Section 2.2. We discuss the steps from Section 2.2 enumerated (1)–(5) below.

(1) Our procedure takes an input list of rational polynomials $G = [g_1, \ldots, g_k]$ and computes a list of their pairwise coprime and squarefree factors[4] $Q = [q_1, \ldots, q_n]$. An efficient method to factor a *single* rational polynomial is formalized in Isabelle/HOL by Divasón et al. [9]; we slightly modified their proof to find factors for a *list* of polynomials while ensuring that the resulting factors are pairwise coprime, which implies that their product $\prod_i q_i$ is squarefree.

(2) This step makes n calls to `find_consistent_signs_at_roots`, one for each $Q \setminus \{q_i\}$.

(3) We choose the polynomial $p = (x - \texttt{crb}(\prod_i q_i))(x + \texttt{crb}(\prod_i q_i))(\prod_i q_i)'$, where $(\prod_i q_i)'$ is the formal polynomial derivative of $\prod_i q_i$ and $\texttt{crb}(\prod_i q_i)$ is a computable positive integer with larger magnitude than any real root of $\prod_i q_i$. The choice of $\texttt{crb}(\prod_i q_i)$ uses a proof of the Cauchy root bound [1, Section 10.1] by Thiemann and Yamada [39]. We prove that p satisfies the four properties of step (3) from Section 2.2:

- **i)** Since $\prod_i q_i$ is squarefree, $(\prod_i q_i)'$ is coprime with $\prod_i q_i$ and, thus, also coprime with each of the q_i's. Because $\texttt{crb}(\prod_i q_i)$ is strictly larger in magnitude than all of the roots of the roots of the q_i's, it follows that p is also coprime with all of the q_i's.
- **ii)** By Rolle's theorem[5] (which is already formalized in Isabelle/HOL's standard library), $(\prod_i q_i)'$ has a root between every two roots of $\prod_i q_i$ and therefore `p` also has a root in every interval between any two roots of the q_i's.
- **iii)** and *iv)* This choice of p has roots at $-\texttt{crb}(\prod_i q_i)$ and $\texttt{crb}(\prod_i q_i)$, which are respectively smaller and greater than all roots of the q_i's.

(4) Each polynomial q_i is sign invariant between its roots.[6] Accordingly, the q_i's are sign invariant between the roots of $\prod_i q_i$ (and to the left/right of all roots of the q_i's).

(5) We use the `find_consistent_signs_at_roots` algorithm with Q and our chosen p.

Putting the pieces together, we verify that `find_consistent_signs` finds exactly the consistent sign assignments for its input polynomials. The `decision_procedure` theorem follows by induction over the `fml` type representing formulas of univariate real arithmetic and our formalized semantics for those formulas.

3.2 Matrix Library

Matrices feature prominently in our algorithm: the combination step uses the Kronecker product, while the reduction step requires matrix inversion and an algorithm for finding a basis from the rows (or, equivalently, columns) of a matrix. There are a number of linear algebra libraries available in Isabelle/HOL [10, 34, 38], each building on a different underlying representation of matrices. We use the formalization by Thiemann and Yamada [38] as it provides most of the matrix algorithms required by our decision procedure and supports efficient code extraction [38, Section 1]. Naturally, any such choice leads to tradeoffs; we now detail some challenges of working with the library and some new results we prove.

3.2.1 Combination Step: Kronecker Product

We define the Kronecker product for matrices `A`, `B` over a `ring` as follows:

[4] This is actually overkill: we do not necessarily need to *completely* factor every polynomial in G to transform G into a set of pairwise coprime factors. BKR suggest a parallel algorithm based in part on the literature [40] to find a "basis set" of squarefree and pairwise coprime polynomials.

[5] For differentiable function $f : \mathbb{R} \mapsto \mathbb{R}$ with $f(a) = f(b)$, $a < b$, there exists $a < z < b$ where $f'(z) = 0$.

[6] By the intermediate value theorem (which is already formalized in Isabelle/HOL's standard library), if q_i changes sign, e.g., from positive to negative, between two adjacent roots, then there exists a third root in between those adjacent roots, which is a contradiction.

```
definition kronecker_product :: "'a :: ring mat ⇒ 'a mat ⇒ 'a mat"
where "kronecker_product A B = (
let ra = dim_row A; ca = dim_col A; rb = dim_row B; cb = dim_col B in
  mat (ra * rb) (ca * cb)
    (λ(i,j). A $$ (i div rb, j div cb) * B $$ (i mod rb, j mod cb)))"
```

Matrices with entries of type `'a` are constructed with `mat m n f`, where `m, n :: nat` are the number of rows and columns of the matrix respectively, and `f :: nat × nat ⇒ 'a` is such that `f i j` gives the matrix entry at position `i, j`. Accordingly, `M $$ (i,j)` extracts the `(i,j)`-th entry of matrix `M`, and `dim_row, dim_col` return the number of rows and columns of a matrix respectively.

We prove basic properties of our definition of the Kronecker product: it is associative, distributes over addition, and satisfies the mixed-product identity for matrices `A`, `B`, `C`, `D` with compatible dimensions (for `A * C` and `B * D`): `kronecker_product (A * C) (B * D) = (kronecker_product A B) * (kronecker_product C D)`. The mixed-product identity implies that the Kronecker product of invertible matrices is invertible. Briefly, for invertible matrices `A`, `B` with respective inverses A^{-1}, B^{-1}, the mixed product identity gives: `(kronecker_product A B) * (kronecker_product` A^{-1} B^{-1}`) = kronecker_product (A *` A^{-1}`) (B *` B^{-1}`) = I` where `I` is the identity matrix. In other words, `kronecker_product A B` and `kronecker_product` A^{-1} B^{-1} are inverses. We use this to prove that the matrix obtained by the combination step is invertible (part of the inductive hypothesis from Section 3.1.1).

▶ Remark 5. Prathamesh [31] formalized Kronecker products for Isabelle/HOL's default matrix type. For computational purposes, we provide a new formalization that is compatible with the matrix representation of Thiemann and Yamada [38].

3.2.2 Reduction Step: Gauss–Jordan and Matrix Rank

Our reduction step makes extensive use of the Gauss–Jordan elimination algorithm by Thiemann and Yamada [37]. First, we use matrix inversion based on Gauss–Jordan elimination to invert the matrix M in the matrix equation (Section 2.3.1 and Step 1 in Fig. 4). We also contribute new proofs surrounding their Gauss–Jordan elimination algorithm in order to use it to extract a basis from the rows (equivalently columns) of a matrix (Step 3 in Fig. 4).

Suppose that an input matrix `A` has more rows than columns, e.g., the matrix in Step 2 of Fig. 4. The following definition of `rows_to_keep` returns a list of (distinct) row indices of `A`.

```
definition rows_to_keep:: "('a::field) mat ⇒ nat list"
where "rows_to_keep A = map snd (pivot_positions (gauss_jordan_single (A^T)))"
```

Here, `gauss_jordan_single` returns the row-reduced echelon form (RREF) of `A` after Gauss–Jordan elimination and `pivot_positions` finds the positions, i.e., `(row, col)` pairs, of the first nonzero entry in each row of the matrix; both are existing definitions from the library by Thiemann and Yamada [37]. Our main new result for `rows_to_keep` is:

```
lemma rows_to_keep_rank:
assumes "dim_col A ≤ dim_row A"
shows "vec_space.rank (length (rows_to_keep A)) (take_rows A (rows_to_keep A)) =
      vec_space.rank (dim_row A) A"
```

Here `vec_space.rank n M` (defined by Bentkamp [38]) is the finite dimension of the vector space spanned by the columns of `M`. Thus, the lemma says that keeping only the pivot rows of matrix `A` (with `take_rows A (rows_to_keep A)`) preserves the rank of `A`. At a high level, the proof of `rows_to_keep_rank` is in three steps:

1. First, we prove a version of *rows_to_keep_rank* for the pivot *columns* of a matrix and where *A* is assumed to be a matrix in RREF. The RREF assumption for *A* enables direct analysis of the shape of its pivot columns.
2. Next, we lift the result to an arbitrary matrix *A*, which can always be put into RREF form by *gauss_jordan_single*.
3. Finally, we formalize the following classical result that column rank is equal to row rank: *vec_space.rank (dim_row A) A = vec_space.rank (dim_col A) (A^T)*. We lift the preceding results for pivot columns to also work for pivot rows by matrix transposition (pivot rows of matrix *A* are the pivot columns of the transpose matrix A^T).

To complete the proof of the reduction step, recall that the matrix in Step 2 of Fig. 4 is obtained by dropping columns of an invertible matrix. The resulting matrix has full column rank but more rows than columns. We show that when *A* in *rows_to_keep_rank* has full column rank (its *rank* is *dim_col A*) then *length (rows_to_keep A) = dim_col A* and so the matrix consisting of pivot rows of *A* is square, has full rank, and is therefore invertible.

▶ Remark 6. Divasón and Aransay formalized the equivalence of row and column rank for Isabelle/HOL's default matrix type [11] while we have formalized the same result for Bentkamp's definition of matrix rank [38]. Another technical drawback of our choice of libraries is the locale argument *n* for *vec_space*. Intuitively (for real matrices) this carves out subsets of $\mathbb{R}^n$ to form the vector space spanned by the columns of *M*. Whereas one would usually work with *n* fixed and implicit within an Isabelle/HOL locale, we pass the argument explicitly here because our theorems often need to relate the rank of vector spaces in $\mathbb{R}^m$ and $\mathbb{R}^n$ for $m \neq n$. This negates some of the automation benefits of Isabelle/HOL's locale system.

3.3 Code Export

We export our decision procedure to Standard ML, compile with `mlton`, and test it on 10 microbenchmarks from [18, Section 8]. While we leave extensive experiments for future work since our implementation is unoptimized, we compare the performance of our procedure using BKR sign determination (Sections 2.3.2–2.3.4) versus an *unverified* implementation that naively uses the matrix equation (Section 2.3.1). We also ran Li *et al.*'s `univ_rcf` decision procedure [18] which can be directly executed as a proof tactic in Isabelle/HOL (code kindly provided by Wenda Li). The benchmarks were ran on an Ubuntu 18.04 laptop with 16GB RAM and 2.70 GHz Intel Core i7-6820HQ CPU. Results are in Table 1.

The most significant bottleneck in our current implementation is the computation of Tarski queries $N(p, q)$ when solving the matrix equation. Recall for our algorithm (Section 2.3.1) the input q to $N(p, q)$ is a *product* of (subsets of) polynomials appearing in the inputs. Indeed, Table 1 shows that the algorithm performs well when the factors have low degrees, e.g., ex1, ex2, ex4, and ex5. Conversely, it performs poorly on problems with many factors and higher degrees, e.g., ex3, ex6, and ex7. Further, as noted in experiments by Li and Paulson [20], the Sturm-Tarski theorem in Isabelle/HOL currently uses a straightforward method for computing remainder sequences which can also lead to significant (exponential) blowup in the bitsizes of rational coefficients of the involved polynomials. This is especially apparent for ex6 and ex7, which have large polynomial degrees and high coefficient complexity; these time out without completing even a single Tarski query. From Table 1, the BKR approach successfully reduces the number of Tarski queries as the number of input factors grows – the number of queries for BKR is dependent on the polynomial degrees and the number of consistent sign assignments, while the naive approach always requires exactly $(\frac{n}{2} + 1)2^n$

Table 1 Comparison of decision procedures using naive and BKR sign determination and Li *et al.*'s `univ_rcf` tactic in Isabelle/HOL [18]. All formulas are labeled following [18, Section 8]; formulas with $\wedge$ indicate conjunctions of the listed examples. Columns: **#Poly** counts the number of distinct polynomials appearing in the formula (maximum degree among polynomials in parentheses), **#Factor** counts the number of distinct factors from (1) in Section 2.2 (maximum degree among factors in parentheses), **#$N(p,q)$** counts the number of Tarski queries made by each approach, and **Time** reports time taken (seconds, 3 d.p.) for each decision procedure to run to completion. Cells with - indicate a timeout after 1 hour.

Formula	#Poly	#Factor	#$N(p,q)$ (Naive)	#$N(p,q)$ (BKR)	Time (Naive)	Time (BKR)	Time ([18])
ex1	4 (12)	3 (1)	20	31	0.003	0.006	3.020
ex2	5 (6)	7 (1)	576	180	5.780	0.442	3.407
ex3	4 (22)	5 (22)	112	120	1794.843	1865.313	3.580
ex4	5 (3)	5 (2)	112	95	0.461	0.261	3.828
ex5	8 (3)	7 (3)	576	219	28.608	8.333	3.806
ex6	22 (9)	22 (8)	50331648	-	-	-	6.187
ex7	10 (12)	10 (11)	6144	-	-	-	-
ex1 $\wedge$ 2	9 (12)	9 (1)	2816	298	317.432	3.027	3.033
ex1 $\wedge$ 2 $\wedge$ 4	13 (12)	12 (2)	28672	555	-	51.347	3.848
ex1 $\wedge$ 2 $\wedge$ 5	16 (12)	14 (3)	131072	826	-	436.575	3.711

queries for n factors[7] (which are reported in Table 1 whether completed or not). On the other hand, there is some overhead for smaller problems, e.g., ex1, ex3, that arises from the recursion in BKR.

The `univ_rcf` tactic relies on an external solver (we used Mathematica 12.1.1) to produce *untrusted* certificates which are then formally checked (by reflection) in Isabelle/HOL [18]. This procedure is optimized and efficient: except for ex7 where the tactic timed out, most of the time (roughly 3 seconds per example) is actually spent to start an instance of the external solver.

An important future step, e.g., to enable use of our procedure as a tactic in Isabelle/HOL, is to avoid coefficient growth by using pseudo-division [24, Section 3] or more advanced techniques: for example, using subresultants to compute polynomial GCDs (and thereby build the remainder sequences) [12]. Pseudo-division is also important in the multivariate generalization of BKR (discussed in Appendix A), where the polynomial coefficients of concern are themselves (multivariate) polynomials rather than rational numbers. The pseudo-division method has been formalized in Isabelle/HOL [18], but it is not yet available on the AFP.

4 Related Work

Our work fits into the larger body of formalized univariate decision procedures. Most closely related are Li *et al.*'s formalization of a CAD-based univariate QE procedure in Isabelle/HOL [18] and the *`tarski`* univariate QE strategy formalized in PVS [24]. We discuss each in turn.

[7] For n factors, Section 2.2's transformation yields n restricted sign determination subproblems involving $n-1$ polynomials each and one subproblem involving n polynomials. Using naive sign determination to solve all of these subproblems requires $n(2^{n-1}) + 2^n = (\frac{n}{2}+1)2^n$ Tarski queries in total.

The univariate CAD algorithm underlying Li *et al.*'s approach [18] decomposes $\mathbb{R}$ into a set of sign-invariant regions, so that every polynomial of interest has constant sign within each region. A real algebraic sample point is chosen from every region, so the set of sample points captures all of the relevant information about the signs of the polynomials of interest *for the entirety of* $\mathbb{R}$. BKR (and Renegar) take a more indirect approach, relying on consistent sign assignments which merely indicate the *existence* of points with such signs. Consequently, although CAD will be faster in the average case, BKR and CAD have different strengths and weaknesses. For example, CAD works best on full-dimensional decision problems [21], where only rational sample points are needed (this allows faster computation than the computationally expensive real algebraic numbers that general CAD depends on). The Sturm-Tarski theorem is also invoked in Li *et al*'s procedure to decide the sign of a univariate polynomial at a point (using only rational arithmetic) [18, Section 5]. (This was later extended to bivariate polynomials by Li and Paulson [19].) This is theoretically similar to our procedure to find the consistent sign assignments for $q_1, \ldots, q_n$ at the roots of p, as both rely on the mathematical properties of Tarski queries; however, for example, we do not require isolating the real roots of p within intervals, whereas such isolation predicates their computations. This difference reflects our different goals: theirs is to encode algebraic numbers in Isabelle/HOL, ours is to perform full sign determination with BKR.

PVS's `tarski` uses Tarski queries and a version of the matrix equation to solve univariate decision problems [24]. Unlike our work, `tarski` has already been optimized in significant ways; for example, `tarski` computes Tarski queries with pseudo-divisions. However, `tarski` does not maintain a *reduced* matrix equation as our work does. Further, `tarski` was designed to solve existential conjunctive formulas, requiring DNF transformations otherwise [7].

In addition, as previously mentioned, our work is somewhat similar in flavor to Cohen and Mahboubi's (multivariate) formalization of Tarski's algorithm [5]. In particular, the characterization of the matrix equation and the parts of the construction that do not involve reduction share considerable overlap, as BKR derives the idea of the matrix equation from Tarski [2]. However, the reduction step is only present in BKR and is a distinguishing feature of our work.

5 Conclusion and Future Work

This paper describes how we have verified the correctness of a decision procedure for univariate real arithmetic in Isabelle/HOL. To the best of our knowledge, this is the first formalization of BKR's key insight [2, 32] for recursively exploiting the matrix equation. Our formalization lays the groundwork for several future directions, including:

1. Optimizing the current formalization and adding parallelism.
2. Proving that the univariate sign determination problem is decidable in NC [2, 32] and other complexity-theoretic results. This (ambitious) project would require developing a complexity framework that is compatible with all of the libraries we use.
3. Verifying a multivariate sign determination algorithm and decision procedure based on BKR. As mentioned previously, multivariate BKR has an error in its complexity analysis; variants of decision procedures for $\text{FOL}_{\mathbb{R}}$ based on BKR's insight that attempt to mitigate this error could eventually be formalized for useful points of comparison. Two of particular interest are that of Renegar [32], who develops a full QE algorithm, and that of Canny [3], in which coefficients can involve some more general terms, like transcendental functions.

References

1 Saugata Basu, Richard Pollack, and Marie-Françoise Roy. *Algorithms in Real Algebraic Geometry*. Springer, Berlin, Heidelberg, second edition, 2006. doi:10.1007/3-540-33099-2.

2 Michael Ben-Or, Dexter Kozen, and John H. Reif. The complexity of elementary algebra and geometry. *J. Comput. Syst. Sci.*, 32(2):251–264, 1986. doi:10.1016/0022-0000(86)90029-2.

3 John F. Canny. Improved algorithms for sign determination and existential quantifier elimination. *Comput. J.*, 36(5):409–418, 1993. doi:10.1093/comjnl/36.5.409.

4 Cyril Cohen. *Formalized algebraic numbers: construction and first-order theory*. PhD thesis, École polytechnique, November 2012. URL: https://perso.crans.org/cohen/papers/thesis.pdf.

5 Cyril Cohen and Assia Mahboubi. Formal proofs in real algebraic geometry: from ordered fields to quantifier elimination. *Log. Methods Comput. Sci.*, 8(1), 2012. doi:10.2168/LMCS-8(1:2)2012.

6 George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In H. Barkhage, editor, *Automata Theory and Formal Languages*, volume 33 of *LNCS*, pages 134–183. Springer, 1975. doi:10.1007/3-540-07407-4_17.

7 Katherine Cordwell, César Muñoz, and Aaron Dutle. Improving automated strategies for univariate quantifier elimination. Technical Memorandum NASA/TM-20205010644, NASA, Langley Research Center, Hampton VA 23681-2199, USA, January 2021. URL: https://ntrs.nasa.gov/citations/20205010644.

8 Katherine Cordwell, Yong Kiam Tan, and André Platzer. The BKR decision procedure for univariate real arithmetic. *Archive of Formal Proofs*, April 2021. Formal proof development. URL: https://www.isa-afp.org/entries/BenOr_Kozen_Reif.html.

9 Jose Divasón, Sebastiaan J. C. Joosten, René Thiemann, and Akihisa Yamada. A formalization of the Berlekamp-Zassenhaus factorization algorithm. In Yves Bertot and Viktor Vafeiadis, editors, *CPP*, pages 17–29. ACM, 2017. doi:10.1145/3018610.3018617.

10 Jose Divasón and Jesús Aransay. Rank-nullity theorem in linear algebra. *Archive of Formal Proofs*, January 2013. Formal proof development. URL: https://isa-afp.org/entries/Rank_Nullity_Theorem.html.

11 Jose Divasón and Jesús Aransay. Gauss-Jordan algorithm and its applications. *Archive of Formal Proofs*, September 2014. Formal proof development. URL: https://isa-afp.org/entries/Gauss_Jordan.html.

12 Lionel Ducos. Optimizations of the subresultant algorithm. *J. Pure Appl. Algebra*, 145(2):149–163, 2000. doi:10.1016/S0022-4049(98)00081-4.

13 Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Hoang Le Truong, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, et al. A formal proof of the Kepler conjecture. *Forum of Mathematics, Pi*, 5, 2017. doi:10.1017/fmp.2017.1.

14 John Harrison. Floating-point verification using theorem proving. In Marco Bernardo and Alessandro Cimatti, editors, *SFM*, volume 3965 of *LNCS*, pages 211–242. Springer, 2006. doi:10.1007/11757283_8.

15 John Harrison. Verifying nonlinear real formulas via sums of squares. In Klaus Schneider and Jens Brandt, editors, *TPHOLs*, volume 4732 of *LNCS*, pages 102–118. Springer, 2007. doi:10.1007/978-3-540-74591-4_9.

16 Johannes Hölzl. Proving inequalities over reals with computation in Isabelle/HOL. In Gabriel Dos Reis and Laurent Théry, editors, *PLMMS*, pages 38–45, Munich, August 2009.

17 Wenda Li. The Sturm-Tarski theorem. *Archive of Formal Proofs*, September 2014. Formal proof development. URL: https://isa-afp.org/entries/Sturm_Tarski.html.

18 Wenda Li, Grant Olney Passmore, and Lawrence C. Paulson. Deciding univariate polynomial problems using untrusted certificates in Isabelle/HOL. *J. Autom. Reason.*, 62(1):69–91, 2019. doi:10.1007/s10817-017-9424-6.

19 Wenda Li and Lawrence C. Paulson. A modular, efficient formalisation of real algebraic numbers. In Jeremy Avigad and Adam Chlipala, editors, *CPP*, pages 66–75. ACM, 2016. `doi:10.1145/2854065.2854074`.

20 Wenda Li and Lawrence C. Paulson. Counting polynomial roots in Isabelle/HOL: a formal proof of the Budan-Fourier theorem. In Assia Mahboubi and Magnus O. Myreen, editors, *CPP*, pages 52–64. ACM, 2019. `doi:10.1145/3293880.3294092`.

21 Scott McCallum. Solving polynomial strict inequalities using cylindrical algebraic decomposition. *Comput. J.*, 36(5):432–438, 1993. `doi:10.1093/comjnl/36.5.432`.

22 Sean McLaughlin and John Harrison. A proof-producing decision procedure for real arithmetic. In Robert Nieuwenhuis, editor, *CADE*, volume 3632 of *LNCS*, pages 295–314. Springer, 2005. `doi:10.1007/11532231_22`.

23 César A. Muñoz, Anthony J. Narkawicz, and Aaron Dutle. A decision procedure for univariate polynomial systems based on root counting and interval subdivision. *J. Formaliz. Reason.*, 11(1):19–41, 2018. `doi:10.6092/issn.1972-5787/8212`.

24 Anthony Narkawicz, César A. Muñoz, and Aaron Dutle. Formally-verified decision procedures for univariate polynomial computation based on Sturm's and Tarski's theorems. *J. Autom. Reason.*, 54(4):285–326, 2015. `doi:10.1007/s10817-015-9320-x`.

25 Tobias Nipkow. Linear quantifier elimination. *J. Autom. Reason.*, 45(2):189–212, 2010. `doi:10.1007/s10817-010-9183-0`.

26 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. `doi:10.1007/3-540-45949-9`.

27 Lawrence C. Paulson. The foundation of a generic theorem prover. *J. Autom. Reason.*, 5(3):363–397, 1989. `doi:10.1007/BF00248324`.

28 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*, volume 2 of *EPiC Series in Computing*, pages 1–11. EasyChair, 2010. URL: `https://easychair.org/publications/paper/wV`.

29 André Platzer. *Logical Foundations of Cyber-Physical Systems*. Springer, Cham, 2018. `doi:10.1007/978-3-319-63588-0`.

30 André Platzer, Jan-David Quesel, and Philipp Rümmer. Real world verification. In Renate A. Schmidt, editor, *CADE*, volume 5663 of *LNCS*, pages 485–501. Springer, 2009. `doi:10.1007/978-3-642-02959-2_35`.

31 T.V.H. Prathamesh. Tensor product of matrices. *Archive of Formal Proofs*, January 2016. Formal proof development. URL: `https://isa-afp.org/entries/Matrix_Tensor.html`.

32 James Renegar. On the computational complexity and geometry of the first-order theory of the reals, part III: Quantifier elimination. *J. Symb. Comput.*, 13(3):329–352, 1992. `doi:10.1016/S0747-7171(10)80005-7`.

33 Alexey Solovyev. *Formal Computations and Methods*. PhD thesis, University of Pittsburgh, January 2013. URL: `https://d-scholarship.pitt.edu/16721/`.

34 Christian Sternagel and René Thiemann. Executable matrix operations on matrices of arbitrary dimensions. *Archive of Formal Proofs*, June 2010. Formal proof development. URL: `https://isa-afp.org/entries/Matrix.html`.

35 Thomas Sturm. A survey of some methods for real quantifier elimination, decision, and satisfiability and their applications. *Math. Comput. Sci.*, 11(3-4):483–502, 2017. `doi:10.1007/s11786-017-0319-z`.

36 Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry*. RAND Corporation, Santa Monica, CA, 1951. Prepared for publication with the assistance of J.C.C. McKinsey. URL: `https://www.rand.org/pubs/reports/R109.html`.

37 René Thiemann and Akihisa Yamada. Formalizing Jordan normal forms in Isabelle/HOL. In Jeremy Avigad and Adam Chlipala, editors, *CPP*, pages 88–99. ACM, 2016. `doi:10.1145/2854065.2854073`.

38 René Thiemann and Akihisa Yamada. Matrices, Jordan normal forms, and spectral radius theory. *Archive of Formal Proofs*, August 2015. Formal proof development. URL: `https://isa-afp.org/entries/Jordan_Normal_Form.html`.

39 René Thiemann, Akihisa Yamada, and Sebastiaan Joosten. Algebraic numbers in Isabelle/HOL. *Archive of Formal Proofs*, December 2015. Formal proof development. URL: `https://isa-afp.org/entries/Algebraic_Numbers.html`.

40 Joachim von zur Gathen. Parallel algorithms for algebraic problems. *SIAM J. Comput.*, 13(4):802–824, 1984. `doi:10.1137/0213050`.

A Comments on Multivariate BKR

The ultimate intent is for the univariate formalization to serve as the basis for an extension to the multivariate case. The main part of the univariate construction that must be adapted for multivariate polynomials is the computation of Tarski queries. In the univariate case, this is accomplished with remainder sequences per the following (standard) result:

▶ **Theorem 7** (Generalized Sturm's theorem [32, Proposition 8.1]). *Given coprime univariate polynomials p, q with $p \neq 0$, form the Euclidean remainder sequence $p_1 = p$, $p_2 = p'q$, and p_i is the negated remainder of p_{i-2} divided by p_{i-1} for $i \geq 3$. This terminates at some $p_{k+1} = 0$ because the remainder has lower degree than the divisor at every step. Let a_i be the leading coefficient of p_i for $1 \leq i \leq k$. Consider the two sequences $a_1, \dots, a_k$ and $(-1)^{\deg p_1} a_1, \cdots, (-1)^{\deg p_k} a_k$. If $S^+(p,q)$ is the number of sign changes in $a_1, \dots, a_k$ and $S^-(p,q)$ is the number of sign changes in $(-1)^{\deg p_1} a_1, \cdots, (-1)^{\deg p_k} a_k$, then $N(p,q) = S^-(p,q) - S^+(p,q)$.*

Following the idea of BKR, we intend to treat multivariate polynomials in n variables as univariate polynomials (whose coefficients are polynomials in $n-1$ variables) and so compute remainder sequences of polynomials with attention to a single variable. These remainder sequences will be sequences of polynomials in $n-1$ variables rather than integers, but we only need to know the *signs* of those polynomials (rather than their values). That reduces the problem of sign determination for polynomials in n variables to a sign determination problem for polynomials in $n-1$ variables. In this way we intend to successively reduce multivariate computations to a series of (already formalized) univariate computations.

This intuition can be captured by the following concrete example. Consider $p = x^2y + 1$ and $q = xy + 1$. Suppose we choose to first eliminate y. If x is 0, then the analysis for the remaining $p = q = 1$ is simple. Otherwise, both x and x^2 are nonzero. Now, we calculate the remainder sequence from Theorem 7: $p_1 = x^2y + 1$, $p_2 = x^3y + x^2$, and $p_3 = -(1-x)$. To find p_3, we calculate $x^2y + 1 = \frac{1}{x}(x^3y + x^2) + (1-x)$, where $\frac{1}{x}$ is well-defined since $x \neq 0$.

The leading coefficients of p_1, p_2, and p_3 as polynomials in y are $a_1 = x^2$, $a_2 = x^3$, and $a_3 = -(1-x)$. Here, we must use our univariate algorithm to fix some consistent sign assignment in x on the a_i's, taking into account our earlier stipulation that x and x^2 are nonzero. Say that we choose, for example, x positive, x^3 positive, and $-(1-x)$ negative. (A full QE procedure would need to consider *all* possible consistent sign assignments.) Because of our chosen sign assignment, a_1 is positive, a_2 is positive, and a_3 is negative. Still following Theorem 7, $S^+(p,q) = 1$ and $S^-(p,q) = 0$. The Tarski query $N(\{1\})$ is then computed as $N(\{1\}) = N(p,q) = S^-(p,q) - S^+(p,q) = -1$.

If we wish to find the signs of q at the roots of p, we can use this way of computing Tarski queries to build the matrix equation for p and q. Computing $N(\{\}) = 1$, and following the method of the base case (in which the candidate signs list is $[[+1], [-1]]$), we find:

$$\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

Looking at the LHS vector, we see that only its second entry is nonzero. This means that -1 is the only consistent sign assignment for q at the zeros of p, *given our assumptions that x is positive and $-(1-x)$ is negative.*

We can check this as follows: Given our assumption that $x \neq 0$, the only root of p is $-\frac{1}{x^2}$. Plugging this into q, we obtain $x(-\frac{1}{x^2}) + 1 = -\frac{1}{x} + 1$. Because x is assumed to be positive, the sign of $-\frac{1}{x} + 1$ is the same as the sign of $x(-\frac{1}{x} + 1) = -1 + x = -(1-x)$, which we have assumed to be negative.

Thus, -1 is a consistent sign assignment for q at the roots of p. To find the other consistent sign assignments, we repeat this process with all other consistent choices for the signs of x and a_1, a_2, a_3.

Formalising a Turing-Complete Choreographic Language in Coq

Luís Cruz-Filipe
Department Mathematics and Computer Science, University of Southern Denmark, Odense, Denmark

Fabrizio Montesi
Department Mathematics and Computer Science, University of Southern Denmark, Odense, Denmark

Marco Peressotti
Department Mathematics and Computer Science, University of Southern Denmark, Odense, Denmark

Abstract

The theory of choreographic languages typically includes a number of complex results that are proved by structural induction. The high number of cases and the subtle details in some of them lead to long reviewing processes, and occasionally to errors being found in published proofs. In this work, we take a published proof of Turing completeness of a choreographic language and formalise it in Coq. Our development includes formalising the choreographic language, its basic properties, Kleene's theory of partial recursive functions, the encoding of these functions as choreographies, and a proof that this encoding is correct.

With this effort, we show that theorem proving can be a very useful tool in the field of choreographic languages: besides the added degree of confidence that we get from a mechanised proof, the formalisation process led us to a significant simplification of the underlying theory. Our results offer a foundation for the future formal development of choreographic languages.

2012 ACM Subject Classification Theory of computation → Process calculi; Theory of computation → Recursive functions; Theory of computation → Logic and verification

Keywords and phrases Choreographic Programming, Formalisation, Turing Completeness

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.15

Related Version *Previous Version*: `https://arxiv.org/abs/2102.02627`

Supplementary Material *Software*: `https://doi.org/10.5281/zenodo.4479102`

Funding Work partially supported by Villum Fonden, grant no. 29518.

1 Introduction

Background. In the setting of concurrent and distributed systems, choreographic languages are used to define interaction protocols that communicating processes should abide to [20, 30, 33]. These languages are akin to the "Alice and Bob" notation found in security protocols, and inherit the key idea of making movement of data manifest in programs [29]. This is usually obtained through a linguistic primitive like `Alice.e` → `Bob.x`, read "`Alice` communicates the result of evaluating expression `e` to `Bob`, which stores it in its local variable `x`".

In recent years, the communities of concurrency theory and programming languages have been prolific in developing methodologies based on choreographies, yielding results in program verification, monitoring, and program synthesis [1, 19]. For example, in *multiparty session types*, types are choreographies used for checking statically that a system of processes implements protocols correctly [18]. Further, in *choreographic programming*, choreographic languages are elevated to full-fledged programming languages [27], which can express how data should be pre- and post-processed by processes (encryption, validation, anonymisation, etc.). Choreographic programming languages showed promise in a number of contexts, including parallel algorithms [6], cyber-physical systems [25, 24, 16], self-adaptive systems [11], system integration [15], information flow [22], and the implementation of security protocols [16].

The problem. Proofs in the field of choreographic languages are extremely technical. They have to cover many cases, and they typically involve translations from/to other languages that come with their own structures and semantics. The level of complexity makes peer-reviewing challenging. For example, it has recently been discovered that a significant number (at least 5) of key results published in peer-reviewed articles on multiparty session types actually do not hold, and that their statements require modification [31].

This article. The aim of this article is to show that computer-aided verification – in particular, interactive theorem proving – can be successfully applied to the study of choreographies and to provide solid foundations for future developments. Before presenting our scientific contributions, it is interesting to look at the story behind this article, as it tells us that interactive theorem proving is not just a tool to check what we already know.

Our development started in late 2018. Its starting point was the theory of Core Choreographies (CC), a minimalistic language that we previously proposed for the study of choreographic programming [7]. CC is designed to include only the essential features of choreographic languages and minimal computational capabilities at processes (computing the successor of a natural number and deciding equality of two natural numbers). Nevertheless, it is expressive enough to be Turing complete, which is proven by developing a provably-correct translation of Kleene's partial recursive functions [21] into choreographies that implement the source functions by means of communication [7].

At the TYPES conference in 2019, we gave an informal progress report on the formalisation of CC using the Coq theorem prover [8]. Our effort revealed soon a crux of unparsimonious complexity in the theory: a set of term-rewriting rules for a precongruence relation used in the semantics of the language for (i) expanding procedure calls with the bodies of their respective procedures and (ii) reshuffling independent communications in choreographies to represent correctly concurrent execution. This relation is closed under context and transitivity, and it can always be involved in the derivation of reductions, which led to tedious induction on the derivation of these term rewritings in almost all cases of proofs that had to do with the semantics of choreographies. In addition to being time consuming, formalising this aspect makes the theory presented in [7] much more complicated (we discuss this in Section 2.4).

At the time, the second author had been teaching a few editions of a course that includes theory of choreographies. Interestingly, the same technical aspects that made the formalisation of CC much more intricate than its original theory were all found to be subtly complicated by students. Motivated by this observation and our early efforts in formalising [7], this author developed a revisited theory of CC for his course material that dispenses with the problematic notions and shows that they are actually *unnecessary* [28]. The choreography theory in [28] is the one that we deal with in this article.

Thus, besides its scientific contributions, this article also shows that theorem proving can be used in research: the insights that we got doing this formalisation led to changes in the original theory. We show that this did not come at the cost of expressive power: the translation of Kleene's partial recursive functions from [7] still works as-is for the theory in [28]. Furthermore, while formalising the theory we realised that some assumptions in some results were actually not necessary, yielding stronger results.

Contributions. This article presents the first formalised theory of a full-fledged choreographic language, including its syntax and semantics, and the main properties of determinism, confluence, and deadlock-freedom by design. This theory is formalised in Coq, using its module system to make it parametric. We show that the choreographic language is Turing complete, by encoding Kleene's partial recursive functions as choreographies and proving this encoding sound. The full development can be downloaded at [10].

Structure. For compactness we state our definitions and results in Coq syntax. In Section 2, we present the syntax and semantics of our choreographic language, based on its formalisation in Coq, and establish the main theoretical properties of this language. Section 3 presents the theory and formalisation of Kleene's partial recursive functions, and Section 4 describes their encoding as choreographies and the proof of Turing completeness of the choreographic language. We discuss the relevance of our results and future work in Section 5.

2 Choreographies

In this section, we introduce the choreographic language of Core Choreographies (CC), together with the corresponding formalisation.

A choreography specifies a protocol involving different participants (processes) that can communicate among themselves and possess local computational capabilities. Each process also has storage, which is accessible through variables. There are two kinds of communications: value communications, where the sender process locally evaluates an expression and sends the result to the receiver process, who stores it in one of its variables; and label selection, where the sender selects one among some alternative behaviours (tagged by labels) offered by the receiver. A choreography can also define (recursive) procedures, which can be invoked by their name anywhere. The formal syntax of choreographies is given in Section 2.2.

2.1 Preliminaries

We define the type of choreographies as a parametric Coq `Module`, taking eight parameters: the types of process identifiers (processes for short) `Pid`, local variables `Var` (used to access the processes' storage), values `Val`, expressions `Expr`, Boolean expressions `BExpr`, procedure names `RecVar` (from *recursion variables*), and the evaluation functions mapping expressions to values and Boolean expressions to Booleans.

The first six parameters are datatypes that are equipped with a decidable equality. Due to difficulties with using the definitions in Coq's standard libraries, we reimplemented this as a `Module Type DecType`, and defined a functor `DecidableType` providing the usual lemmas to simplify function definitions by case analysis on equality of two objects.

Evaluation requires a *local state*, mapping process variables to actual values. We model states as functors, taking `Var` and `Val` as parameters and returning a `Module` containing this function type together with an operator to update the state (by changing the value assigned to one variable) and lemmas characterising it. For the semantics of choreographies, we also need *global states*, which take `Pid` as an additional parameter and map each process to a local state. This type is again enriched with operations to update a global state.

Both modules include a definition of extensional equality: for two local states `s` and `s'`, `eq_state s s'` holds iff ∀ `x`, `s x` = `s' x`, and for global states `eq_state_ext s s'` holds iff ∀ `p`, `eq_state (s p) (s' p)`.

An evaluation function is a function mapping expressions to values, given a local state. Evaluation must be compatible with extensional equality on states.

```
Module Type Eval (Expression Vars Input Output : DecType).

Parameter eval : Expression.t → (Vars.t → Input.t) → Output.t.
Parameter eval_wd : ∀ f f', (∀  x, f x = f' x) → ∀ e, eval e f = eval e f'.
```

This module is instantiated twice in the choreography type: with arguments `Expr`, `Var`, `Val` and `Val`, evaluating expressions to values, and with arguments `BExpr`, `Var`, `Val` and `bool`, evaluating Boolean expressions to Booleans.

2.2 Syntax

We present the Coq definition of the type of choreographies, and afterwards briefly explain each constructor and its pretty-printing.

```
Inductive Label : Type :=
 | left : Label
 | right : Label.

Inductive Eta : Type :=
 | Com : Pid → Expr → Pid → Var → Eta
 | Sel : Pid → Pid → Label → Eta.

Inductive Choreography : Type :=
 | Interaction : Eta → Choreography → Choreography
 | Cond : Pid → BExpr → Choreography → Choreography → Choreography
 | Call : RecVar → Choreography
 | RT_Call : RecVar → (list Pid) → Choreography → Choreography
 | End : Choreography.

Definition DefSet := RecVar → (list Pid)*Choreography.

Record Program : Type :=
  { Procedures : DefSet;
    Main : Choreography }.
```

Constructor `Interaction` builds a choreography that starts with a communication (`Eta`), which can be either a value communication (`Com`) or a label selection (`Sel`). These choreographies are written as `p#e⟶ q$x;;C` or `p⟶ q[l];;C`, respectively (in general, `eta;; C` is a choreography whenever `eta:Eta`). Label selections were inherited by choreographies from linear logic and behavioural types: they are used to communicate a choice made by the sender to the receiver. In minimalistic theories of choreographies and behavioural types, it is common to restrict the set of labels that can be communicated (`Label`) to two elements, generically called `left` and `right` [7, 3]. These labels are typically used to propagate information about the local evaluation of a conditional expression, which generates two possible execution branches.

A choreography that starts by locally evaluating an expression is built using `Cond`, written `If p??b Then C1 Else C2`, while invoking procedure `X` is built as `Call X`. A procedure may involve several processes; the auxiliary term `RT_Call X ps C` represents a procedure that has already started executing, but the processes in `ps` have not yet entered it – the term `C` is obtained from the actions of procedure that have been executed (see the semantics below). We remark that `ps` has type `list Pid`, but it is interpreted as a set (in particular, it is always manipulated by means of set operations). `End` denotes the terminated choreography.

A DefSet (set of procedure definitions) is a mapping assigning to each RecVar a list of processes and a choreography; intuitively, the list of processes contains the processes that are used in the procedure. A Program is a pair containing a set of procedure definitions and the choreography to be executed at the start.

Terms built using RT_Call are meant to be runtime terms, generated while executing a choreography; therefore, programs written by programmers should not contain such terms. We call such a choreography initial, and define it inductively in the natural way.

Well-formedness. There are a number of well-formedness requirements on choreographies. Some of these come from practical motivations and are typically explicitly required, while others are more technical and not always written down in other articles. We summarise these conditions.

A choreography is well-formed if its processes do not self-communicate: the two arguments of type Pid to Com and Sel in all its subterms are always distinct. Furthermore, the list of process names in the argument of RT_Call is never empty. These conditions are defined separately by recursion in the natural way.

For a program P to be well-formed, there are more requirements on procedure definitions and the annotations of the runtime terms. First, both Main P and all choreographies in Procedures P must be well-formed, and furthermore the latter must all be initial choreographies. In Main P, all runtime terms must be consistently annotated: the set ps in RT_Call X ps C should only contain processes that occur in the definition of X (as specified in Procedures P).

Second, a program must be finitely specifiable. Instead of requiring the type RecVar to be finite, which would significantly complicate the formalisation, we require that there exist Xs:list RecVar such that every procedure in Xs, as well as Main P, only calls procedures in Xs. Hence, it becomes irrelevant what the remaining procedure definitions are. Well-formedness is thus parameterised on Xs. (Vars P X and Procs P X denote the first and second components of Procedures P X, respectively, and within_Xs Xs C holds if choreography C only uses procedure names in Xs.)

```
Definition Program_WF (Xs:list RecVar) (P:Program) : Prop :=
  Choreography_WF (Main P) ∧ within_Xs Xs (Main P) ∧ consistent (Vars P) (Main P) ∧
  ∀ X,  In X Xs → Choreography_WF (Procs P X) ∧ initial (Procs P X) ∧
            (Vars P X) ≠ nil ∧ within_Xs Xs (Procs P X).
```

The third and last condition is that the set of procedure definitions in a program must be well-annotated: if Procedures P X=(ps,C), then the set of processes used in C must be included in ps. The set of processes used in C is in turn recursively defined using the information in Procedures P, so computing an annotation is not straightforward. For this reason, it can be convenient in practice to over-annotate a program – which is why well-annotation only requires a set inclusion. (Function CCC_pn computes the set of processes occurring in a choreography, given the set of processes used in each procedure.)

```
Definition well_ann (P:Program) : Prop :=
  ∀ X,  set_incl _ (CCC_pn (Procs P X) (Vars P)) (Vars P X).

Definition CCP_WF (P:Program) := well_ann P ∧ ∃ Xs, Program_WF Xs P.
```

While Program_WF is decidable, well_ann and CCP_WF (for CC program) are not, due to the quantifications in their definitions. This motivates defining Program_WF separately.

Formalising well-formedness requires some auxiliary definitions (the sets of processes and procedure names used in a choreography) and several inductive definitions. Most of them are straightforward, if sometimes cumbersome; the complexity of the final definition can make proofs of well-formedness quickly grow in size and number of cases, so we provide a number of inversion results such as the following to make subsequent proofs easier.

```
Lemma CCP_WF_eta : ∀ Defs C eta,
  CCP_WF (Build_Program Defs (eta;;C)) → CCP_WF (Build_Program Defs C).
```

2.3 Semantics

The semantics of CC is defined by means of labelled transition systems, in several layers. At the lowest layer, we define the transitions that a choreography can make (`CCC_To`), parameterised on a set of procedure definitions; then we pack these transitions into the more usual presentation – as a labelled relation `CCP_To` on *configurations* (pairs program/state). Finally, we define multi-step transitions `CCP_ToStar` as the transitive and reflexive closure of the transition relation. This layered approach makes proofs about transitions cleaner, since the different levels of induction are separated.

Transition labels. We have two types of transition labels. The first one is a simple inductive type with constructors corresponding to the possible actions a choreography can take: value communications, label selections, local conditional, or local procedure call. This type is called `RichLabel`: rich labels are not present in the informal theory [28], but they are needed to obtain strong enough induction hypotheses in proofs of results about CC that we will need in Section 4 for Turing completeness. The labels in the informal theory correspond to observable actions; they are formalised as `TransitionLabel`, and they forget the internal details of actions. The two types are connected by a function `forget:RichLabel → TransitionLabel`.

```
Inductive RichLabel : Type :=
| R_Com (p:Pid) (v:Value) (q:Pid) (x:Var) : RichLabel
| R_Sel (p:Pid) (q:Pid) (l:Label) : RichLabel
| R_Cond (p:Pid) : RichLabel
| R_Call (X:RecVar) (p:Pid) : RichLabel.

Inductive TransitionLabel : Type :=
| L_Com (p:Pid) (v:Value) (q:Pid) : TransitionLabel
| L_Sel (p:Pid) (q:Pid) (l:Label) : TransitionLabel
| L_Tau (p:Pid) : TransitionLabel.
```

Transition relations. `CCC_To` is defined inductively by a total of 11 clauses, corresponding to the 11 rules in the informal presentation. We include some of them below, with some proof terms omitted.

```
Inductive CCC_To (Defs : DefSet) :
  Choreography → State → RichLabel → Choreography → State → Prop :=
 | C_Com p e q x C s s' : let v := (eval_on_state e s p) in
      eq_state_ext s' (update s q x v) →
      CCC_To Defs (p #e ⟶ q$x;; C) s (R_Com p v q x) C s'
 | C_Delay_Eta eta C C' s s' t: disjoint_eta_rl eta t →
      CCC_To Defs C s t C' s' →
      CCC_To Defs (eta;; C) s t (eta;; C') s'
```

```
| C_Call_Start p X s s':
       eq_state_ext s s' →
       set_size _ (fst (Defs X)) > 1 → In p (fst (Defs X)) →
       CCC_To Defs
              (Call X) s
              (R_Call X p)
              (RT_Call X (set_remove _ p (fst (Defs X))) (snd (Defs X))) s'
| C_Call_Enter p ps X C s s':
       eq_state_ext s s' → set_size _ ps > 1 → In p ps →
       CCC_To Defs
              (RT_Call X ps C) s
              (R_Call X p)
              (RT_Call X (set_remove _ p ps) C) s'
| C_Call_Finish p ps X C s s':
       eq_state_ext s s' → set_size _ ps = 1 → In p ps →
       CCC_To Defs
              (RT_Call X ps C) s (R_Call X p) C s'.
```

The first constructor defines a transition for a value communication, with the state being updated with the value received at the receiver. Since states are functions, we do not want to require that the resulting state be `update s q x v` – the state obtained directly from `s` by updating the value at `q`'s variable `x` with `v` – but only that it be extensionally equal to it (the values of variables at all processes are the same). (The stronger requirement would break, e.g., confluence, since updating two different variables in a different order does not yield the same state.)

The original informal theory allows for out-of-order execution of independent interactions [7, 28], a well-established feature of choreographic languages [5, 18]. For example, given a choreography that consists of two independent communications such as `p#e⟶ q$x;;r#e'⟶ s$y;;End` ("`p` communicates `e` to `q` and `r` communicates `e'` to `s`") where `p`, `q`, `r`, and `s` are distinct processes, we should be able to observe the first and the second value communication in whichever order. Out-of-order execution is modelled by three rules, of which the second constructor shown is an example. Here, a choreography is allowed to reduce under a prefix `eta` if its label does not share any processes with `eta`. This side-condition is checked by `disjoint_eta_rl eta t`; several auxiliary predicates named `disjoint_type1_type2` are defined to simplify writing these conditions.

Procedure calls are managed by four rules, of which the main three are shown. A procedure call is expanded when the first process involved in it enters it (rule `C_Call_Start`). The remaining processes and the procedure's definition are stored in a runtime term, from which we can observe transitions either by more processes entering the procedure (rule `C_Call_Enter`) or by out-of-order execution of internal transitions of the procedure (rule `C_Delay_Call`, not shown). When the last process enters the procedure, the runtime term is consumed (rule `C_Call_Finish`). The missing rule addresses the edge case when a procedure only uses one process.

In order to prove results about transitions, it is often useful to infer the resulting choreography and state. The constructors of `CCC_To` cannot be used for this purpose, since the resulting state is not uniquely determined. Therefore, we prove a number of lemmas stating restricted forms of transitions that are useful for forward reasoning.

```
Lemma C_Com' : ∀ Defs p e q x C s, let v := (eval_on_state e s p) in
       CCC_To Defs (p #e ⟶ q$x;; C) s (R_Com p v q x) C (update s q x v).
```

Afterwards, we formalise the transition relations as defined in [28].

```
Definition Configuration : Type := Program *State.

Inductive CCP_To : Configuration → TransitionLabel → Configuration → Prop :=
 | CCP_To_intro Defs C s t C' s' : CCC_To Defs C s t C' s' →
     CCP_To (Build_Program Defs C,s) (forget t) (Build_Program Defs C',s').

Inductive CCP_ToStar : Configuration → list TransitionLabel → Configuration → Prop :=
 | CCT_Refl c : CCP_ToStar c nil c
 | CCT_Step c1 t c2 l c3 : CCP_To c1 t c2 → CCP_ToStar c2 l c3 → CCP_ToStar c1 (t::l) c3.
```

We also define the suggestive notations `c —[tl ]⟶ c'` for `CCP_To c tl c'` and `c —[ts ]⟶ *c'` for `CCP_To_Star c ts c'`.

2.4 Progress, Determinism, and Confluence

The challenging – and interesting – part of formalising CC is establishing the basic properties of the language, which are essential for more advanced results and typically not proven in detail in publications. We discuss some of the issues encountered, as these were the driving force behind the changes relative to [7].

The first key property of choreographies is that they are deadlock-free by design: any choreography that is not terminated can reduce. Since the only terminated choreography in CC is `End`, this property also implies that any choreography either eventually reaches the terminated choreography `End` or runs infinitely. These properties depend on the basic result that transitions preserve well-formedness.

```
Lemma CCC_ToStar_CCP_WF : ∀ P s l P' s', CCP_WF P → (P,s) —[l]⟶ * (P',s') → CCP_WF P'.

Theorem progress : ∀ P, Main P ≠ End → CCP_WF P → ∀ s, ∃ tl c', (P,s) —[tl]⟶  c'.

Theorem deadlock_freedom : ∀ P, CCP_WF P →
  ∀ s ts c', (P,s) —[ts]⟶ * c' → {Main (fst c') = End} + {∃ tl c'', c' —[tl]⟶  c''}.
```

This is the first place where we benefit from the change in both the syntax and semantics of choreographies from [7] to [28], which removes idiosyncrasies that required clarifications in the reviewing process of [7]. In the language of [7], procedures are defined inside choreographies by means of a def $X = C_X$ in C constructor in choreographies. This makes the definition of terminated choreography much more complicated, since `End` could occur inside some of these terms. Furthermore, procedure calls were expanded by structural precongruence, so that a choreography as def X = End in X would also be terminated. Separating procedure definitions from the main choreography in a program and promoting procedure calls to transitions makes stating and proving progress much simpler. The direct syntactic characterisation of termination also has advantages, since it is intuitive and easily verifiable.

The second key property is confluence, which is an essential ingredient of the proof of Turing completeness below: if a choreography has two different transition paths, then these paths either end at the same configuration, or both resulting configurations can reach the same one. This is proved by first showing the diamond property for choreography transitions, then lifting it to one-step transitions, and finally applying induction.

```
Lemma diamond_Chor : ∀ Defs C s tl1 tl2 C1 C2 s1 s2,
  CCC_To Defs C s tl1 C1 s1 → CCC_To Defs C s tl2 C2 s2 →
  tl1 ≠ tl2 → ∃ C' s', CCC_To Defs C1 s1 tl2 C' s' ∧ CCC_To Defs C2 s2 tl1 C' s'.
```

```
Lemma diamond_1 : ∀ c tl1 tl2 c1 c2,
  c —[tl1 ]⟶ c1 → c —[tl2 ]⟶ c2 →
  tl1 ≠ tl2 → ∃ c', c1 —[tl2 ]⟶ c' ∧ c2 —[tl1 ]⟶ c'.

Lemma diamond_4 : ∀ P s tl1 tl2 P1 s1 P2 s2,
  (P,s) —[tl1 ]⟶ *(P1,s1) → (P,s) —[tl2 ]⟶ *(P2,s2) →
  (∃ P' tl1' tl2' s1' s2',
    (P1,s1) —[tl1' ]⟶ *(P',s1') ∧ (P2,s2) —[tl2' ]⟶ *(P',s2') ∧ eq_state_ext s1' s2').
```

As an important consequence, we get that any two executions of a choreography that end in a terminated choreography must yield the same state.

```
Lemma termination_unique : ∀ c tl1 c1 tl2 c2,
  c —[tl1]⟶ * c1 → c —[tl2]⟶ * c2 →
  Main (fst c1) = End → Main (fst c2) = End → eq_state_ext (snd c1) (snd c2).
```

The complexity of the proof of confluence was the determining factor for deciding to start our work from the variation of the choreographic language presented in [28] instead of that in [7]. The current proof of confluence takes about 300 lines of Coq code, including a total of 11 lemmas. This is in stark contrast with the previous attempt, which already included over 30 lemmas with extremely long proofs. The reason for this complexity lay, again, in both inlined procedure definitions and structural precongruence. Inlined procedure definitions forced us to deal with all the usual problems regarding bound variables and renaming (e.g. dealing with capture-avoiding substitutions, working up to α-renaming); structural precongruence introduced an absurd level of complexity because it allowed choreographies to be rewritten arbitrarily.

To understand this issue, consider again the choreography `p#e⟶ q$x;; r#e'⟶ s$y;; End`. As we have previously discussed, this choreography can execute first either the communication between `p` and `q` or the one between `r` and `s`. In our framework, the first transition is modelled by rule `C_Com`, while the second is obtained by applying rule `C_Delay_Eta` followed by `C_Com`. In a framework with reductions and structural precongruence, instead, the second transition is modelled by first rewriting the choreography as `r#e'⟶ s$y;; p#e⟶ q$x;; End` and then applying rule `C_Com` [7]. The set of legal rewritings is formally defined by the structural precongruence relation $\preceq$, and there is a rule in the semantics allowing C_1 to reduce to C_2 whenever $C_1 \preceq C_1'$, $C_2' \preceq C_2$, and C_1' reduces to C_2. Thus, the proof of confluence also needs to take into account all the possible ways into which choreographies may be rewritten in a reduction. In a proof of confluence, where there are two reductions, there are four possible places where choreographies are rewritten; given the high number of rules defining structural precongruence, this led to an explosion of the number of cases. Furthermore, induction hypotheses typically were not strong enough, and we were forced to resort to complicated auxiliary notions such as explicitly measuring the size of the derivation of transitions, and proving that transitions could be normalised. This process led to a seemingly ever-growing number of auxiliary lemmas that needed to be proved, and after several months of work with little progress it became evident that the problem lay in the formalism.

With the current definitions, the theory of CC is formalised in two files. The first file, which defines the preliminaries, contains 40 definitions, 58 lemmas and around 700 lines of code. The second file, which defines CC-specific results, contains 48 definitions, 106 lemmas, 2 theorems and around 2100 lines of code.

3 Partial Recursive Functions

In order to formalise Turing completeness of our choreographic language, we need a model of computation. In [7], the model chosen was Kleene's partial recursive functions [21], and the proof proceeds by showing that these can all be implemented as a choreography, for a suitable definition of implementation. This proof structure closely follows that of the original proof of computational completeness for Turing machines [32].

In this section, we describe our formalisation of partial recursive functions, and the main challenges and design options that it involved. Following standard pratice, we routinely use lambda notation for denoting these functions.

3.1 Syntax

The class of partial recursive functions is defined inductively as the smallest class containing the constant unary zero function $Z = \lambda x.0$, the unary successor function $S = \lambda x.x + 1$ and the n-ary projection functions $P^n_k = \lambda x_1 \dots x_n.x_k$ (base functions), and closed under the operations of composition, primitive recursion, and minimisation. All functions have an arity (natural number); the arity of Z and S is 1, and the arity of P^n_k is n. Given a function g of arity m and m functions $f_1, \dots, f_m$ of arity k, then the composition $C(g, \vec{f})$ has arity k; if g has arity k and h has arity $k + 2$, then function $R(g, h)$ defined by primitive recursion from g and h has arity $k + 1$; and if h has arity $k + 1$, then its minimisation $M(h)$ has arity k.

We formalise this class as a dependent inductive type `PRFunction` taking the arity of the function as a parameter. In order to ensure the correct number of arguments in composition, we require $f_1, \dots, f_m$ to be given as a vector of length m (the type of vectors of length m whose elements have type `A` is written in Coq as `t A m`).

```
Inductive PRFunction : nat → Set :=
  | Zero : PRFunction 1
  | Successor : PRFunction 1
  | Projection : ∀ {m k:nat}, k < m → PRFunction m
  | Composition : ∀ {k m:nat} (g:PRFunction m) (fs:t (PRFunction k) m), PRFunction k
  | Recursion : ∀ {k:nat} (g:PRFunction k) (h:PRFunction (2+k)), PRFunction (1+k)
  | Minimisation : ∀ {k:nat} (h:PRFunction (1+k)), PRFunction k.
```

Note the required proof term on the constructor for projections. The parameter `k` is one unit lower than the parameter k in the mathematical definition, since Coq's natural numbers start at 0 – this choice simplifies the development.

3.2 Semantics

A partial recursive function of arity m is meant to denote a partial function of type $\mathbb{N}^m \to \mathbb{N}$. The denotation of Z, S and P^n_k was already given above; the remaining operators are interpreted as follows, where we write $\vec{x}$ for $x_1, \dots, x_k$.

$$\begin{aligned} C(g, \vec{f})(\vec{x}) &= g(f_1(\vec{x}), \dots, f_m(\vec{x})) \\ R(g, h)(0, \vec{x}) &= g(\vec{x}) \\ R(g, h)(n + 1, \vec{x}) &= h(n, R(g, h)(n, \vec{x}), \vec{x}) \\ M(h)(\vec{x}) &= n \text{ if } h(\vec{x}, n) = 0 \text{ and } h(\vec{x}, i) > 0 \text{ for all } 0 \le i < n \end{aligned}$$

Minimisation can lead to partiality, since there may be no n satisfying the conditions given in its definition. This partiality propagates, since any value depending on an undefined value is also undefined. Kleene's original work does not completely specify this mechanism:

for example, if f is a unary function that is undefined everywhere, should $C(Z, f)$ be also everywhere undefined, or constantly zero? It is common practice to follow a call-by-value semantics and assume that a function is undefined whenever any of its arguments is undefined, even in the case where those arguments are not used for computing the result; we take this approach in this work.

Since Coq does not allow for defining partial functions, we take an operational approach to the semantics of partial recursive functions, and interactively define the bounded evaluation of an m-ary function f on a vector of length m in n steps, which has type `option nat`. This construction proceeds in three steps. First, we deal with minimisation by defining

```
Fixpoint find_zero_from {k} (h:t (option nat) (1+k) → option nat)
  (ns:t (option nat) k) (init:nat) (steps:nat) : option nat :=
  match steps with
  | 0 ⇒ None
  | S m ⇒ match h (shiftin (Some init) ns) with
          | None ⇒ None
          | Some 0 ⇒ Some init
          | Some (S _) ⇒ find_zero_from h ns (S init) m end end.
```

which tries to find the smallest zero of `h` starting at `init` using a bound of `steps` steps for the first value, `steps`−1 for the next value, etc. (The type `t T n` is the type of vectors containing exactly `n` elements of type `T`.) With this, we recursively define the evaluation function

```
Fixpoint eval_opt {m} (f:PRFunction m) : ∀ (steps:nat) (ns:t (option nat) m), option nat.
```

Defining this function directly is complex due to the dependent type `PRFunction m` – one needs to do induction on `m` and then reason about the possible cases for `f` in each case, which is not easy to write directly. Instead, we define `eval_opt` directly. Finally, we define

```
Definition eval {m} (f:PRFunction m) (steps:nat) (ns:t nat m) : option nat
  := eval_opt f steps (map Some ns)
```

as our intended evaluation function.

Evaluation starts by checking that all arguments are defined. If this is the case, then the base functions always return their value; composition and recursion call the functions that they depend upon with the same number of steps; and minimisation initiates a search from 0 with the bounds explained above.

In order to ensure that the interactive definitions are correct, we prove a number of lemmas stating that the defining equations for each class of functions hold. For example, for recursion we have the following three lemmas.

```
Lemma Recursion_correct_base : ∀ k (g:PRFunction k) (h:PRFunction (2+k)) (ns:t nat (1+k)),
  ∀ steps, hd ns = 0 → eval (Recursion g h) steps ns = eval g steps (tl ns).

Lemma Recursion_correct_step : ∀ k (g:PRFunction k) (h:PRFunction (2+k)) (ns:t nat (1+k)),
  ∀ steps x y, hd ns = S x → (eval (Recursion g h) steps (x :: tl ns)) = Some y →
  eval (Recursion g h) steps ns = eval h steps (x :: y :: tl ns).

Lemma Recursion_correct_step' : ∀ k (g:PRFunction k) (h:PRFunction (2+k)) (ns:t nat (1+k)),
  ∀ steps x, hd ns = S x → (eval (Recursion g h) steps (x :: tl ns)) = None →
  eval (Recursion g h) steps ns = None.
```

These results rely on a number of auxiliary results, notably about the function `find_zero_from`.

3.3 Examples

For further proof of correctness, we chose some functions that typically are used as examples in textbooks on the topic – addition, multiplication, sign – and some relations – greater than, smaller than, equals – and showed that the usual definitions are correct. For example, sum is defined as $R(P^1_1, C(S, P^3_2))$ (this is also used as an example in [7]). Our formalisation defines `PR_add` as `Recursion (Projection aux11) (Composition Successor [Projection aux23])`, and includes

```
Lemma add_correct : ∀ m n steps, eval PR_add steps [m; n] = Some (m + n).
```

(The parameters m and k are implicit in the constructor for projections; the proof terms are named to correspond to the informal usage, so that `aux11:0<1`. Thus P^1_1 is represented by `@Projection 0 1 aux11`.)

3.4 Convergence and Uniqueness

The next step is to show that the value by `eval` is unique and stable (augmenting the number of steps can only change it from `None` to `Some n`, and not conversely).

This is the first place where we have to do induction over `PRFunction`. The induction principle automatically generated by Coq from the type definition is not strong enough for our purposes: the constructor for composition includes elements of type `PRFunction` inside a vector argument, and these functions are not available on inductive proofs. We use a standard technique to overcome this limitation: we assign a depth to every element of `PRFunction` (corresponding to the depth of its abstract syntax tree), and prove results by induction over the depth of functions. In particular, this allows us to prove the following general induction principle.

```
Theorem PRFunction_induction : ∀ (P:∀ (n:nat) (f:PRFunction n), Prop),
  P _ Zero → P _ Successor →
  (∀ i j (Hp:i<j), P _ (Projection Hp)) →
  (∀ m k g fs, (∀ H, P m fs[@H]) → P k g → P _ (Composition g fs)) →
  (∀ k g h, P _ g → P _ h → P (1+k) (Recursion g h)) →
  (∀ k h, P _ h → P k (Minimisation h)) →
  ∀ n f, P n f.
```

(The notation `v[@H]` denotes the k-th element of vector `v`, where `H` is a proof that k is smaller than the length of `v`.)

Using this principle (and sometimes directly induction over depth), we can prove all mentioned properties of evaluation. We then define convergence and divergence in the natural way.

```
Definition converges {k} (f:PRFunction k) ns y := ∃ steps, eval f steps ns = Some y.
Definition diverges {k} (f:PRFunction k) ns := ∀ steps, eval f steps ns = None.
Lemma converges_inj : ∀ {k} f ns y y', converges (k:=k) f ns y → converges f ns y' → y = y'.
Lemma converges_diverges : ∀ {k} f ns, (diverges (k:=k) f ns ↔ ∀ y, ~converges f ns y).
```

Finally, we prove a number of results for establishing convergence of each class of functions. These results are used later, when proving Turing completeness of choreographies. For example, for recursion we have:

```
Lemma Recursion_converges_base : ∀ k g h ns y,
  converges g (tl ns) y → converges (@Recursion k g h) (0::tl ns) y.
```

```
Lemma Recursion_converges_step : ∀ k g h ns x y z,
  converges (@Recursion k g h) (x::ns) y →
  converges h (x::y:: ns) z → converges (Recursion g h) (S x::ns) z.
```

Conversely, if recursion converges, then all intermediate computations must also converge.

```
Lemma converges_Recursion_base : ∀ {m} (g:PRFunction m) h ns y,
  converges (Recursion g h) ns y → hd ns = 0 → converges g (tl ns) y.

Lemma converges_Recursion_step : ∀ {m} (g:PRFunction m) h ns x y,
  converges (Recursion g h) ns y → hd ns = (S x) →
  ∃ z, converges (Recursion g h) (x :: tl ns) z ∧ converges h (x :: z :: tl ns) y.

Lemma converges_Recursion_full : ∀ {m} (g:PRFunction m) h ns y,
  converges (Recursion g h) ns y →
  ∀ x, x ≤ hd ns → ∃ z, converges (Recursion g h) (x :: tl ns) z.
```

For completeness, the formalisation also includes corresponding results for divergence; these are currently unused.

This part of the development contains 22 definitions and 84 lemmas, with a total of 1388 lines of code. (This excludes some results on basic data structures that we could not find in the standard library.)

4 Turing Completeness of Choreographies

We are now ready to show that the choreographic language is Turing complete, in the sense that every partial recursive function can be implemented as a choreography (for a suitable definition of implementation). The construction is very similar to that in [7]: a significant part of the formalisation amounted to transcribing all the relevant definitions to Coq syntax. This contributes to confirming that the simplifications introduced in [28] and the additional notions and properties (rich labels, well-formedness, etc.) introduced by our formalisation are mostly internal and aimed at simplifying metatheoretical reasoning on choreographies.

4.1 Concrete Language

The first step is to instantiate the parameters in the definition of `CC` with the right types. Process identifiers, values and procedure names are natural numbers. Each process contains two variables; we use `Bool` for this type, and alias its elements to `xx` and `yy` for clarity. Expressions are an inductive type with three elements: `this` (evaluating to the process's value at `xx`), `zero` (evaluating to 0) and `succ_this` (evaluating to the successor of the value at `xx`). Boolean expressions are a singleton type with one element `compare`, which evaluates to `true` exactly when the process's two variables store the same element.

We restrict the syntax of choreographies to mimic the operators from the original development in [7], where processes only had one storage variable. Thus, incoming value communications are always stored at variable `xx`. However, in that calculus the conditional compared the value stored in two processes. We model this as a communication whose result is stored at `yy`, followed by a call to `compare`. We define these operations as macros.

```
Definition Send p e q := p#e ⟶ q$xx.
Definition IfEq p q C1 C2 := q#this ⟶ p$yy;; If p ? compare Then C1 Else C2.
```

4.2 Encoding

The most complex step of the formalisation is formalising the encoding of partial recursive functions as choreographies. This is naturally a recursive construction, but there are some challenges. First, non-base functions need to store intermediate computation results in auxiliary processes; second, recursion and minimisation use procedure definitions to implement loops. The strategy in [7] was to use auxiliary processes sequentially: since we can statically determine how many processes are needed from the definition of the function to encode, we can always determine the first unused process. The language used therein did not have the problem with recursion variables, but the same technique applies.

The key definition is the following: a choreography C implements function $f : \mathbb{N}^m \to \mathbb{N}$ with input processes $p_1, \ldots, p_m$ and output process q iff: for any state s where $p_1, \ldots, p_m$ contain the values $n_1, \ldots, n_m$ in their variable `xx`, (i) if $f(n_1, \ldots, n_m) = n$, then all executions of C from s terminate, and do so in a state where q stores n in its variable `xx`; and (ii) if $f(n_1, \ldots, n_m)$ is undefined, then execution of C from s never terminates. This is captured in the following Coq definition.

```
Definition implements (P:Program) {n} (f:PRFunction n) (ps:t Pid n) (q:Pid) :=
  ∀ (xs:t nat n) (s:State), (∀ Hi, s (ps[@Hi]) xx = xs[@Hi]) →
  (∀ y, converges f xs y ↔ ∃ s' ts P',
      (P,s) —[ts]⟶ * (P',s') ∧ s' q xx = y ∧ Main P' = End) ∧
  (diverges f xs ↔ ∀ s' ts P', (P,s) —[ts]⟶ * (P',s') → Main P' ≠ End).
```

The idea is that we recursively define the set of procedure definitions needed to encode $f : \mathbb{N}^m \to \mathbb{N}$, taking as parameters not only the processes $p_1, \ldots, p_m$ and q, but also the indices of the first unused process and the first unused procedure. The encoding of f is a program whose set of procedure definitions is obtained by instantiating the last two values to $\max(p_1, \ldots, p_m, q) + 1$ and 0, respectively, and whose main choreography is `Call 0`. Furthermore, we also ensure that the choreography terminates by calling the first unused procedure (which by default is defined as the terminated choreography). This makes it easy to ensure that procedure calls compose nicely in the recursive steps of the construction.

We start by defining two auxiliary functions `Pi` and `Gamma`, which given a function in `PRFunction` return the number of processes and procedures needed to encode it, respectively. (Function `Pi` is exactly the function Π from [7].) No results about these functions are needed – their definition suffices to prove all needed results, namely that no process (resp. procedure) higher than `Pi f` (resp. `Gamma f`) is used when encoding `f`.

Ironically, the most challenging part of the definition is composition (which is straightforward in the informal presentation), and not minimisation (which is responsible for introducing partiality). The base cases are directly encoded by suitably adapting the definitions from [7], and those of recursion and minimisation have a recursive structure very close to the informal textbook definition. However, the definition of composition needs to be recursive (due to the variable number of argument functions), and working with vectors adds a layer of complexity. As such, a number of auxiliary functions were defined to deal with composition, defining a choreography that encodes a vector of functions all with the same inputs and returning outputs in consecutive processes.

The recursive definition of encoding `Encoding_rec` is again written interactively, and afterwards a number of lemmas prove that it behaves as expected, e.g.:

```
Lemma Zero_Procs : ∀ d Hd ps q n X,
  Encoding_rec Zero d Hd ps q n X X = Send (hd ps) zero q;; Call (S X).
```

```
Lemma Recursion_Procs_g : ∀ k (g:PRFunction k) h d (Hd:depth (Recursion g h) < S d) ps q n,
  ∀ X Y, X ≤ Y < X + Gamma g → let Hg := (...) in
  Encoding_rec (Recursion g h) _ Hd ps q n X Y =
    Encoding_rec _ _ Hg (tl ps) n (n+3) X Y.
```

where we omit the definition of the proof term `Hg`. Note that `Encoding_rec` has the complex type `∀ (m:nat) (f:PRFunction m) (d:nat), depth f<d → t Pid m → Pid → nat → RecVar → RecVar → Choreography` – it receives a function of depth smaller than d, the set of input processes, the output process, the index of the first unused process and the index of the first unused procedure definition, and returns a mapping of procedure definitions to choreographies (where all previously defined procedures are unchanged).

Finally, we prove that encoding always returns a well-formed choreography. This is implicit in [7], but it is an essential property that should hold. For convenience, each condition of well-formedness is proved separately, capitalising on the fact that the encoding returns an initial choreography. The proof follows the recursive structure of the definition of `Encoding_rec`, and is relatively automatic once the relevant splitting in cases is done.

```
Lemma Encoding_WF : ∀ {n} (f:PRFunction n) ps q, ~In q ps → CCP_WF (Encoding f ps q).
```

4.3 Soundness

Soundness of the encoding – the property that the encoding of f implements f – is proven by analysing the execution path obtained by always reducing the first action in the choreography, and invoking confluence. We split the proof into a number of lemmas, stating the obvious reductions from each procedure definition to the procedure call at its end. We give some examples of these results.

```
Lemma Zero_reduce : ∀ Defs (ps: t Pid 1) q X s,
  ∃ t, (Build_Program Defs (Send (hd ps) zero q;; Call X),s)
    —[t]⟶ (Build_Program Defs (Call X), update s q xx 0).

Lemma Recursion_reduce_0 : ∀ Defs X n s,
  ∃ t, (Build_Program Defs (Send (n + 2) zero (S n);; Call X),s)
    —[t]⟶ (Build_Program Defs (Call X),update s (S n) xx 0).

Lemma Recursion_reduce_1_true : ∀ m Defs X Y n (ps:t Pid (S m)) q s,
  s (S n) xx = s (hd ps) xx → ∃ t s',
  (Build_Program Defs (IfEq (S n) (hd ps) (Send n this q;; Call X) (Call Y)),s)
    —[t]⟶ * (Build_Program Defs (Call X), s')
  ∧ s' q xx = s n xx ∧ ∀ p, p ≠ q → s' p xx = s p xx.

Lemma Recursion_reduce_1_false : ∀ m Defs X Y n (ps:t Pid (S m)) q s,
  s (S n) xx ≠ s (hd ps) xx → ∃ t,
  (Build_Program Defs (IfEq (S n) (hd ps) (Send n this q;; Call X) (Call Y)),s)
    —[t]⟶ * (Build_Program Defs (Call Y), update s (S n) yy (s (hd ps) xx)).
```

We briefly explain the lemmas about recursion. Encoding $R(g, h)$ uses three auxiliary procedures. The first one initializes the recursion by placing a zero on the first auxiliary process (Lemma `Recursion_reduce_0`), and calls the first procedure in the encoding of g. The second one, placed immediately after the procedures used for encoding g, checks whether the value in the auxiliary process is the value where we want to stop, and in this case places the result in the return process (Lemma `Recursion_reduce_1_true`). Otherwise, it calls the first procedure in the encoding of h (Lemma `Recursion_reduce_1_true`). (This explanation assumes the values of `X` and `Y` to be instantiated in the right way, but the lemmas do not depend on this.) The third procedure, invoked when h terminates, increases the value of the auxiliary process controlling the loop (Lemma `Recursion_reduce_2`, omitted). All these lemmas also include characterisations of the resulting state that are needed for applying the induction hypothesis in the main proof.

Using these results, we can prove by induction that, if s is a state where $n_1, \ldots, n_m$ are stored in the appropriate processes, (i) if $f(n_1, \ldots, n_k)$ converges, then there is some execution path of the encoding of f from s that terminates with the expected result; (ii) using confluence, all execution paths from s must terminate in the same state; and (iii) that if $f(n_1, \ldots, n_k)$ diverges, then no execution of the encoding of f from s terminates. These three results are combined in a single theorem, stating that the default encoding of f (where $n_1, \ldots, n_m$ are stored in processes $1, \ldots, m$, and the result is returned in process 0) is sound.

```
Theorem encoding_sound : ∀ n (f:PRFunction n), implements (Encoding' f) f (vec_1_to_n n) 0.
```

This part of the development contains 28 definitions and 65 lemmas, with a total of 2352 lines of code.

5 Discussion

We presented a formalisation of a choreographic language and proved it Turing complete. To the best of our knowledge, this is the first time that such a task has been achieved – the only comparable work in progress consists of a preliminary presentation on a certified compiler from choreographies to CakeML [17], which however does not deal with the major challenge of recursion (the choreography language used therein is simplistic and can only express finite behaviour) [23]. Moreover, we showed how formalising proofs unveiled subtle problems in definitions and can influence the development of the theory, making a case for a more systematic use of theorem provers in research in the field. The number of choreographic languages proposed in the literature is increasing rapidly, to include features of practical value such as asynchronous communication, non-determinism, broadcast, dynamic network topologies, and more [1, 14, 19]. Hopefully, our work can contribute a solid foundation for the development of these features. This recalls the situation found in the field of process calculi, and indeed similar conclusions are drawn in an article that presents the formalisation of a higher-order process calculus in Coq [26].

Our formalisation of Kleene's partial recursive functions is similar to the one in [34], of which we were not aware when we started this project. The library of proofs of undecidability formalised in Coq, being developed at the University of Saarbrücken, also includes definitions and results related to ours [12, 13].

It is interesting that the proof of Turing completeness – both construction and proofs – still closely follows the original theory [7], despite the significant changes to the lowest layers that we had to make. This suggests that the major formalisation challenge currently lies in the foundational work.

Our formalisation includes some design options. The most significant one, in our opinion, is the restriction to only two labels in selections. However, as is well known in the field of session types, this is not a serious restriction [3]. Labels are typically used to communicate choices based on a conditional; more complex decisions are expressed as nested conditionals, and can be communicated by sending multiple label selections.

Restricting the set of labels to two elements also has a strong impact on the formalisation of realisability [2, 4], which we do not discuss in this article. Choreography realisability deals with identifying sufficient conditions for a choreography to be implementable in a distributed setting, and generating an implementation in a process calculus automatically. The formalisation of this construction, described in [9], was completed after the current work, and heavily relies on the set of labels being fixed. An important result is that any choreography can be amended into a realisable one, so that in particular our Turing-completeness result immediately implies Turing-completeness of the process calculus used for implementations.

We aimed at making our development reusable, so that it can readily be extended to more expressive choreographic languages. In the future, we plan to look at interesting extensions (such as those mentioned above) and explore how easy it is to extend the current formalisation to those frameworks. We conjecture that this will prove much simpler than the current effort, thanks to the structures established in this work.

References

1 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2–3):95–230, 2016.

2 Samik Basu, Tevfik Bultan, and Meriem Ouederni. Deciding choreography realizability. In John Field and Michael Hicks, editors, *Procs. POPL*, pages 191–202. ACM, 2012. `doi:10.1145/2103656.2103680`.

3 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In Paul Gastin and François Laroussinie, editors, *Procs. CONCUR*, volume 6269 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 2010. `doi:10.1007/978-3-642-15375-4_16`.

4 Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8:1–8:78, 2012. `doi:10.1145/2220365.2220367`.

5 Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In Roberto Giacobazzi and Radhia Cousot, editors, *Procs. POPL*, pages 263–274. ACM, 2013. `doi:10.1145/2429069.2429101`.

6 Luís Cruz-Filipe and Fabrizio Montesi. Choreographies in practice. In Elvira Albert and Ivan Lanese, editors, *Procs. FORTE*, volume 9688 of *Lecture Notes in Computer Science*, pages 114–123. Springer, 2016. `doi:10.1007/978-3-319-39570-8_8`.

7 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`.

8 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Choreographies in Coq. In *TYPES 2019, Abstracts*, 2019. Extended abstract.

9 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Certifying choreography compilation. *CoRR*, abs/2102.10698, 2021. URL: `https://arxiv.org/abs/2102.10698`.

10 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. A formalisation of a Turing-complete choreographic language in Coq, 2021. `doi:10.5281/zenodo.4479102`.

11 Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies: Theory and implementation. *Log. Methods Comput. Sci.*, 13(2), 2017. `doi:10.23638/LMCS-13(2:1)2017`.

12 Yannick Forster, Dominik Kirst, and Gert Smolka. On synthetic undecidability in Coq, with an application to the Entscheidungsproblem. In Assia Mahboubi and Magnus O. Myreen, editors, *Procs. CPP*, pages 38–51. ACM, 2019. `doi:10.1145/3293880.3294091`.

13 Yannick Forster, Fabian Kunze, and Maximilian Wuttke. Verified programming of Turing machines in Coq. In Jasmin Blanchette and Cătălin Hriţcu, editors, *Procs. CPP*, pages 114–128. ACM, 2020. `doi:10.1145/3372885.3373816`.

14 Simon J. Gay, Vasco T. Vasconcelos, Philip Wadler, and Nobuko Yoshida. Theory and applications of behavioural types (Dagstuhl Seminar 17051). *Dagstuhl Reports*, 7(1):158–189, 2017. `doi:10.4230/DagRep.7.1.158`.

15 Saverio Giallorenzo, Ivan Lanese, and Daniel Russo. Chip: A choreographic integration process. In Hervé Panetto, Christophe Debruyne, Henderik A. Proper, Claudio Agostino Ardagna, Dumitru Roman, and Robert Meersman, editors, *Procs. OTM, part II*, volume 11230 of *Lecture Notes in Computer Science*, pages 22–40. Springer, 2018. doi:10.1007/978-3-030-02671-4_2.

16 Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. Choreographies as objects. *CoRR*, abs/2005.09520, 2020. URL: https://arxiv.org/abs/2005.09520.

17 Alejandro Gomez-Londono and Johannes Aman Pohjola. Connecting choreography languages with verified stacks. In *Procs. of the Nordic Workshop on Programming Theory*, pages 31–33, 2018. URL: http://hdl.handle.net/102.100.100/86327?index=1.

18 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9, 2016. Also: POPL, pages 273–284, 2008. doi:10.1145/2827695.

19 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016. doi:10.1145/2873052.

20 Intl. Telecommunication Union. Recommendation Z.120: Message Sequence Chart, 1996.

21 Stephen Cole Kleene. *Introduction to Metamathematics*, volume 1. North-Holland Publishing Co., 1952.

22 Alberto Lluch-Lafuente, Flemming Nielson, and Hanne Riis Nielson. Discretionary information flow control for interaction-oriented specifications. In Narciso Martí-Oliet, Peter Csaba Ölveczky, and Carolyn L. Talcott, editors, *Logic, Rewriting, and Concurrency*, volume 9200 of *Lecture Notes in Computer Science*, pages 427–450. Springer, 2015. doi:10.1007/978-3-319-23165-5_20.

23 Alejandro Gómez Londoño. Choreographies and cost semantics for reliable communicating systems. Licentiate thesis, Chalmers University of Technology, 2020.

24 Hugo A. López and Kai Heussen. Choreographing cyber-physical distributed control systems for the energy sector. In Ahmed Seffah, Birgit Penzenstadler, Carina Alves, and Xin Peng, editors, *Procs. SAC*, pages 437–443. ACM, 2017. doi:10.1145/3019612.3019656.

25 Hugo A. López, Flemming Nielson, and Hanne Riis Nielson. Enforcing availability in failure-aware communicating systems. In Elvira Albert and Ivan Lanese, editors, *Procs. FORTE*, volume 9688 of *Lecture Notes in Computer Science*, pages 195–211. Springer, 2016. doi:10.1007/978-3-319-39570-8_13.

26 Petar Maksimovic and Alan Schmitt. HOCore in Coq. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2015. doi:10.1007/978-3-319-22102-1_19.

27 Fabrizio Montesi. *Choreographic Programming.* Ph.D. Thesis, IT University of Copenhagen, 2013. http://www.fabriziomontesi.com/files/choreographic_programming.pdf.

28 Fabrizio Montesi. Introduction to Choreographies. Accepted for publication by Cambridge University Press, 2020.

29 Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978. doi:10.1145/359657.359659.

30 Object Management Group. Business Process Model and Notation. http://www.omg.org/spec/BPMN/2.0/, 2011.

31 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, 2019. doi:10.1145/3290343.

32 Alan M. Turing. Computability and λ-definability. *J. Symb. Log.*, 2(4):153–163, 1937. doi:10.2307/2268280.

33 W3C. WS Choreography Description Language. http://www.w3.org/TR/ws-cdl-10/, 2004.

34 Vincent Zammit. A proof of the s-m-n theorem in coq. Technical report, The Computing Laboratory, The University of Kent, Canterbury, Kent, UK, 1997. URL: https://kar.kent.ac.uk/21524/.

A Natural Formalization of the Mutilated Checkerboard Problem in Naproche

Adrian De Lon ✉
Universität Bonn, Germany

Peter Koepke ✉
Universität Bonn, Germany

Anton Lorenzen ✉
Universität Bonn, Germany

Abstract

ℕaproche is an emerging *natural proof assistant* that accepts input in a controlled natural language for mathematics, which we have integrated with LaTeX for ease of learning and to quickly produce high-quality typeset documents. We present a self-contained formalization of the *Mutilated Checkerboard Problem* in ℕaproche, following a proof sketch by John McCarthy. The formalization is embedded in detailed literate style comments. We also briefly describe the ℕaproche approach.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases checkerboard, formalization, formal mathematics, controlled language

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.16

Supplementary Material *System (Source Code)*: `https://github.com/naproche/naproche`

1 Introduction

We illustrate the potential of *natural interactive theorem proving* by a formalization of the *Mutilated Checkerboard Problem* in the interactive proof assistant ℕaproche (Natural Proof Checking). The formalization employs the (controlled) natural mathematical language ForTheL (Formula Theory Language), which is immediately readable by mathematicians and has obvious first-order semantics. ForTheL allows familiar definition-axiom-theorem-proof text structures. The language is integrated into LaTeX (see also [10]) so that the formalization document can be viewed and printed in high-quality mathematical typesetting. In the spirit of *literate programming* [8], the actual formalization, indicated with a grey background, is embedded into ample commentary for the benefit of human readers. Thus the whole article is a valid proof-checked ForTheL document.

The *Mutilated Checkerboard Problem*, which will be explained in detail in the formalization in Section 3, was proposed by John McCarthy as a challenge ("tough nut") [12] to automatic and interactive theorem proving. By now there are many formalizations of the problem (see the survey article *A Tough Nut for Mathematical Knowledge Management* by Manfred Kerber and Martin Pollet [7]).

Mathematically, our formalization follows McCarthy's sketch *The mutilated checkerboard in set theory* [13]. Our formalization is fully self-contained and includes definitions and axioms about finite sets, functions and cardinalities. Since ℕaproche is based on a weakly typed first-order logic, modelling the checkerboard employs first-order relations, functions and constants. Our axioms are evidently true in a "standard model" of a checkerboard and finite sets. In a stronger foundational theory like Zermelo-Fraenkel set theory, our axioms are provable when one replaces "set" by "finite set".

12th International Conference on Interactive Theorem Proving (ITP 2021).
Editors: Liron Cohen and Cezary Kaliszyk; Article No. 16; pp. 16:1–16:11
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In Section 2 we briefly describe the $\mathbb{N}$aproche system and its input language ForTheL. Section 3 contains the actual formalization of the *Mutilated Checkerboard Problem*; the text includes explanations of axiomatic and mathematical details of the argument. Further technical remarks about the formalization are elaborated in Section 4. In the final Section 5 we propose further improvements to the $\mathbb{N}$aproche system.

We think that naturalness of interaction will be a crucial factor for the acceptance of proof assistants by the mathematical community.

2 The Natural Proof Assistant $\mathbb{N}$aproche

While state-of-the-art interactive theorem provers have been used to prove and certify highly non-trivial research mathematics, they are still, according to Lawrence Paulson, "unsuitable for mathematics. Their formal proofs are unreadable." [17]. *Natural proof assistants* intend to bridge the wide gap between intuitive mathematical texts and the formal rigour of logical calculi. This requires in particular

- input languages close to the mathematical vernacular, including symbolic expressions;
- text structurings like the axiom-definition-theorem-proof schema;
- natural argumentative phrases for various proof tactics;
- familiar logics and mathematical ontologies;
- strong automatic theorem proving to fill in obvious proof details;
- an intuitive editor for text and theory development which interactively integrates the checking process and guides formalization.

The $\mathbb{N}$aproche proof assistant stems from two long-term efforts devoted to these goals: the Evidence Algorithm (EA) / System for Automated Deduction (SAD) projects at the universities of Kiev and Paris [15, 16, 19, 20], and the Naproche project at Bonn [11, 3, 4, 9]. The ForTheL input language of SAD has been extended and embedded in LaTeX, allowing mathematical typesetting; the original proof-checking mechanisms have been made more efficient and varied. Moreover, $\mathbb{N}$aproche has been integrated into the Isabelle Proof Interactive Development Environment (Isabelle/PIDE) [21] which supports interactive editing and checking of mathematical texts. $\mathbb{N}$aproche, however, is not connected to the standard logics implemented in Isabelle. Some comprehensive readable formalizations at the level of undergraduate mathematics have been undertaken in $\mathbb{N}$aproche and are available online [2].

$\mathbb{N}$aproche uses classical first-order logic as its underlying logic (following the SAD approach), giving direct access to strong ATPs like E [18]. The input language ForTheL has been carefully designed to approximate the weakly typed natural language of mathematics whilst being efficiently translatable to the language of first-order logic. In ForTheL, standard mathematical types are called *notions*, and these are internally represented as unary predicates with first-order definitions. This leads to a flexible type system where number systems can be cumulative ($\mathbb{N} \subseteq \mathbb{R}$), and notions can depend on parameters (subsets of $\mathbb{N}$, divisors of n).

The first-order language of notions, constants, relations, and functions is introduced and extended by *signature* and *definition* commands as in this example which is unrelated to the checkerboard formalization:

Signature. A real number is a notion.
Definition. $\mathbb{R}$ is the set of real numbers.
Signature. 0 is a real number.
Definition. A nonzero number is a real number that is not equal to 0.
Signature. Let x, y be real numbers. Let y be a nonzero number. $\frac{x}{y}$ is a real number.

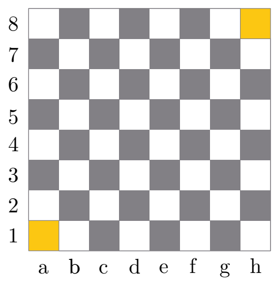
ForTheL requires that all variables and terms have some type in the declared system of notions. As part of proof-checking, Naproche will first check the type-correctness of all terms in a ForTheL statement before proceeding to the further processing of the statement. This type-checking phase is called *ontological checking*. Due to the first-order dependent definition of types, type-checking is more involved than in static type systems. On the other hand this approach naturally supports standard partial functions like $\frac{x}{y}$ for reals x, y. The guard $y \neq 0$ will only be checked after the checking process has gone through all the text preceding the term $\frac{x}{y}$ in question.

Our formalization of the *Mutilated Checkerboard Problem* in the next section can be read as a gentle introduction to ForTheL, which declares a language of checkerboards, dominoes and tilings, postulates some axioms, and proceeds to show simple propositions which result in the final non-tileability.

The formalization is carried out in the Isabelle 2021 PIDE which includes a Naproche component. Further mathematical and logical particulars are contained in the literate comments in the formalization; technical information on the use of Naproche and Isabelle is given in Section 4.

3 The Mutilated Checkerboard Problem Formalized in Naproche

3.1 Introduction



The *Mutilated Checkerboard Problem* asks the following:

Consider an 8×8 checkerboard with two diametrically opposed corners 🟨 removed, leaving 62 squares.

Is it possible to place 31 dominoes of size 2×1 so as to cover all 62 remaining squares?

Max Black proposed this problem in his book *Critical Thinking* (1946). It was later discussed by Martin Gardner in his *Scientific American* column, *Mathematical Games*. John McCarthy, one of the founders of Artificial Intelligence, described it as a *Tough Nut for Proof Procedures* and discussed fully automatic or interactive proofs of the solution.

3.2 Setting up the checkerboard

We introduce *types* (or *notions*) and constants to model checkerboards as a Cartesian product of *ranks* $1, 2, \dots, 8$ and *files* $\mathrm{a}, \mathrm{b}, \dots, \mathrm{h}$, where we follow standard checkerboard notation. In future versions of Naproche these signature declarations should be grouped as a single declaration of an inductively defined set, allowing phrasings such as *1, 2, 3, 4, 5, 6, 7, 8 are ranks*. Note that the effect of signature declarations is to extend the underlying first-order language. Naproche treats $1, 2, \dots$ as new constant symbols which have no connection to the homonymous integers and in particular do not carry assumptions about distinctness. Here our approach diverges from McCarthy's who employs integers modulo 8, but this would require us to formalize part of the theory of $\mathbb{Z}/8\mathbb{Z}$.

Naproche allows to group elements into *classes* and *sets* as long as they are *setsized* (informally also called *small*).

Signature 1. A rank is a notion.
Let r, s denote ranks.

Axiom 2. r is setsized.

Signature 3. 1 is a rank.

Signature 4. 2 is a rank.

Signature 5. 3 is a rank.

Signature 6. 4 is a rank.

Signature 7. 5 is a rank.

Signature 8. 6 is a rank.

Signature 9. 7 is a rank.

Signature 10. 8 is a rank.

Definition 11. $\mathbf{R} = \{1, 2, 3, 4, 5, 6, 7, 8\}$.

Signature 12. A file is a notion.

Let f, g denote files.

Axiom 13. f is setsized.

Signature 14. a is a file.

Signature 15. b is a file.

Signature 16. c is a file.

Signature 17. d is a file.

Signature 18. e is a file.

Signature 19. f is a file.

Signature 20. g is a file.

Signature 21. h is a file.

Definition 22. $\mathbf{F} = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}, \mathrm{f}, \mathrm{g}, \mathrm{h}\}$.

Signature 23. A square is a notion.

Axiom 24. (f, r) is a square.

Let v, w, x, y, z denote squares.

Is there a set of all squares? This may not be true for an arbitrary notion, but it is true for squares, so we assume it as an axiom. Note that we can always form the class C of all inhabitants of a notion as long as $x \in C$ can only be true for setsized x. Morse and Kelley [6, 14] use the same approach in their axiomatization of set theory.

Definition 25. $\mathbf{C}$ is the class of squares x such that $x = (f, r)$ for some element f of $\mathbf{F}$ and some element r of $\mathbf{R}$.

Axiom 26. $\mathbf{C}$ is a set.

3.3 Preliminaries about sets and functions

We enrich the small built-in set theory with further properties and axioms that will be used in the course of our argument. To keep the document fully self-contained we formulate the necessary definitions and axioms ourselves. Note that there are many degrees of freedom in picking an axiomatic setting.

In Definition 27, we do not define the *relation* $A \subseteq B$, but rather the *type* of subsets of a given set B. The type is parametrized by a member of the type *set*. This is a kind of dependent type, depending on B, in the sense of dependent type theory. Therefore the wording *A [...] is a [...]* is used in Definition 27 as well as in Definition 29. More specifically, we use a form with an argument prefixed by *of*, a so-called *of*-notion. This allows certain grammatical variants and constructs like *B has a subset A* or *every subset of A satisfies [...]*. Note that an *element* similarly is an *of*-notion so that one could write phrases like *x is an element of B* or even more complicated ones like *for all nonequal elements A, B of C*. One could also have defined the *relation* $A \subseteq B$ by a statement of the form: *$A \subseteq B$ iff [...]*. Definition 30 defines disjointness in that format.

Let A, B, C denote sets.

Definition 27. A subset of B is a set A such that every element of A is an element of B.

Axiom 28 (Extensionality). If A is a subset of B and B is a subset of A then $A = B$.

Definition 29. A proper subset of B is a subset A of B such that $A \neq B$.

Definition 30. A is disjoint from B iff there is no element of A that is an element of B.

Definition 31. A family is a set F such that every element of F is a set.

Definition 32. A disjoint family is a family F such that A is disjoint from B for all nonequal elements A, B of F.

Definition 33. $B \cap C = \{x \in B \mid x \in C\}$.

Definition 34. $B \setminus C = \{x \in B \mid x \notin C\}$.

The notion of *object* is the built-in *largest notion*, containing all other notions. Also note that the proof of the lemma below really is omitted and not merely hidden: with its internal "reasoner" and in non-trivial cases with the help of automated theorem provers such as E, ℕaproche can accept some theorems without any additional argumentation.

Lemma 35. Every set is an object.

The built-in ordered pair notation that we already used in the first subsection does not include the universal property of ordered pairs, so we postulate it as an axiom.

Axiom 36. Let $\alpha, \beta, \gamma, \delta$ be objects. If $(\alpha, \beta) = (\gamma, \delta)$ then $\alpha = \gamma$ and $\beta = \delta$.

(Unary) functions are built into ℕaproche; $F(t)$ denotes the application of a function F to an argument t and $\mathrm{Dom}(F)$ stands for the domain of F. In our exposition we shall use functions to compare cardinalities of black and white squares. As with sets, we introduce some further properties of functions.

Let F, G denote functions.

Definition 37. $F : A \to B$ iff $\mathrm{Dom}(F) = A$ and $F(x)$ is an element of B for all elements x of A.

Bijective functions are the basis of the modern theory of cardinalities; sets have the same cardinality iff there is a bijection between them.

Definition 38. $F : A \leftrightarrow B$ iff $F : A \to B$ and there exists G such that $G : B \to A$ and (for all elements x of A we have $G(F(x)) = x$) and (for all elements y of B we have $F(G(y)) = y$).

3.4 Cardinalities of Finite Sets

Definition 39. A is equinumerous with B iff there is F such that $F : A \leftrightarrow B$.

Lemma 40. Assume that A is equinumerous with B. Then B is equinumerous with A.

Lemma 41. Assume that A is equinumerous with B and B is equinumerous with C. Then A is equinumerous with C.

Proof. Take a function F such that $F : A \leftrightarrow B$. Take a function G such that $G : B \to A$ and (for all elements x of A we have $G(F(x)) = x$) and (for all elements y of B we have $F(G(y)) = y$). Take a function H such that $H : B \leftrightarrow C$. Take a function I such that $I : C \to B$ and (for all elements x of B we have $I(H(x)) = x$) and (for all elements y of C we have $H(I(y)) = y$). Define $J(x) = H(F(x))$ for x in A. $J : A \leftrightarrow C$. Indeed define $K(y) = G(I(y))$ for y in C. ◀

For the finite checkerboard problem we only need to consider finite sets. We can thus assume that all sets considered are finite, and then we have the following finiteness axiom:

Axiom 42. If A is a proper subset of B then A is not equinumerous with B.

3.5 The Mutilated Checkerboard

Defining the mutilated checkerboard is straightforward: we simply remove the two corners.

Definition 43. $\mathbf{C}' = \{(\mathrm{a}, 1), (\mathrm{h}, 8)\}$.

Definition 44. $\mathbf{M} = \mathbf{C} \setminus \mathbf{C}'$.

Let the mutilated checkerboard stand for $\mathbf{M}$.

3.6 Dominoes

To define dominoes, we introduce concepts of adjacency by first declaring new relations and then axiomatizing them. As usual, chaining of relation symbols indicates a conjunction.

Signature 45. r is vertically adjacent to s is a relation.
Let $r \sim s$ stand for r is vertically adjacent to s.

Axiom 46. If $r \sim s$ then $s \sim r$.

Axiom 47. $1 \sim 2 \sim 3 \sim 4 \sim 5 \sim 6 \sim 7 \sim 8$.

Signature 48. f is horizontally adjacent to g is a relation.

Let $f \sim' g$ stand for f is horizontally adjacent to g.

Axiom 49. If $f \sim' g$ then $g \sim' f$.

Axiom 50. $\mathrm{a} \sim' \mathrm{b} \sim' \mathrm{c} \sim' \mathrm{d} \sim' \mathrm{e} \sim' \mathrm{f} \sim' \mathrm{g} \sim' \mathrm{h}$.

Definition 51. x is adjacent to y iff there exist f, r, g, s such that $x = (f, r)$ and $y = (g, s)$ and (($f = g$ and r is vertically adjacent to s) or ($r = s$ and f is horizontally adjacent to g)).

Definition 52. A domino is a set D such that $D = \{x, y\}$ for some adjacent squares x, y.

3.7 Domino Tilings

Definition 53. A domino tiling is a disjoint family T such that every element of T is a domino.

Let A denote a subset of $\mathbf{C}$.

Definition 54. A domino tiling of A is a domino tiling T such that for every square x x is an element of A iff x is an element of some element of T.

We shall prove:

Theorem. The mutilated checkerboard has no domino tiling.

3.8 Colours

We shall solve the mutilated checkerboard problem by a cardinality argument. Squares on an actual checkerboard are coloured black and white and we can count colours on dominoes and on the mutilated checkerboard $\mathbf{M}$.

The introduction of colours can be viewed as a creative move typical of mathematics: changing perspectives and introducing aspects that are not part of the original problem. The mutilated checkerboard was first discussed under a cognition-theoretic perspective: can one solve the problem *without* inventing new concepts and completely stay within the realm of squares, subsets of the checkerboard and dominoes.

Signature 55. x is black is a relation.

Signature 56. x is white is a relation.

Axiom 57. x is black iff x is not white.

Axiom 58. If x is adjacent to y then x is black iff y is white.

Axiom 59. $(\mathrm{a}, 1)$ is black.

Axiom 60. $(\mathrm{h}, 8)$ is black.

Definition 61. $\mathbf{B}$ is the class of black elements of $\mathbf{C}$.

Definition 62. $\mathbf{W}$ is the class of white elements of $\mathbf{C}$.

Lemma 63. $\mathbf{B}$ is a set.

Lemma 64. $\mathbf{W}$ is a set.

3.9 Counting Colours on Checkerboards

The original checkerboard has an equal number of black and white squares. Since our setup does not include numbers for counting, we rather work with equinumerosity. The following argument formalizes that we can invert the colours of a checkerboard by swapping the files a and b, c and d, and so on. We formalize swapping by a first-order function symbol Swap.

Signature 65. Let x be an element of $\mathbf{C}$. $\mathrm{Swap}\, x$ is an element of $\mathbf{C}$.

Let t denote an element of $\mathbf{R}$.

Axiom 66. $\mathrm{Swap}(\mathrm{a}, t) = (\mathrm{b}, t)$ and $\mathrm{Swap}(\mathrm{b}, t) = (\mathrm{a}, t)$.

Axiom 67. $\mathrm{Swap}(\mathrm{c}, t) = (\mathrm{d}, t)$ and $\mathrm{Swap}(\mathrm{d}, t) = (\mathrm{c}, t)$.

Axiom 68. $\mathrm{Swap}(\mathrm{e}, t) = (\mathrm{f}, t)$ and $\mathrm{Swap}(\mathrm{f}, t) = (\mathrm{e}, t)$.

Axiom 69. $\mathrm{Swap}(\mathrm{g}, t) = (\mathrm{h}, t)$ and $\mathrm{Swap}(\mathrm{h}, t) = (\mathrm{g}, t)$.

The somewhat unsightly case-splits in the following lemmas are necessary to guide the prover, since ℕaproche as yet has no concept of finite data types and only features well-ordered induction. As a consolation price, we can omit the last case.

Lemma 70. Let x be an element of $\mathbf{C}$. $\mathrm{Swap}\, x$ is adjacent to x.

Proof. Take f, r such that $x = (f, r)$. r is an element of $\mathbf{R}$. Case $f = \mathrm{a}$. End. Case $f = \mathrm{b}$. End. Case $f = \mathrm{c}$. End. Case $f = \mathrm{d}$. End. Case $f = \mathrm{e}$. End. Case $f = \mathrm{f}$. End. Case $f = \mathrm{g}$. End. ◀

Swap is an involution.

Lemma 71. Let x be an element of $\mathbf{C}$. $\mathrm{Swap}(\mathrm{Swap}\, x) = x$.

Proof. Take f, r such that $x = (f, r)$. r is an element of $\mathbf{R}$. Case $f = \mathrm{a}$. End. Case $f = \mathrm{b}$. End. Case $f = \mathrm{c}$. End. Case $f = \mathrm{d}$. End. Case $f = \mathrm{e}$. End. Case $f = \mathrm{f}$. End. Case $f = \mathrm{g}$. End. ◀

Lemma 72. Let x be an element of $\mathbf{C}$. x is black iff $\mathrm{Swap}\, x$ is white.

Using Swap we can define a witness of $\mathbf{B} \leftrightarrow \mathbf{W}$.

Lemma 73. $\mathbf{B}$ is equinumerous with $\mathbf{W}$.

Proof. Define $F(x) = \mathrm{Swap}\, x$ for x in $\mathbf{B}$. Define $G(x) = \mathrm{Swap}\, x$ for x in $\mathbf{W}$. Then $F : \mathbf{B} \to \mathbf{W}$ and $G : \mathbf{W} \to \mathbf{B}$. For all elements x of $\mathbf{B}$ we have $G(F(x)) = x$. For all elements x of $\mathbf{W}$ we have $F(G(x)) = x$. $F : \mathbf{B} \leftrightarrow \mathbf{W}$. ◀

Given a domino tiling one can also swap the squares of each domino, leading to similar properties.

Signature 74. Assume that T is a domino tiling of A. Let x be an element of A. $\mathrm{Swap}^A_T(x)$ is a square y such that there is an element D of T such that $D = \{x, y\}$.

Lemma 75. Assume that T is a domino tiling of A. Let x be an element of A. Then $\mathrm{Swap}^A_T(x)$ is an element of A.

Proof. Let $y = \mathrm{Swap}^A_T(x)$. Take an element D of T such that $D = \{x, y\}$. ◀

Swapping dominoes is also an involution.

Lemma 76. Assume that T is a domino tiling of A. Let x be an element of A. Then $\mathrm{Swap}^A_T(\mathrm{Swap}^A_T(x)) = x$.

Proof. Let $y = \mathrm{Swap}^A_T(x)$. Take an element Y of T such that $Y = \{x, y\}$. Let $z = \mathrm{Swap}^A_T(y)$. Take an element Z of T such that $Z = \{y, z\}$. Then $x = z$. ◀

Lemma 77. Assume that T is a domino tiling of A. Let x be a black element of A. Then $\mathrm{Swap}^A_T(x)$ is white.

Proof. Let $y = \mathrm{Swap}_T^A(x)$. Take an element Y of T such that $Y = \{x, y\}$. ◀

3.10 The Theorem

We can easily show that a domino tiling involves as many black as white squares.

Lemma 78. Let T be a domino tiling of A. Then $A \cap \mathbf{B}$ is equinumerous with $A \cap \mathbf{W}$.

Proof. Define $F(x) = \mathrm{Swap}_T^A(x)$ for x in $A \cap \mathbf{B}$. Define $G(x) = \mathrm{Swap}_T^A(x)$ for x in $A \cap \mathbf{W}$. Then $F : A \cap \mathbf{B} \to A \cap \mathbf{W}$ and $G : A \cap \mathbf{W} \to A \cap \mathbf{B}$. For all elements x of $A \cap \mathbf{B}$ we have $G(F(x)) = x$. For all elements x of $A \cap \mathbf{W}$ we have $F(G(x)) = x$. $F : A \cap \mathbf{B} \leftrightarrow A \cap \mathbf{W}$. ◀

In mutilating the checkerboard, one only removes black squares

Lemma 79. $\mathbf{M} \cap \mathbf{W} = \mathbf{W}$.

Proof. $\mathbf{M} \cap \mathbf{W}$ is a subset of $\mathbf{W}$. $\mathbf{W}$ is a subset of $\mathbf{M}$. Proof. Let x be an element of $\mathbf{W}$. $x \neq (\mathrm{a}, 1)$ and $x \neq (\mathrm{h}, 8)$. Indeed $(\mathrm{h}, 8)$ is black. End. ◀

Now the theorem follows by putting together the previous cardinality properties. Note that the phrasing *[...] has no domino tiling* in the theorem is automatically derived from the definition of *a domino tiling of [...]*.

Theorem 80. The mutilated checkerboard has no domino tiling.

Proof. Proof by contradiction. Assume T is a domino tiling of $\mathbf{M}$. $\mathbf{M}\cap\mathbf{B}$ is equinumerous with $\mathbf{M} \cap \mathbf{W}$. Indeed $\mathbf{M}$ is a subset of $\mathbf{C}$. $\mathbf{M} \cap \mathbf{B}$ is equinumerous with $\mathbf{W}$. $\mathbf{M} \cap \mathbf{B}$ is equinumerous with $\mathbf{B}$. Contradiction. Indeed $\mathbf{M} \cap \mathbf{B}$ is a proper subset of $\mathbf{B}$. ◀

4 Comments on the Formalization

We use Naproche within the current release of the Proof Interactive Development Environment (PIDE) Isabelle 2021 [5]. Isabelle 2021 is available for the operating systems Linux, Windows, and macOS. The distribution can be unpacked somewhere in one's home folder and started by clicking on the Isabelle executable in the Isabelle folder. The *Documentation* panel contains a tutorial on Naproche, which links to a standalone version of our formalization called `checkerboard.ftl.tex`. Opening a `.ftl` or `.ftl.tex` file in Isabelle/PIDE will automatically activate its parsing and checking. Files in `.ftl.tex` format can be typeset by LaTeX provided that text like the above **Signatures** or **Definitions** are entered in a simple LaTeX format. Only text in a `\begin{forthel} ... \end{forthel}` environment is let through to the checking process. Everything else is treated as a comment by Naproche, but may be relevant for LaTeX typesetting.

```
\section{Example of a Signature Command}
\begin{forthel}
    \begin{signature}
        Let $x,y$ be real numbers. Let $y$ be a nonzero number.
        $\frac{x}{y}$ is a real number.
    \end{signature}
\end{forthel}
```

Isabelle/PIDE can show pop-up first-order translations of statements while hovering above them and indicates checking progress with coloured backgrounds. Results of checking and error messages are shown in an output window.

Checking the formalization takes roughly one to two minutes, assuming somewhat up-to-date hardware. Omitting proofs (as in Lemma 40) is convenient and concise, but significantly increases the checking time, since E has to perform a considerable number of sledgehammer-like proof searches. Lemma 40 also shows that automated theorem provers and humans have different strengths. The result is immediate from the definitions to human readers. Conversely, sometimes it is be better to continue spelling out the details of a proof even after the computer accepts it, to help the human reader understand the rest of the proof.

5 Perspectives

The readability and naturalness of non-trivial texts which proof-check in the current, still modest ℕaproche system call for a significant extension of this project. For ℕaproche to become a true assistant in mathematical research and teaching, ad hoc methods have to be replaced by professional approaches and tools:

- the input language ForTheL has to be extended for wide mathematical coverage, informed by typical mathematical texts; ForTheL needs a formal grammar and vocabulary to be processed by strong linguistic methods; the vocabulary may also encompass standard (LaTeX) symbols and possibly contain semantic information;
- logical processing has to be geared to the strengths of current automated theorem provers; *Sledgehammer*-like methods should provide efficient premise selection in large texts and theories (see also [1] for a discussion of hammers);
- the creation of libraries of ForTheL documents requires import and export mechanisms corresponding to quoting and referencing in the mathematical literature;
- proof-checking of documents should be organized as an enrichment of ForTheL texts by the generated translations and derivations; these should be stored as auxiliary files to minimize re-checking or to assemble derivations into a correctness certificate for the text;
- the natural text processing of ℕaproche should be interfaced with other proof assistants to leverage their strengths and libraries; we have begun work on a ℕaproche $\rightarrow$ Lean translation;
- the use and user experience of natural proof checking in teaching and research have to be studied and taken care of in the further development.

References

1 Jasmin C. Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *Journal of Formalized Reasoning*, 9(1):101–148, January 2016. doi:10.6092/issn.1972-5787/4593.

2 Naproche contributors. FLib. URL: https://github.com/naproche-community/FLib.

3 Marcos Cramer. *Proof-checking mathematical texts in controlled natural language*. PhD thesis, University of Bonn, 2013.

4 Steffen Frerix and Peter Koepke. Automatic proof-checking of ordinary mathematical texts. *Proceedings of the Workshop Formal Mathematics for Mathematicians*, 2018.

5 Isabelle contributors. The Isabelle2021 release, February 2021. URL: https://isabelle.in.tum.de.

6 John L. Kelley. *General Topology*. Springer-Verlag New York, 1975.

7 Manfred Kerber and Martin Pollet. A tough nut for mathematical knowledge management. In Michael Kohlhase, editor, *Mathematical Knowledge Management*, pages 81–95, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. `doi:10.1007/11618027_6`.

8 Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992.

9 Peter Koepke. Textbook mathematics in the Naproche-SAD system. *Joint Proceedings of the FMM and LML Workshops*, 2019.

10 Peter Koepke, Anton Lorenzen, and Adrian De Lon. Interpreting mathematical texts in Naproche-SAD. In *Intelligent Computer Mathematics: 13th International Conference, CICM 2020*, pages 284–289. Springer, 2020.

11 Daniel Kühlwein, Marcos Cramer, Peter Koepke, and Bernhard Schröder. The Naproche system, January 2009.

12 John McCarthy. Tough nut for proof procedures, 1964. Stanford AI Memo.

13 John McCarthy. The mutilated checkerboard in set theory, 2001.

14 Anthony Perry Morse; Trevor J McMinn. *A theory of sets*. New York ; London : Academic press, 1965.

15 Andrei Paskevich. *Méthodes de formalisation des connaissances et des raisonnements mathématiques: aspects appliqués et théoriques*. PhD thesis, Université Paris 12, 2007.

16 Andrei Paskevich. The syntax and semantics of the ForTheL language, 2007.

17 Lawrence C. Paulson. ALEXANDRIA: Large-scale formal proof for the working mathematician. URL: `https://www.cl.cam.ac.uk/~lp15/Grants/Alexandria/`.

18 Stephan Schulz. The E theorem prover. URL: `https://eprover.org`.

19 Konstantin Verchinine, Alexander Lyaletski, and Andrei Paskevich. System for automated deduction (SAD): a tool for proof verification. *Automated Deduction–CADE-21*, pages 398–403, 2007. `doi:10.1007/978-3-540-73595-3_29`.

20 Konstantin Verchinine, Alexander Lyaletski, Andrei Paskevich, and Anatoly Anisimov. On correctness of mathematical texts from a logical and practical point of view. In *International Conference on Intelligent Computer Mathematics*, pages 583–598. Springer, 2008. `doi:10.1007/978-3-540-85110-3_47`.

21 Makarius Wenzel. Interaction with formal mathematical documents in Isabelle/PIDE, 2019. `arXiv:1905.01735`.

A Variant of Wagner's Theorem Based on Combinatorial Hypermaps

Christian Doczkal ✉
Université Côte d'Azur, Inria Sophia Antipolis Méditerranée (STAMP), France

Abstract

Wagner's theorem states that a graph is planar (i.e., it can be embedded in the real plane without crossing edges) iff it contains neither K_5 nor $\mathsf{K}_{3,3}$ as a minor. We provide a combinatorial representation of embeddings in the plane that abstracts from topological properties of plane embeddings (e.g., angles or distances), representing only the combinatorial properties (e.g., arities of faces or the clockwise order of the outgoing edges of a vertex). The representation employs combinatorial hypermaps as used by Gonthier in the proof of the four-color theorem. We then give a formal proof that for every simple graph containing neither K_5 nor $\mathsf{K}_{3,3}$ as a minor, there exists such a combinatorial plane embedding. Together with the formal proof of the four-color theorem, we obtain a formal proof that all graphs without K_5 and $\mathsf{K}_{3,3}$ minors are four-colorable. The development is carried out in Coq, building on the mathematical components library, the formal proof of the four-color theorem, and a general-purpose graph library developed previously.

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

Keywords and phrases Coq, MathComp, Graph-Theory, Hypermaps, Planarity

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.17

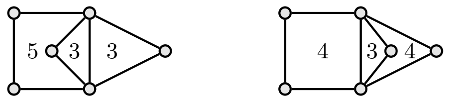
Supplementary Material All proofs in this paper have been formalized using the Coq interactive theorem prover and are included in version 0.9 of the `coq-community/graph-theory` library:
Software (Code Repository): `https://github.com/coq-community/graph-theory`
archived at `swh:1:dir:81966a945703d0e2a85fe34e920f4fe9a46b6e03`

Funding *Christian Doczkal*: This work was supported by the French *Agence Nationale de la Recherce (ANR)* within the program *Investissements d'Avenir UCA^{JEDI}* (ANR-15-IDEX-01).

1 Introduction

Despite the importance of graph theory in mathematics and computer science, formalizations of graph theory results, as opposed to verified graph algorithms, remain few and spread between different systems. This includes early works in HOL4 [3, 2] and Mizar [12], as well as some landmark results such as the formalization of the four-color theorem [10] in Coq or the formal proof of the Kepler conjecture [11] in HOL Light and Isabelle. Unfortunately, none of these has lead to the development of a widely-used general-purpose graph theory library. Since we started to develop such a general-purpose library in 2017 [6, 7, 8], there has been some renewed interest in the formalization of graph theory [14, 15]. In [8], one of the main results is a formal proof that the graphs of treewidth at most two are precisely those that do not include K_4, the complete graph with four vertices, as a minor. Other classes of graphs can also be described in terms of excluded minors, and this paper is concerned with the characterization of planar graphs as those that contain neither K_5 nor $\mathsf{K}_{3,3}$ (cf. Figure 1) as a minor. This is known as Wagner's theorem.

The textbook definition (e.g. in [5]) of a graph being planar is that there exists a drawing (or embedding) in the real plane without crossing edges. However, much of the information provided by such a drawing (e.g., the precise location of vertices or the angles at which an edge leaves a vertex) are irrelevant for most proofs about planar graphs as they can be changed almost at will by shifting or deforming the drawing. A more abstract alternative would be to take the characterization in terms of excluded minors as the definition of planarity. However, this would not provide any geometric information at all. In particular, a graph can have multiple embeddings that differ in their combinatorial properties. For instance, consider the following two drawings of the same graph:



On the left, the (inner) faces have arities 5, 3, and 3, while the arities on the right are 4, 3, and 4. Some proofs about planar graphs crucially rely on these kinds of combinatorial properties of a given plane embedding. For instance, this is the case for the proof of the four-color theorem (FCT), and the formal proof of the FCT in Coq [9, 10] represents drawings of graphs using a structure called combinatorial hypermaps [4, 17]. This representation is quite far away from the ordinary representations of graphs as a collection of vertices and edges, instead representing vertices and edges as permutations on more primitive objects called "darts".

In this paper, we use combinatorial hypermaps to represent embeddings of simple graphs, and then give a formal and constructive proof that every simple graph containing neither K_5 nor $\mathsf{K}_{3,3}$ as a minor can be represented by a planar hypermap.[1] This corresponds to one direction of Wagner's theorem, the direction that's mathematically more interesting.[2] In particular, we bridge the gap between the hypermap representation of graphs used in [9, 10] and the more standard representation of simple graphs as a finite type of vertices with an edge relation. The latter representation is used pervasively in the graph theory library we developed previously [8] and on which we base the parts of the argument that deal with structural properties like minors and separators. As it comes to hypermaps, we build on the formalization used in the proof of the four color theorem [9, 10]. Thus, as a corollary of this work, we obtain a formal proof of a "structural" four-color theorem, i.e., a proof that every graph not containing the aforementioned minors is four-colorable. This theorem does not mention hypermaps in its statement. Hence, the question whether planar hypermaps are a faithful representation of plane embeddings is secondary. What is important is that this representation allows for machine-checked proofs of interesting properties.

2 Graph Theory Preliminaries

In this section we review some standard notions from graph theory that are used in the proof of Wagner's theorem. We mostly use the conventions and terminology from previous work [8].

A *(simple) graph* is a pair $(G, -)$ where G is a finite type of objects called *vertices* and "$-$" is an irreflexive and symmetric relation on G. We use single capital letters $F, G, \ldots$ to denote graphs as well as their underlying type of vertices. That is, we write $x, y : G$ to denote that x and y are vertices of G. We also write $x{-}y$ to say that x and y are linked by an edge and $N(x) := \{y \mid x{-}y\}$ for the *open neighborhood of* x. If $x, y : G$, we write $G + xy$ for G with an additional xy-edge. For a set of vertices V, we write $G[V]$ for the subgraph induced by V, $G - V := G[\overline{V}]$ for the subgraph induced by the complement of V, and $G - x := G[\overline{\{x\}}]$ for the graph that results from deleting the vertex x (and any incident edges) from G.[3]

[1] For technical reasons, we also exclude graphs with isolated vertices (cf. Remark 21).
[2] We briefly comment on what would be required to prove the converse direction in Section 10.
[3] Technically, the vertices of $G[V]$ are dependent pairs of vertices $x : G$ and proofs $x \in V$, but we will ignore this in the mathematical presentation (cf. [8]).

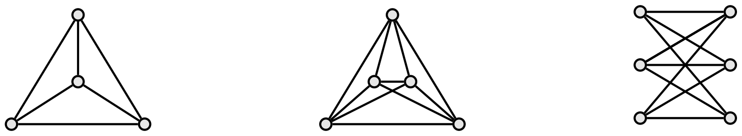


Figure 1 K_4 (left), K_5 (middle), and $\mathsf{K}_{3,3}$ (right).

We write $|G|$ for the size of G, i.e. the number of vertices of G. We write G/xy for the graph that results from merging the vertices x and y in G, which is implemented by removing the vertex y and attaching its neighbors to x. We write K_n for the complete graph with n vertices and $\mathsf{K}_{3,3}$ for the complete bipartite graph with two times three vertices (cf. Figure 1).

A *path* (in some graph G) is a nonempty sequence of vertices with subsequent vertices linked by the edge relation, and an xy-path is a path starting at x and ending at y. A *cycle* is an xy-path for some $x, y : G$ such that $x{-}y$. If π_1 and π_2 are paths, we write $\pi_1 +\!\!+ \pi_2$ for their concatenation. A set of vertices A is *connected*, if any two vertices in A are connected by a path contained in A. Two sets of vertices A and B are *neighboring*, if there exist vertices $x \in A$ and $y \in B$ such that $x{-}y$.

A set of vertices S *separates* x and y, if $x, y \notin S$ and every xy-path contains a vertex from S. A set that separates any two vertices, i.e. whose removal would disconnect the graph, is called a *(vertex) separator*. In particular, $\emptyset$ is a separator iff G has multiple disconnected components. A graph G is k-connected if $k < |G|$ and every separator has size at least k. In particular, K_{k+1} is k-connected, since there are no separators in a complete graph. A *separation* of G is a pair (V_1, V_2) of sets of vertices such that $V_1 \cup V_2$ covers G and there is no edge from $\overline{V_1}$ to $\overline{V_2}$. A separation (V_1, V_2) is *proper*, if both $\overline{V_1}$ and $\overline{V_2}$ are nonempty.

▶ **Fact 1.** *Let G be a simple graph. Every separator S of G can be extended into a proper separation (V_1, V_2) of G such that $S = V_1 \cap V_2$.*

We are interested in the characterization of planar graphs through excluded minors. Intuitively, a minor of a graph is a graph that can be obtained from the original graph through a series of edge deletions, vertex deletions, and edge contractions. Following our previous work [8], we define the minor relation using functions we call minor maps:

▶ **Definition 2.** *Let G and H be simple graphs. A function $\phi : H \to 2^G$ is called a* minor map *if:*

M1. *$\phi(x)$ is nonempty and connected for all $x : H$,*

M2. *$\phi(x) \cap \phi(y) = \emptyset$ whenever $x \neq y$ for all $x, y : H$.*

M3. *$\phi(x)$ neighbors $\phi(y)$ for all $x, y : H$ such that $x{-}y$.*

H is a minor *of G, written $H \prec G$ if there exists a minor map $\phi : H \to 2^G$.*

If $\phi : H \to 2^G$ is a minor map, then $\phi(x)$ is the set of vertices being collapsed to x (by contracting all the edges in $\phi(x)$) when exhibiting H as a minor of G.

▶ **Fact 3.** *$\prec$ is transitive.*

▶ **Definition 4.** *A graph G is called H-*free*, if H is not a minor of G.*

Note that if G is H-free, then, by transitivity, so is every minor of G. Also note that if $x{-}y$, then G/xy corresponds to an edge contraction. Hence, we have the following lemma.

▶ **Lemma 5.** *If $x{-}y$, then $G/xy \prec G$*

It is easy to see that $G[V] \prec G$, for any set V of vertices of G, and thus $G[V]$ is H-free whenever G is. However, when V is one of the two sides of a separation arising from a separator $\{x, y\}$, we can even add an xy-edge, as shown below.

▶ **Lemma 6.** *Let (V_1, V_2) be a proper separation of G with $V_1 \cap V_2 = \{x, y\}$ with $x \neq y$ and $\{x, y\}$ a smallest separator. Then every minor of $(G + xy)[V_1]$ is also a minor of G.*

Proof. If the xy-edge is used to justify $H \prec (G+xy)[V_1]$ for some H, the xy-edge can always be replaced by a path through $V_2 \setminus V_1$, which is not otherwise needed to establish $H \prec G$. ◀

3 Wagner's Theorem

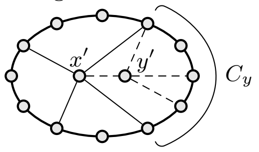
Before we turn to the formal proof of Wagner's theorem using combinatorial hypermaps, we first sketch the proof relying on an informal notion of plane embedding (i.e., drawings of the graph without crossing edges), leaving the technical details of the modeling to Section 6.

The proof of Wagner's theorem consists of two parts. The main induction deals with the case for 3-connected graphs. This is then extended to the general case though a number of comparatively straightforward combinations of plane embeddings for subgraphs. Below, we sketch the two arguments, including forward references to two types of lemmas: those that are interesting from a mathematical point of view (marked with "$\star$") and those that depend on the modeling of plane embeddings using hypermaps (marked with "$\dagger$"). The proofs are inspired by those in [1, 5].

▶ **Proposition 7.** *Let G be 3-connected, K_5-free, and $\mathsf{K}_{3,3}$-free. Then G can be embedded in the plane.*

Proof sketch. The proof proceeds by induction on $|G|$.

1. Since G is 3-connected, we have $4 \leq |G|$. If $|G| = 4$, then G is K_4, which can easily be embedded in the plane (Figure 1, Proposition $22^{\dagger}$). Hence, we can assume $5 \leq |G|$.
2. Thus, we obtain $x, y : G$ such that $x{-}y$ and G/xy is again 3-connected (Theorem $11^{\star}$).
3. Since $|G/xy| < |G|$, we obtain a plane embedding for G/xy by induction (Lemma 5). Let v_{xy} be the vertex resulting from the contraction of the xy-edge and set $H := G/xy - v_{xy}$. Let X (resp. Y) be the set of vertices in H that are neighbors of x (resp. y) in G.
4. Since G/xy is 3-connected and since all vertices in $X \cup Y$ are neighbors of v_{xy}, removing v_{xy} and all incident edges form the plane embedding of G/xy yields a plane embedding $\hat{H}$ of H with a face whose boundary contains all vertices from X and Y (Lemma $28^{\star\dagger}$).
5. Since G/xy is 3-connected, we have that H is 2-connected. Hence, the face of $\hat{H}$ whose boundary contains X and Y is bounded by a (duplicate-free) cycle C (Theorem $25^{\star\dagger}$).
6. Splitting C at the elements of X yields a number of segments where every segment overlaps with each of its two neighboring segments in exactly one element of X (unless there are only two segments). Since $\mathsf{K}_5 \not\prec G$ and $\mathsf{K}_{3,3} \not\prec G$, all elements of Y must be contained in one of the segments of C; call this segment C_y (Lemma $12^{\star}$)
7. Adding a vertex x' to $\hat{H}$ inside C and making it adjacent to all vertices in X yields a graph with an embedding that has a face containing x' and C_y. Thus, we can place a vertex y' within this face and add edges to x' and all vertices in Y as shown below:



This yields a plane embedding of G. ◀

It remains to take care of the cases where G is not 3-connected.

▶ **Theorem 8.** *Let G be K_5-free, and $\mathsf{K}_{3,3}$-free. Then G can be embedded in the real plane.*

Proof. By induction on $|G|$. By Propositions 7 and 22, we can assume that $5 \leq |G|$ and that G has a smallest separator S with $|S| \leq 2$. We obtain a proper separation (V_1, V_2) with $V_1 \cap V_2 = S$. If $S = \{x, y\}$, we set $H := G + xy$ and have that neither $H[V_1]$ nor $H[V_2]$ contains K_5 or $\mathsf{K}_{3,3}$ as a minor (Lemma 6), allowing us to obtain plane embeddings of $H[V_1]$ and $H[V_2]$ by induction. Due to the added xy-edge, both embeddings must have a face with x and y adjacent on the boundary of some face. Without loss of generality, we can assume that this is the (unbounded) outer face. By stretching and scaling, we can "glue" together the two embeddings along these outer edges, obtaining a plane embedding of H (Lemma 30*). Removing the xy edge (or keeping it if it was present in G), provides a plane embedding of G. The cases for $S = \emptyset$ and $S = \{x\}$ are similar, but do not require the use of a "marker" edge. ◀

Note that the proof of Theorem 8 makes reference to intuitive operations such as stretching and scaling. In particular, the fact that one can turn an arbitrary face into the outer face is usually argued using a stereographic projection to the sphere and back to the plane [1]. All of these will be no-ops for our representation of plane embeddings using hypermaps.

4 The Combinatorial Part

This section is concerned with the purely combinatorial part of the proof of Proposition 7, justifying steps (2) and (6). The former amounts to locating an edge in a 3-connected graph such that contracting this edge yields a smaller 3-connected graph. The latter is about justifying (using the names from the proof of Proposition 7) that in the cycle C all the neighbors of y are contained in a segment spanned by two successive neighbors of x. This is the part of the proof where assumptions of K_5-freeness and $\mathsf{K}_{3,3}$-freeness are used. Both arguments are combinatorial in the sense that neither argument makes any reference to plane embeddings.

For step (2), the argument is based on smallest separators, and we repeatedly use the following property:

▶ **Proposition 9.** *If S is a smallest separator of G, then S neighbors every maximal component of $G - S$.*

Recall that G/xy is implemented by removing y and updating the edge relation accordingly.

▶ **Lemma 10.** *Let G be 3-connected with $5 \leq |G|$, and let $x, y : G$ such that $x{-}y$ and G/xy is not 3-connected. Then there exists some $z : G$ such that $\{x, y, z\}$ is a separator.*

Proof. Since G is 3-connected, we have that G/xy is 2-connected. Moreover, G/xy is not 3-connected by assumption. Hence, G/xy has a smallest separator S with $|S| = 2$. We have that $x \in S$, because otherwise S would be a 2-separator of G. Thus, $S = \{x, z\}$ for some z, and $\{x, y, z\}$ is a separator of G. ◀

▶ **Theorem 11.** *If G is 3-connected and $5 \leq |G|$, then there exists an xy-edge such that G/xy is 3-connected.*

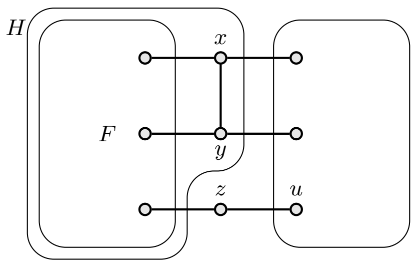


Figure 2 Objects from the proof of Theorem 11 (cf. [1, Theorem 9.10]).

Proof. Assume the theorem does not hold, i.e., assume that G/xy is not 3-connected for all $x, y : G$ such that $x{-}y$. We obtain a contradiction as follows:

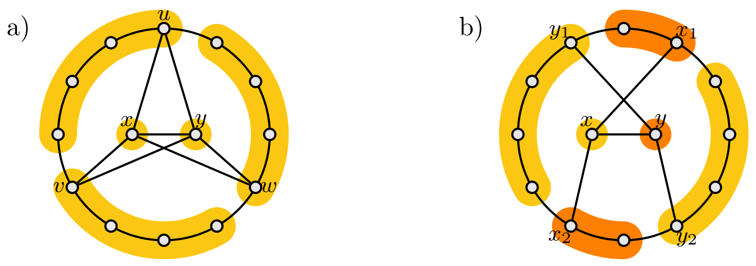
By Lemma 10, every xy-edge can be extended to a separator $\{x, y, z\}$. Choose x, y, z, and F such that $x{-}y$, $\{x, y, z\}$ is a separator, F is connected and disjoint from $\{x, y, z\}$, and with $|F|$ maximal for all possible choices of x, y, z and F. Now set $H := F \cup \{x, y\}$. Since G is 3-connected, $\{x, y, z\}$ is indeed a smallest separator of G. Thus, x, y, and z are pairwise distinct and by Proposition 9 there exists some vertex $u \notin H$ such that $z{-}u$ (cf. Figure 2). Let v such that $\{z, u, v\}$ is a separator (Lemma 10). Now it suffices to show that $H \setminus \{v\}$ is connected, because this yields a component larger than F, contradicting the choice of F. If $v \notin H$ this is trivial and if $v \in \{x, y\}$, this follows since $\{x, y, z\}$ is a smallest separator. (Proposition 9 ensures that both x and y have neighbors in F.) Hence, we can assume $v \in F$. Now if $H \setminus \{v\}$ was disconnected, then there would be some vertex w such that every xw-path in H passes through v. However, since F is maximal and therefore has no outgoing edges other than those to x, y, and z, this would entail that $\{v, z\}$ is a separator (separating x from w), contradicting the assumption that G is 3-connected. ◀

We remark that, just like all the other results presented in this paper, the proof of Theorem 11 does not require any classical axioms. The conclusion of the theorem involves only decidable predicates and quantifiers over finite domains (i.e., the vertices of G), and these behave classically. Similarly, there are only finitely many choices for x, y, z, and F, so we can easily obtain a combination where $|F|$ is maximal among all possible choices.

In order to formally state the lemma justifying step (6) of Proposition 7, we need to introduce some operations on duplicate-free lists viewed as cycles. Let T be some type and let C be a duplicate free list over T. For $x \in C$, we write $\mathsf{next}\, C\, x$ for the element following x in C or the first element of C if x is at the very end. For $x, y \in C$ with $x \neq y$, we write $\mathsf{arc}\, C\, x\, y$ for the part of C (seen as a cycle) that starts at x and ends right before y. In particular, the results of $\mathsf{next}\, C\, x$ and $\mathsf{arc}\, C\, x\, y$ are invariant under cyclic shifts of C.

▶ **Lemma 12.** *Let G be a simple, K_5-free, and $\mathsf{K}_{3,3}$-free graph, let $x, y : G$ such that $x{-}y$ and let C be a duplicate-free cycle in G containing neither x nor y. Let X be the sub-sequence of C containing $N(x)$ and let Y be the sub-sequence of C containing $N(y)$. If X and Y each contain at least two vertices, then there exists some vertex $z \in X$ such that $Y \subseteq \mathsf{arc}\, C\, z\, (\mathsf{next}\, X\, z) \cup \{\mathsf{next}\, X\, z\}$.*

Proof. We first show that there are at most two vertices in $X \cap Y$. Assume, for the sake of contradiction, three distinct vertices $u, v, w \in X \cap Y$. W.l.o.g., we can assume that $[u, v, w]$ is a sub-cycle of C. Hence, we obtain K_5 as a minor of G by collapsing by mapping the vertices of K_5 to the sets $\{x\}$, $\{y\}$, $\mathsf{arc}\, C\, u\, v$, $\mathsf{arc}\, C\, v\, w$, and $\mathsf{arc}\, C\, w\, u$ as shown in Figure 3(a), contradicting the assumption that G is K_5-free.



■ **Figure 3** Obtaining K_5 (left) and $\mathsf{K}_{3,3}$ (right) as minors in Lemma 12.

Next, we show that there cannot be a sub-cycle $[x_1, y_1, x_2, y_2]$ of C such that $\{x_1, x_2\} \subseteq X$ and $\{y_1, y_2\} \subseteq Y$. If such a sub-cycle were to exist, we could exhibit $\mathsf{K}_{3,3}$ as a minor of G by mapping the three pairwise-independent left-hand-side vertices to $\{x\}$, $\mathsf{arc}\, C\, y_1\, x_2$, and $\mathsf{arc}\, C\, y_2\, x_1$ and the three right hand side vertices to $\{y\}$, $\mathsf{arc}\, C\, x_1\, y_1$, and $\mathsf{arc}\, C\, x_2\, y_2$, contradicting $\mathsf{K}_{3,3}$-freeness of G (cf. Figure 3(b)).

Now, assume that the theorem does not hold, i.e., assume that for every $x' \in X$, there exists some $y' \in Y$ such that $y' \notin \mathsf{arc}\, C\, x'(\mathsf{next}\, X\, x') \cup \{\mathsf{next}\, X\, x'\}$. We consider two cases:

- If $Y \subseteq X$, we have that $Y = [y_1, y_2]$ for two distinct vertices y_1 and y_2. Now $\mathsf{arc}\, C\, y_1\, y_2$ must contain some vertex $x_2 \in X \setminus \{y_1, y_2\}$, for otherwise $\mathsf{next}\, X\ y_1 = y_2$ and both y_1 and y_2 are contained in $\mathsf{arc}\, C\, y_1\, y_2 \cup \{y_2\}$. By symmetry, we also have that $\mathsf{arc}\, C\, y_2\, y_1$ must contain some $x_1 \in X \setminus \{y_1, y_2\}$. However, then $[x_1, y_1, x_2, y_2]$ is an alternating subcycle, whose existence we excluded above. Contradiction.
- Otherwise, there exists some $y_1 \in Y \setminus X$. Let x_1 such that $y_1 \in \mathsf{arc}\, C\, x_1\, (\mathsf{next}\, X\, x_1)$ and set $x_2 := \mathsf{next}\, X\, x_1$. By assumption, there must be some $y_2 \in Y$ with $y_2 \notin \mathsf{arc}\, C\, x_1\, x_2 \cup \{x_2\}$. Hence, $[x_1, y_1, x_2, y_2]$ is again an excluded alternating subcycle. Contradiction. ◀

Lemma 12 can be considered to be the combinatorial core argument underlying Wagner's theorem. It is the place where absence of certain substructures (i.e., the minors K_5 and $\mathsf{K}_{3,3}$) is turned into a positive statement that allows reversing the contraction of the xy-edge. We remark that while the arc construction was already present in mathcomp, splitting a cycle along a subcycle required a plethora of additional lemmas about arcs and cycles.

5 Combinatorial Hypermaps

We now turn towards the modeling of embeddings in the plane using combinatorial hypermaps. In this section we briefly review hypermaps and their most important properties. The presentation follows [9], because the formal development underpinning this part is based on the formal proof of the four-color theorem presented there. Consequently, none of the results in this section are new.

▶ **Definition 13.** *A* (combinatorial) hypermap *is a tuple $\langle D, e, n, f\rangle$ where D is a finite type, and $e, n, f : D \to D$ such that $n \circ f \circ e \equiv \mathrm{id}_D$. The elements of D are referred to as* darts.

The condition $n \circ f \circ e \equiv \mathrm{id}_D$ ensures that the functions e, n, and f are bijective (i.e. permutations on D). In particular, any two of the permutations determine the third. Each of the permutations partitions the type D into a number of cycles and these cycles are used

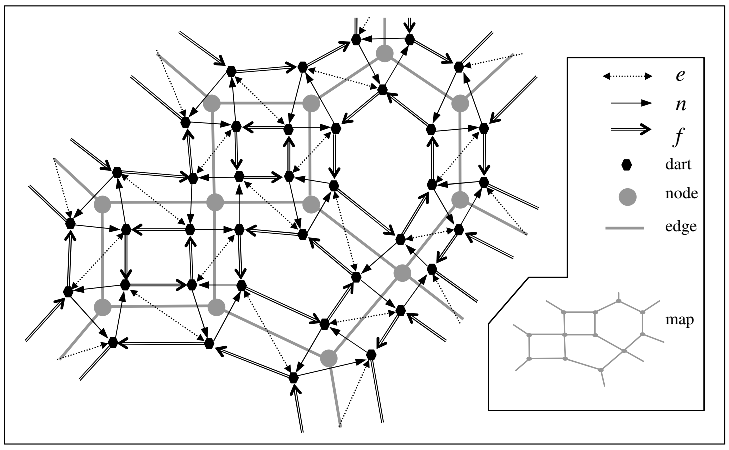


Figure 4 A hypermap. (Reprinted with permission from [9], ©2005 Georges Gonthier).

to represent the edges, nodes[4], and faces of graphs. That is, a hypermap $\langle D, e, n, f\rangle$ can be seen as describing a graph embedded on a surface (not necessarily the plane) as follows (cf. Figure 4):

- every n-cycle represents a node of the graph, listing incident edges in counterclockwise order.
- every e-cycle represents an edge of the graph, linking the nodes (i.e., n-cycles) it intersects.
- every f-cycle represents a face, listing in counterclockwise order one dart from every node on the boundary of the face.

Even though one of the three permutations is technically redundant, keeping it makes the definition completely symmetric and facilitates symmetry reasoning. In particular, if $\langle D, e, n, f\rangle$ is a hypermap, then so are $\langle D, f, e, n\rangle$ and $\langle D, n, f, e\rangle$. As we do for graphs, we will usually use the same letter for a hypermap and its underlying type of darts.

▶ **Definition 14.** *Let $\langle D, e, n, f\rangle$ be a hypermap.*

- *D is called* plain *if every e-cycle has size 2.*
- *D is called* loopless *if x and $e(x)$ belong to different n-cycles for all $x : D$.*
- *D is called* simple *if two n-cycles are linked by at most one e-cycle.*

Plain hypermaps correspond to graphs where every edge is adjacent to two vertices, i.e. graphs without hyperedges. As we will make precise later, plain loopless simple hypermaps correspond to simple graphs, i.e., graphs without self loops and with at most one edge between two vertices. The (partial) hypermap in Figure 4 satisfies all three properties, as will most of the hypermaps we will be dealing with.

We fix a hypermap $\langle D, e, n, f\rangle$ for the rest of the section. Moreover, we will use the same letter D for the hypermap as a whole as well as the underlying type of darts.

The number of "holes" that would be needed in a surface in order to embed a given hypermap in it can be computed using the Euler characteristic.

▶ **Definition 15** (Genus). *The* genus *of D is $((2C + |D|) - (E + N + F))/2$ where C is the number of connected components of $e \cup n \cup f$ (interpreting the functions as functional relations) and E, N, and F are the number of cycles of e, n, and f respectively. A map of genus 0, i.e., a map satisfying the equation $E + N + F = 2C + |D|$ is called* planar.

[4] In line with the terminology of [9, 10], we say "node" when referring to an n-cycle of a hypermap. In line with [8], we continue to use "vertex" when referring to vertices of simple graphs.

The following general properties of hypermaps are established in [9].

▶ **Proposition 16.** $E + N + F \leq 2C + |D|$.

▶ **Proposition 17.** $(2C + |D|) - (E + N + F)$ *is even.*

Proposition 16 implies that the (natural number) subtraction in Definition 15 is never truncating and Proposition 17 implies that the division in the genus formula is always an integer division without remainder.

For our use of hypermaps as representations of embeddings in the plane, we will need to modify hypermaps and prove that these modifications preserve planarity. Directly proving that an operation such as adding an edge across a face preserves the genus of the hypermap can be cumbersome. It is often simpler to express the operation in terms of more atomic planarity-preserving operations. The most important of these operations are the *Walkup* [16, 18] operations.

▶ **Definition 18.** *For $x : D$,* WalkupE x *is the hypermap where x has been removed by skipping over x in the n and f permutations and adapting e as necessary. Similarly,* WalkupN x *(resp.* WalkupF x*) are the hypermaps where n (resp. f) is the permutation being adapted after suppressing x from the other two.*

As shown in [9], the Walkup operations never increase the genus of a hypermap and, in particular, always preserve planarity. In addition, the Walkup operations can be shown to preserve the genus in many circumstances, allowing us to prove preservation of planarity for operations that extend the hypermap by expressing them as inverse Walkup operations. Thus, the characterization of planarity in terms of Euler's formula combined with expressing operations as combinations of Walkup operations provides for an easy means of proving that various operations on hypermaps preserve planarity.

In addition to showing that certain operations preserve planarity, we also need to establish some properties of planar hypermaps in general. For instance, we need to show that in every two-connected plane graph, all faces are bounded by (duplicate free) cycles (step (5)). For the topological model of plane graphs, this property is established using the Jordan curve theorem (JCT), which states that every closed simple curve divides the plain into an "inside" and an "outside". Since hypermaps make no reference to the real plane, we could not use this theorem, even if it was available in Coq. However, the essence of the application of the JCT to plane graphs is captured by the following theorem on hypermaps:

▶ **Theorem 19** (Jordan curve theorem for hypermaps [9, 10])**.** *Let $\langle D, e, n, f \rangle$ be a hypermap. Then D is planar iff there do not exist distinct darts x, y and a duplicate-free $(n^{-1} \cup f)$-path from x to $n(y)$ visiting y before $n(x)$ (with $y = n(x)$ being allowed).*

Note that when talking about hypermaps, an $(n^{-1} \cup f)$-path is a path in the relation $(n^{-1} \cup f)$. This is to be contrasted with the notion of an xy-path in a simple graph, where we mention the endpoints and leave the relation implicit. Paths in the relation $(n^{-1} \cup f)$ are called *contour paths*, because they go around the outside of a group of faces (cf. Figure 4). Thus, a contour cycle in a planar map corresponds to a closed curve. The Jordan curve theorem for hypermaps establishes that in a planar hypermap there cannot be a contour path starting at the inside of a contour cycle and finishing on the outside without otherwise intersecting the cycle. In the theorem above, the contour cycle and the contour path are spliced together in order to obtain a simpler statement (cf. [9, 10]).

6 Combinatorial Embeddings

In this section, we make precise what it means for a hypermap to represent an embedding of a graph on some surface. We first introduce some additional notation. For a relation $r : D \to D \to \mathbb{B}$ over a finite type D (e.g., the darts of a hypermap) we write r^* for the reflexive transitive closure of r and $r^*(x)$ for the set $\{y \mid r^*\, x\, y\}$. In particular, we write f^* for the transitive closure of a function $f : D \to D$ seen as the relation $\lambda x\, y.\, f x = y$. Note that, because D is finite, f^* is symmetric if f is injective, as is the case for the permutations comprising hypermaps. For a hypermap $\langle D, e, n, f\rangle$, we call two darts x and y *adjacent*, written $\mathsf{adjn}\, x\, y$, if their respective n-cycles are linked by an e-cycle (i.e., if there exists some dart z such that $n^*\, x\, z$ and $n^*\, y\, (e\, z)$).

▶ **Definition 20.** *Let G be a simple graph and let $\langle D, e, n, f\rangle$ be a plain hypermap. We call a function $g : D \to G$ a* (combinatorial) embedding *of G if it satisfies the following properties:*
1. *g is surjective*
2. *$n^*\, x\, y$ iff $g(x) = g(y)$.*
3. *$\mathsf{adjn}\, x\, y$ iff $g(x) - g(y)$.*

An embedding where D is planar, is called a plane embedding*, and an embedding where D is simple is called a* simple embedding*. A graph together with a plane embedding is called a* plane graph*.*

Note that, even though we refer to g as an embedding of a graph, the function maps darts of the hypermap to vertices of the graph. This makes it easier to state the required properties. Surjectivity of g ensures that D represents the whole graph. Condition (2) ensures that the node cycles of D are in one-to-one correspondence to the vertices of G, and condition (3) ensures that adjacent node cycles correspond to adjacent vertices of G. Note that we do *not* require that the hypermap underlying an embedding is simple, i.e., we permit multiple parallel edges. This reduces the number of conditions to check when constructing plane embeddings. Parallel edges can always be removed, obtaining a simple embedding where needed.

▶ Remark 21. Definition 20 abstracts not only from properties that can be changed by continuously deforming the plane, it also does not single out a face as the "outer" face or specify the relationships between the embeddings of disconnected components of a graph, i.e., we do not embed one component in a particular face of the embedding of another component. Consequently, Definition 20 corresponds more to embedding every component of the graph on its own sphere rather then embedding all components together in the plane. Moreover, the degenerate case of a component consisting of a single isolated vertex cannot be represented by hypermaps, because every dart of an n-cycle must also be part of an e-cycle. This is not really an issue: isolated vertices are components without internal structure, and there would be nothing to learn about such vertices from a combinatorial embedding.

With Definition 20 in place, we can now justify step (1) of the proof of Proposition 7, i.e., obtain a plane embedding for K_4. The graph K_4 has 6 edges, so we take the 12-element type $I_{12} := \Sigma n : \mathbb{N}.\, n < 12$ as the type of darts and provide the three permutations as well as a mapping from I_{12} to the vertices of K_4. Since both K_4 and its embedding are concrete objects, we can use the depth-first search algorithm from mathcomp to compute the genus of the map and check the correctness of the embedding. This requires brute-forcing various quantifiers, which causes no problems due to the small size of their domain (i.e. 4 or 12). Thus, we obtain:

▶ **Proposition 22.** *There exists a plane embedding for K_4.*

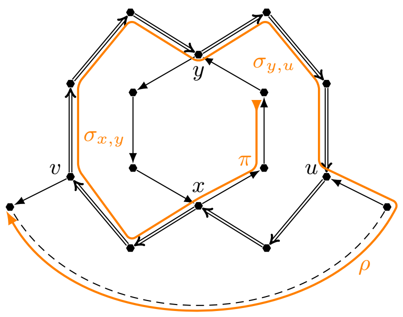


Figure 5 Moebious path from the proof of Lemma 26.

We also show that $\mathsf{K}_{3,3}$ does not have a plane embedding. While this result does not contribute to the main result of this paper, it serves as an example of how Definition 20 and some of the properties described in Section 5 fit together.

▶ **Proposition 23.** *There exists no plane embedding for* $\mathsf{K}_{3,3}$.

Proof. Assume there was an embedding $g : D \to \mathsf{K}_{3,3}$ with D of genus 0. Without loss of generality, we can assume that D is simple. Thus, we have $N = 6$, $E = 9$, $|D| = 2 * E = 18$, and $C = 1$. By the definition of genus, it suffices to show $(5 - F)/2 > 0$ to obtain a contradiction. Since every vertex of $\mathsf{K}_{3,3}$ has at least two neighbors and since D is simple, every face-cycle must use at least 3 darts. Moreover, $\mathsf{K}_{3,3}$ has no odd-length cycles, so every face-cycle of D must indeed use at last 4 darts. Thus $F \leq 4$, since $|D| = 18$. Finally, $F \neq 4$ since the division in the genus formula is always without remainder (Proposition 17). ◀

We now come to the main result of this section, namely that the faces of 2-connected plane graphs are bounded by irredundant cycles. In order to state this property precisely, we define a notion of face for simple graphs relative to an embedding.

▶ **Definition 24.** *If* $g : \langle D, e, n, f\rangle \to G$ *is an embedding, a* face *of* G *under* g *is a cycle in* G *that can be obtained as the image of an* f*-cycle of* D *of under* g.

The theorem we want to prove is the following.

▶ **Theorem 25.** *Let* g *be a plane embedding of a 2-connected graph* G. *Then all the faces under* g *are duplicate-free cycles.*

Before we can prove this theorem, we first need to prove the underlying property on hypermaps. This is where the Jordan curve theorem for hypermaps (Theorem 19) is used.

▶ **Lemma 26.** *Let* $\langle D, e, n, f\rangle$ *be a plain loopless planar hypermap such that for all darts* x, y, z *with* $x, y \notin n^*(z)$ *there exists an* $(n^{-1} \cup f)$*-path from* x *to* y *not containing any dart in* $n^*(z)$. *Then there do not exist distinct darts* x, y *such that* $n^*\, x\, y$ *and* $f^*\, x\, y$.

Proof. Assume there exist $x \neq y$ such that $n^*\, x\, y$ and $f^*\, x\, y$. We show that this contradicts the planarity of G. Without loss of generality, we obtain a duplicate-free n^{-1}-path from y to x whose interior π is disjoint from $f^*(x)$ (We make n^{-1}-steps starting at y and replace x with the first encountered dart in $f^*(y)$). Now we can split the f-cycle containing x and y into two semi-cycles, one from x to y and another from y to x. We call their respective interiors (which are both disjoint from $\pi \cup \{x, y\}$) $\sigma_{x,y}$ and $\sigma_{y,x}$. By assumption, we can obtain $(n^{-1} \cup f)$-paths avoiding $n^*(x)$ and connecting any two darts outside of $n^*(x)$. Thus, we obtain darts $u \in \sigma_{y,x}$ and $v \in \sigma_{x,y}$ and a duplicate-free $(n^{-1} \cup f)$-path from u to v disjoint from the n-cycle containing both x and y whose interior we call ρ. Without loss of generality, we can assume that ρ is also disjoint from $\sigma_{x,y}$ and $\sigma_{y,x}$ (otherwise we shorten ρ, possibly changing the choice of u and v). Finally set $\sigma_{y,u}$ to be the part of $\sigma_{y,x}$ before u. Thus, we have that $m := \pi +\!\!+ [x] +\!\!+ \sigma_{x,y} +\!\!+ [y] +\!\!+ \sigma_{y,u} +\!\!+ u +\!\!+ \rho$ is a duplicate-free $(n^{-1} \cup f)$-path. Moreover, the fist dart in m is $n^{-1}(y)$ (which could be x) and (since $\sigma_{x,y}$ is an f-path) the last dart is $n(v)$ (cf. Figure 5). Since m visits v (which is in $\sigma_{x,y}$) before y, m is a "Moebius contour" and Theorem 19 applies, contradicting the planarity of D. ◀

Now we can prove Theorem 25, justifying step (5) of the proof of Proposition 7.

Proof of Theorem 25. Let G be 2-connected and let $g : \langle D, e, n, f\rangle \to G$ be a plane embedding. Thus D is plain, loopless, and planar. Let s be a face of G under g arising as the image of some f-cycle in D. It suffices to show that all the darts in this f-cycle belong to different n-cycles. Since G is 2-connected, all vertices different from z can be connected using paths that avoid z. These paths can be mapped to $(n^{-1} \cup f)$-paths in D. Hence, Lemma 26 applies, finishing the proof. ◀

The proof of Theorem 25 exhibits a pattern that is repeated for various lemmas about plane embeddings: we first show the underlying lemma for hypermaps and then lift the property to the language of simple graphs and plane embeddings in order to use them in the proofs of Proposition 7 and Theorem 8.

7 Modifying Plane Embeddings

We now describe the operations on plane embeddings and their underlying hypermaps that are required to carry out steps (4) and (7) of the proof of Proposition 7. That is, we show how to remove a vertex from a plane embedding, obtaining a face containing all neighbors of the removed vertex, and we show how to add a vertex, connecting it to an arbitrary subsequence of a face-cycle.

We begin by showing that every subgraph of a plane graph has a plane embedding. While this is intuitively obvious, the precise argument deserves some mention. Again, we need some notation to express the underlying lemma about hypermaps:

Let T be a finite type and let $f : T \to T$ be an injective function and let P be a subset of T. We write ΣP for the *type* of elements of P, i.e., the type of dependent pairs $\Sigma x : T.\, x \in P$. We define $\mathsf{skip}_P f : T \to T$ to be the function which for every $x : T$ returns $f^{n+1}(x)$ for the least n such that $f^{n+1}(x) \in P$ if such an n exists and x otherwise. Such an n always exists when $x \in P$, so $\mathsf{skip}_P f$ can also be seen as a function $\Sigma P \to \Sigma P$. Finally, we write $f \equiv g$, to denote that two functions agree on all arguments.

▶ **Lemma 27.** *Let $\langle D, e, n, f\rangle$ be a hypermap, let $P \subseteq D$, and let $\langle \Sigma P, e', n', f'\rangle$ be another hypermap such that $e' \equiv \mathsf{skip}_P e$ and $n' \equiv \mathsf{skip}_P n$. Then $\mathsf{genus}\, \langle \Sigma P, e', n', f'\rangle \leq \mathsf{genus}\, \langle D, e, n, f\rangle$.*

Proof. By induction on $|D|$. If P is the full set, then the two hypermaps are isomorphic and therefore have the same genus. Thus, we can assume there exists some $z \notin P$. Let $H := \mathsf{WalkupF}\, z$. Since the Walkup operation does not increase the genus, it suffices to show $\mathsf{genus}\, \langle \Sigma P, e', n', f'\rangle \leq \mathsf{genus}\, H$. This follows by induction hypothesis since H is defined by skipping over z in the edge and node permutations and, therefore, $\langle \Sigma P, e', n', f'\rangle$ can be obtained from H, again up to isomorphism, by skipping over the remaining elements of $\overline{P}$. ◀

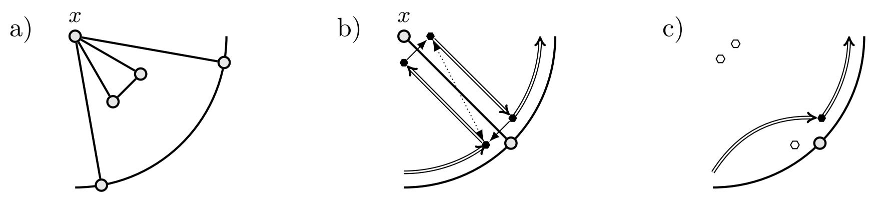


Figure 6 Removing a vertex from a 2-connected plane graph.

Note that Lemma 27 applies to any hypermap, not just plain ones. This small generalization allows us to prove the lemma by induction, removing a single dart at a time. This would not work with plain maps, which always have an even number of darts. Also note that the proof of the lemma above makes extensive use of isomorphisms for hypermaps, a notion that is not defined in the formal development of the four-color theorem, where only an equivalence for hypermaps with the same type of darts is defined. This turned out to be too restrictive for our purposes. As we do for other types of graphs [8], we define isomorphisms between hypermaps as bijections on the underlying type of darts that preserve the three permutations.

▶ **Lemma 28.** *Let G be a 2-connected graph with vertex x and let g be a plane embedding. Then there exists a plane embedding g' for $G - x$ and a face of g' containing all vertices in $N(x)$.*

Proof. Let $D = \langle D, e, n, f\rangle$ be the hypermap underlying g, and $d_x : D$ such that $g(d_x) = x$. Without loss of generality, we can assume that D is a simple hypermap. We set $P := \overline{e^*(n^*(d_x))}$ and set $D' = \langle \Sigma P, \mathsf{skip}_P\, e, \mathsf{skip}_P\, n, f'\rangle$ for some suitable f', which amounts to removing all e-cycles intersecting $n^*(d_x)$. D' is clearly plain, and by Lemma 27 D' is also planar. Since $x \notin g(P)$, the restriction of g to D' yields a plane embedding $g' : D' \to (G - x)$. It remains to show that g' has a face containing $N(x)$. First, 2-connectedness of G rules out the scenario depicted in Figure 6(a), where removing x would disconnect the graph. Moreover, it ensures that every n-cycle (in D) has at least size two. Together with D being simple, this ensures that no n-cycle other than the one for x vanishes and that f' needs to skip over at most one removed dart at a time (Figure 6(b-c)), allowing us to give a simple explicit definition of f': $f'(z) := \text{if } f(z) \in P \text{ then } f(z) \text{ else } n^{-1}(f(z))$

Moreover, we have that for all $d \in n^*(d_x)$, $f(d)$ is in P and on the same (original) n-cycle as $e(d)$, meaning every dart $f(d)$ represents a neighbor of x. Thus, it suffices to show $f'^*\,(f d_1)(f d_2)$ for $d_1, d_2 \in n^*(d_x)$. We prove this claim by induction on the n-path from d_1 to d_2, reducing the problem to showing $f'^*\,(f\,d)(f(n\,d))$ for $d \in n^*(d_x)$. Since D is simple, the f-orbit of $f(d)$ as length at least 3 and therefore the shape $[f(d)] +\!\!+\, o +\!\!+ [(e(n(d)), d]$. Moreover, since D is an embedding for a 2-connected graph, we can use Lemma 26 to show that $e(n(d))$ and d are the only darts from the f-orbit of $f(d)$ that are not in P. Thus, the claim follows from the definition of f' since $n^{-1}(e(n(d)) = f(n(d))$. ◀

Note that the proof above uses Lemma 26 for the second time. When we use the lemma in step (4) of the proof of Proposition 7, we apply it to the 3-connected graph G/xy, exploiting that $G/xy - v_{xy}$ is still 2-connected, which in turn allows us to argue that the obtained face containing all the neighbors is bounded by a duplicate-free cycle (cf. step (5) and Theorem 25).

Finally, we justify step (7) of Proposition 7, which amounts to two applications of the lemma below, where $G + (z, A)$ is a the simple graph G extended with a new vertex z which is made adjacent to all vertices in the set A.

▶ **Lemma 29.** *Let $g : D \to G$ be a plane embedding, let $[x] +\!\!+ p +\!\!+ [y] +\!\!+ q$ be a face of g, and let $\{x, y\} \subseteq A \subseteq \{x, y\} \cup p$. Then there exists a plane embedding of $G + (z, A)$ with a face $[x, z, y] +\!\!+ q$.*

Proof. We first show that for every face $[u] +\!\!+ s$ under some embedding, one can add a single vertex v and obtain an embedding of $G + (v, \{u\})$ with face $[u, v, u] +\!\!+ s$. Moreover, one can always add an edge across a face, splitting a face $[u] +\!\!+ s_1 +\!\!+ [v] +\!\!+ s_2$ into two faces $[v, u] +\!\!+ s_1$ and $[u, v] +\!\!+ s_2$. In each case, we show that the operation can be reversed by a genus-preserving double Walkup operation, showing that the initial addition preserves the genus. The claim then follows by first adding z and the xz-edge and then adding the remaining edges in the order in which they appear in $p +\!\!+ [y]$. ◀

This finishes the justification for the individual steps of the proof of Proposition 7. We remark that Lemmas 28 and 29 are "lossy" in that we do not prove that the untouched part of the embedding remains the same. This would only clutter the statements and is not needed for our purposes. Should the need arise, it would be straightforward to turn the underlying constructions into definitions and provide multiple lemmas, as we do with isomorphisms [8].

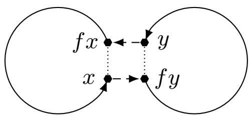
8 Combining Plane Embeddings

It remains to give a formal account of the combinations of plane embeddings performed in the proof of Theorem 8. That is, we need to be able to glue two plane embeddings together, either along a shared vertex or along a shared edge, the latter being used in the case outlined in the informal proof sketch of Theorem 8 given in Section 3.

It is straightforward to show that disjoint unions of planar hypermaps are again planar. As a consequence, both gluing operations can be reduced to obtaining a plane embedding for G/xy from a plane embedding for G. Here, gluing along an edge amounts to merging the respective ends of the two edges one by one. On hypermaps, merging two nodes only changes the node and face permutations, leaving the type of darts and the edge permutation unchanged. Moreover, both the change to the node permutation and the change for the face permutation can be expressed in terms of a singe successor-swapping operation.

Let $f : T \to T$ be an injective function over a finite type T and let $x \neq y$.

$$\mathsf{switch}[x, y, f](z) := \begin{cases} fy & \text{if } z = x \\ fx & \text{if } z = y \\ fz & \text{otherwise} \end{cases}$$



The behavior of $\mathsf{switch}[x, y, f]$ is to either link two f-cycles (if x and y are on different f-cycles, as in the drawing above) or to separate an f-cycle into two cycles (if x and y are on the same f-cycle). Further, we have that

$$\mathsf{merge}\,\langle D, e, n, f\rangle\, d_1\, d_2 := \langle D, e, \mathsf{switch}[d_1, d_2, n], \mathsf{switch}[f^{-1}d_2, f^{-1}d_1, f]\rangle$$

is a hypermap. If d_1 and d_2 are darts from different node cycles, $\mathsf{merge}\, D\, d_1\, d_2$ merges said node cycles, adapting the face cycles accordingly. In particular, $\mathsf{merge}\, D\, d_1\, d_2$ preserves the genus of D if either d_1 and d_2 lie on a common face cycle or if d_1 and d_2 are from separate components of D. In the first case, N is decreased by one while F increases by one; in the second case, both N and F are decreased by one, but so is C.

If $g : D \to G$ is an embedding of some graph G, then for all $x, y : G$ that are not adjacent, and for all d_x and d_y such that $g\,d_x = x$ and $g\,d_y = y$, $\mathsf{merge}\,D\,d_1\,d_2$ can be used to embed G/xy. If x and y lie common face of g, then x and y are the images of two darts d_x and d_y that lie on a common face cycle in D, and $\mathsf{merge}\,D\,d_x\,d_y$ yields and embedding of G/xy. If x and y are not connected in G, any choice of preimages of x and y will yield a plane embedding of G/xy. Hence, for gluing two embeddings together on a single vertex, we can make an arbitrary choice. For gluing along two edges $x{-}x'$ and $y{-}y'$ we know that there must be two faces $[x, x'] +\!\!+\, s_1$ and $[y', y] +\!\!+\, s_1]$ Choosing d_x and d_y to be the preimages of x and y on the respective face cycles ensures that $\mathsf{merge}\,D\,d_x\,d_y$ has an f-cycle containing preimages for x' and y', allowing us to obtain a plane embedding for $(G/xy)/x'y'$, which corresponds to gluing together two components of G along the edges $x{-}x'$ and $y{-}y'$. Note that, due to Definition 20 allowing parallel edges, we do not need to remove darts when gluing along an edge. Putting everything together, we obtain the lemma used in the proof of Theorem 8:

▶ **Lemma 30.** *Let G be a simple graph, and let (V_1, V_2) be a separation, such that $V_1 \cap V_2 = \{x, y\}$ and $x{-}y$. If there are plane embeddings for $G[V_1]$ and $G[V_2]$, then there is also a plane embedding for G.*

9 Main Results

Putting everything together, we obtain the following theorem, which corresponds exactly to the theorem formalized in Coq.

▶ **Theorem 31.** *Let G be a K_5-free and $\mathsf{K}_{3,3}$-free simple graph without isolated vertices. Then there exists a (combinatorial) plane embedding for G.*

```
Theorem Wagner (G : sgraph) : no_isolated G ->
  ~ minor G 'K_3,3 /\ ~ minor G 'K_5 -> inhabited (plane_embedding G).
```

Note that, compared with Theorem 8, we have the additional technical side condition that G may not have isolated vertices. As mentioned in Remark 21, this is necessary, because hypermaps cannot represent isolated vertices. However, isolated vertices can often be treated separately without too much effort as exemplified below.

▶ **Definition 32.** *A (loopless) hypermap $\langle D, e, n, f\rangle$ is k-colorable if there is a coloring of its darts using at most k colors, such that for all $d : D$, the color of $e(d)$ is different from the color of d and the color of $n(d)$ is the same as the color of d. A simple graph is k-colorable, if there is a coloring of its vertices using at most k colors such that adjacent vertices have different colors.*

▶ **Theorem 33** ([9, 10])**.** *Every planar loopless hypermap is 4-colorable*

▶ **Theorem 34.** *Let G be a K_5-free and $\mathsf{K}_{3,3}$-free simple graph. Then G is four-colorable.*

Proof. Let V be the set of vertices with nonempty neighborhood. We obtain a 4-coloring of $G[V]$ using Theorems 31 and 33. This coloring extends to a 4-coloring of G by picking an arbitrary color for the isolated vertices. ◀

10 Conclusion and Future Work

We have introduced a combinatorial approximation of embeddings of graphs in the plane and proved that, with respect to this notion of plane embedding, every K_5-free and $\mathsf{K}_{3,3}$-free graph without isolated vertices is planar. This corresponds to proving the mathematically interesting direction of Wagner's theorem and allows proving a structural variant of the four-color theorem that, unlike the formulations in [10], mentions neither hypermaps nor regions of the real plane. Instead, we bridge the gap between simple graphs and hypermaps, making the four-color theorem available to the setting of a more standard representation of graphs.

The main focus of this work was to bridge the aforementioned gap rather than provide a faithful proof of the usual formulation of Wagner's theorem. Nevertheless, we argue that Theorem 8 and its proof are actually quite faithful to the usual formulation. First, it seems plausible that the notion of plane embedding can be adapted to allow for isolated vertices by relaxing the surjectivity requirement, allowing isolated vertices to not have a dart mapped to them. However, this would come at the cost of some (minor) complications, as one could no longer define a partial inverse for every embedding. More importantly, key arguments of the proof (e.g., Theorems 11 and 25 and Lemmas 12 and 28) closely correspond to what one would find in a detailed paper proof [1, 5]. The main difference is that arguments about modifications of plane embeddings, many of which are normally handled informally, either vanish completely or are replaced by rigorous machine-checked proofs on hypermaps. It should be said that finding these proofs took considerable effort. Hypermaps are complex objects and, apart from the work of Gonthier [9, 10], there is little material in the literature on how to reason efficiently using hypermaps on paper and in an interactive theorem prover. Combined with the fact that some of the proofs are quite technical (e.g. Lemma 26), the learning curve is fairly steep. I hope that this work will contribute to making hypermaps more accessible.

Standing at around 7000 lines (counting additions to the preexisting graph-theory library), the Coq development accompanying this paper is substantial, increasing the total size of the library by more than a third. Around half of these additions deal with operations on hypermaps and plane embeddings. Both the total size and the fraction dealing with hypermaps are bigger than originally envisioned, and I hope that both can still be improved.

As mentioned in Section 1, we have only proved one direction of Wagner's theorem. It remains to show that graphs that can be represented using planar hypermaps have neither K_5 nor $\mathsf{K}_{3,3}$ as a minor. It is relatively straightforward to show that a graph contains K_5 or $\mathsf{K}_{3,3}$ as minor iff it contains an edge subdivision of K_5 or $\mathsf{K}_{3,3}$ as a subgraph [5, Proposition 4.4.2]. This leads to a variant of Wagner's theorem known as Kuratowski's theorem. We have already proved that K_5 and $\mathsf{K}_{3,3}$ do not have plane embeddings (cf. Proposition 23), and that planar graphs are closed under taking subgraphs (Lemma 27). Hence, Kuratowski's theorem and the converse direction of Wagner's theorem could be obtained by proving that planar graphs are closed under removing edge-subdivisions. This direction has already been formalized in Isabelle/HOL [13], and the main obstacle is that reasoning about contained subdivisions (i.e., topological minors) is more cumbersome than reasoning about (normal) minors.

Besides the converse direction of Wagner's theorem, there are many other related theorems that would make for interesting future work. It is well known that in the case of 3-connected planar graphs, all plane embeddings have the same structure [1, Theorem 10.28]. In our setting, this means that the embedding is unique up to isomorphisms of hypermaps. Further, a common strengthening of Proposition 7 is to show that one can obtain a plane embedding in which all inner faces are convex. This strengthening is not expressible using the hypermap model of plane embeddings, and this raises the question whether one could introduce an abstract notion of plane embedding and instantiate it with hypermaps as well as models based on axiomatic geometry or embeddings in the real plane. On the other hand, given that the (combinatorial) plane embedding of a 3-connected planar graph is unique, it should also be possible to directly construct a convex embedding in the real plane for this hypermap, separating the existence and convexity parts of the proof.

References

1 Adrian Bondy and U.S.R. Murty. *Graph Theory*, volume 244 of *Graduate Texts in Mathematics*. Springer-Verlag London, 2008.

2 Ching-Tsun Chou. A formal theory of undirected graphs in higher-order logic. In *TPHOL*, volume 859 of *LNCS*, pages 144–157. Springer, 1994. doi:10.1007/3-540-58450-1_40.

3 Ching-Tsun Chou. Mechanical verification of distributed algorithms in higher-order logic. *The Computer Journal*, 38(2):152–161, January 1995. doi:10.1093/comjnl/38.2.152.

4 Robert Cori. *Un code pour les graphes planaires et ses applications*. PhD thesis, Univ. Paris VII, 1973.

5 Reinhard Diestel. *Graph Theory (2nd edition)*. Graduate Texts in Mathematics. Springer, 2000.

6 Christian Doczkal, Guillaume Combette, and Damien Pous. A formal proof of the minor-exclusion property for treewidth-two graphs. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving (ITP 2018)*, volume 10895 of *LNCS*, pages 178–195. Springer, 2018. doi:10.1007/978-3-319-94821-8_11.

7 Christian Doczkal and Damien Pous. Completeness of an axiomatization of graph isomorphism via graph rewriting in Coq. In *Proc. of 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '20), January 20–21, 2020, New Orleans, LA, USA*, 2020. doi:10.1145/3372885.3373831.

8 Christian Doczkal and Damien Pous. Graph theory in Coq: Minors, treewidth, and isomorphisms. *J. Autom. Reason.*, 64(5):795–825, 2020. doi:10.1007/s10817-020-09543-2.

9 Georges Gonthier. A computer-checked proof of the four colour theorem, 2005. URL: https://www.cl.cam.ac.uk/~lp15/Pages/4colproof.pdf.

10 Georges Gonthier. Formal proof — the four-color theorem. *Notices Amer. Math. Soc.*, 55(11):1382–1393, 2008. URL: http://www.ams.org/notices/200811/tx081101382p.pdf.

11 Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Le Truong Hoang, Cezary Kaliszyk, Victor Magron, Sean Mclaughlin, and Tat Thang Nguyen. A formal proof of the Kepler conjecture. *Forum of Mathematics, Pi*, 5, 2017. doi:10.1017/fmp.2017.1.

12 Yatsuka Nakamura and Piotr Rudnicki. Euler circuits and paths. *Formalized Mathematics*, 6(3):417–425, 1997.

13 Lars Noschinski. *Formalizing Graph Theory and Planarity Certificates*. PhD thesis, Technische Universität München, 2016.

14 Daniel E. Severín. Formalization of the domination chain with weighted parameters (short paper). In John Harrison, John O'Leary, and Andrew Tolmach, editors, *Interactive Theorem Provin (ITP 2019)*, volume 141 of *LIPIcs*, pages 36:1–36:7. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ITP.2019.36.

15 Abhishek Kr Singh and Raja Natarajan. A constructive formalization of the weak perfect graph theorem. In *Proc. of 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '20), January 20–21, 2020, New Orleans, LA, USA*, 2020. doi:10.1145/3372885.3373819.

16 Saul Stahl. A combinatorial analog of the Jordan curve theorem. *J. Comb. Theory, Ser. B*, 35(1):28–38, 1983. doi:10.1016/0095-8956(83)90078-3.

17 William T. Tutte. Duality and trinity. In *Colloquium Mathematical Society Janos Bolyai 10*, pages 1459–1472, 1975.

18 David W. Walkup. How many ways can a permutation be factored into two n-cycles? *Discret. Math.*, 28(3):315–319, 1979. doi:10.1016/0012-365X(79)90138-9.

Formalized Haar Measure

Floris van Doorn
University of Pittsburgh, PA, USA

Abstract

We describe the formalization of the existence and uniqueness of the Haar measure in the Lean theorem prover. The Haar measure is an invariant regular measure on locally compact groups, and it has not been formalized in a proof assistant before. We will also discuss the measure theory library in Lean's mathematical library mathlib, and discuss the construction of product measures and the proof of Fubini's theorem for the Bochner integral.

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

Keywords and phrases Haar measure, measure theory, Bochner integral, Lean, interactive theorem proving, formalized mathematics

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.18

Related Version *Full Version*: `https://arxiv.org/abs/2102.07636`

Supplementary Material *Software (Source Code)*: `https://github.com/leanprover-community/mathlib/tree/master/src/measure_theory`
archived at `swh:1:dir:4b232212fa94778af7c966340636fd0e061d9598`

Funding This paper is supported by the Sloan Foundation (grant G-2018-10067).

Acknowledgements I want to thank Jeremy Avigad, Tom Hales, the Leanprover community and the anonymous reviewers for proofreading this paper.

1 Introduction

Measure theory is an important part of mathematics, providing a rigorous basis for integration and probability theory. The main object of study in this part is the concept of a measure, which assigns a size to the measurable subsets of the space in question. The most widely used measure is the Lebesgue measure λ on $\mathbb{R}$ (or $\mathbb{R}^n$) with the property that $\lambda([a, b]) = b - a$. The Lebesgue measure is *translation invariant*, meaning that translating a measurable set does not change its measure.

An important generalization of the Lebesgue measure is the Haar measure, which provides a measure on locally compact groups. This measure is also invariant under translations, i.e. applying the group operation. For non-abelian groups we have to distinguish between left and right Haar measures, which are invariant when applying the group operation on the left and right, respectively. A Haar measure is essentially unique, which means that any invariant measure is a constant multiple of a chosen Haar measure. The Haar measure is vital in various areas of mathematics, including harmonic analysis, representation theory, probability theory and ergodic theory.

In this paper we describe the formal definition of the Haar measure in the Lean theorem prover [5], and prove its uniqueness. We build on top of the Lean mathematical library mathlib [4]. We heavily use the library for measure theory in mathlib, improving it in the process. As part of this formalization we expanded the library with product measures and Fubini's theorem, along with many other contributions to existing libraries. The results in this paper have been fully integrated into mathlib.

Many proof assistants have a library of measure theory and integration, including Isabelle [17, 12], HOL/HOL4 [13, 2, 15], HOL Light [11], Mizar [7], PVS [14] and Coq [19]. Most of these libraries focus on the Lebesgue measure and the Lebesgue integral, and on applications to probability theory or analysis in Euclidean spaces.

The Isabelle/HOL library has an especially good development of measure theory, including product measures and the Bochner integral [1]. The Lean library also has these concepts, and the development of them was heavily inspired by the Isabelle/HOL library (and partially written by Johannes Hölzl, who also wrote parts of the Isabelle/HOL library). However, to my knowledge, the Haar measure has not been formalized in any proof assistant other than Lean.

In this paper we will discuss the measure theory library in Lean (in Section 3), which was already in place before this formalization started, and was developed by Johannes Hölzl, Mario Carneiro, Zhouhang Zhou and Yury Kudryashov, among others. The other sections describe contributions that are new as of this formalization: product measures (in Section 4), the definition of the Haar measure (in Section 5), and the uniqueness of the Haar measure (in Section 6).

We will link to specific results in mathlib using the icon ↗. To ensure that these links will work while mathlib is updated, we link to the version of mathlib as of writing this paper. Readers of this paper are also encouraged to use the search function of the mathlib documentation pages ↗ but we do not link to specific pages of the documentation, as these links are prone to break as the library evolves.

We used different sources for this formalization. We started out using the notes by Jonathan Gleason [8] that gave a construction of the Haar measure in detail. However, one of the main claims in that paper was incorrect, as described in Section 5. We then used the books by Halmos and Cohn [10, 3] to fill in various details. For product measures we followed Cohn [3]. For Fubini's theorem for the Bochner integral we followed the formalization in Isabelle. We couldn't find a good source for this proof in the literature, so we will provide detailed proofs of this result in Section 4. A different proof can be found in [16]. For the uniqueness of the Haar measure we followed Halmos [10]. While we define the Haar measure for any locally compact group, we prove the uniqueness only for second-countable locally compact groups. The reason is that Halmos gives a uniqueness proof using measurable groups, and only second-countable topological groups form measurable groups. In the proof of a key theorem of Halmos there was a gap, which we fixed in the formalization by changing the statement of that theorem (see Section 6).

2 Preliminaries

2.1 Lean and mathlib

Lean [5] is a dependently typed proof assistant with a strong metaprogramming framework [6]. Its logic contains dependent function types denoted `Π i : ι, X i`, universes `Type*` (the star denotes an arbitrary universe level) and inductive types. It has a large community-driven mathematical library, called mathlib. We use results from various areas in this library, including topology, analysis and algebra. In particular the existing library for topological groups was very helpful for this formalization. The fact that Lean provides a very expressive type theory was convenient in the definition of the Bochner integral, as we will discuss in Section 3.

mathlib uses *type-classes* [20, 18] to organize the mathematical structures associated to various types, using the partially bundled approach. For example, when `G` is a type, the statement `[group G]` states that `G` has the structure of a group, and `[topological_space`

G] states that G has the structure of a topological space. The square brackets mean that the argument will be implicitly inserted in definitions and theorems using type-class inference. Some type-classes are *mixins*, and take other type-classes as arguments. For example, the class topological_group takes as (implicit) argument the structure of a group and a topological_space. This means that we need to write [group G] [topological_space G] [topological_group G] to state that G is a topological group. Similarly, predicates on topological spaces – such as t2_space, locally_compact_space or compact_space – take the topological space structure as argument. This design decision makes it somewhat verbose to get the right structure on a type, but has the advantage that these components can be freely mixed as needed.

Lean distinguishes groups written multiplicatively (group) from groups written additively (add_group). This is convenient since it automatically establishes multiplicative notation (1 * x⁻¹) or additive notation (0 + -x or 0 - x) when declaring a group structure. Also, most theorems exist for both versions; the property x * y = y * x is called mul_comm and the property x + y = y + x is add_comm. To avoid code duplication, there is an attribute @[to_additive] that, when applied to a definition or theorem for multiplicative groups, automatically generates the corresponding declaration for additive groups by replacing all multiplicative notions with the corresponding additive ones.

Some code snippets in this paper have been edited slightly for the sake of readability.

2.2 Mathematics in mathlib

The notation for the image of a set A : set X under a function f : X → Y is f '' A = {y : Y | ∃ x ∈ A, f x = y} and the preimage of B : set Y is f ⁻¹' B = {x : X | f x ∈ B}. The notation λ x, f x is used for the function $x \mapsto f(x)$. We write (x : X) for the element x with its type X given explicitly. This can also be used to *coerce* x to the type X.

mathlib has a large topology library, where many concepts are defined in terms of filters. The following code snippet contains some notions in topology used in this paper.

```
class t2_space (X : Type*) [topological_space X] : Prop :=
(t2 : ∀ x y, x ≠ y → ∃ U V : set X, is_open U ∧ is_open V ∧
  x ∈ U ∧ y ∈ V ∧ U ∩ V = ∅)

def is_compact {X : Type*} [topological_space X] (K : set X) :=
∀ {f : filter X} [ne_bot f], f ≤ 𝓟 K → ∃ x ∈ K, cluster_pt x f

class locally_compact_space (X : Type*) [topological_space X] : Prop :=
(local_compact_nhds : ∀ (x : X) (U ∈ 𝓝 x), ∃ K ∈ 𝓝 x, K ⊆ U ∧
  is_compact K)

class second_countable_topology (X : Type*) [t : topological_space X] :
    Prop :=
(is_open_generated_countable : ∃ C : set (set X), countable C ∧
  t = generate_from C)

class complete_space (X : Type*) [uniform_space X] : Prop :=
(complete : ∀ {f : filter X}, cauchy f → ∃ x, f ≤ 𝓝 x)

class topological_group (G : Type*) [topological_space G] [group G] :=
(continuous_mul : continuous (λ p : G × G, p.1 * p.2))
(continuous_inv : continuous (inv : G → G))
```

Some of these definitions might look unfamiliar, formulated in terms of filters, using in particular the principal filter $\mathcal{P}$ and the neighborhood filter $\mathcal{N}$. For example, the second definition states that a set `K` is *compact* if any filter `f` $\neq \perp$ (i.e. `f` doesn't contain every subset of `X`) containing `K` has a cluster point `x` in `K`, which means that $\mathcal{N}$ `x` $\sqcap$ `f` $\neq \perp$. It is equivalent to the usual notion of compactness that every open cover of `K` has a finite subcover. A complete space is a space where every Cauchy sequence converges, which is formulated in the general setting of *uniform spaces*, which simultaneously generalize metric spaces and topological groups. We define a *locally compact group* to be a topological group in which the topology is both locally compact and Hausdorff (T_2)

Lean's notion of infinite sums is defined for absolutely convergent series as the limit of all finite partial sums:

```
def has_sum {I M : Type*} [add_comm_monoid M] [topological_space M]
  (f : I → M) (x : M) : Prop :=
tendsto (λ s : finset I, ∑ i in s, f i) at_top (𝒩 x)
```

If `M` is Hausdorff, then the sum is unique if it exists, and we denote it by `∑' i, f i`. If the series does not have a sum, it is defined to be `0`.

The type `ennreal` is the type of nonnegative real numbers extended by a new element ∞, which we will also denote $[0,\infty]$ in this paper.

A *normed space* X is a vector space over a normed field F equipped with a norm $\|\cdot\| : X \to \mathbb{R}$ satisfying

- (X, d) is a metric space, with the distance function given by $d(x, y) := \|x - y\|$.
- for $r \in F$ and $x \in X$ we have $\|r \cdot x\| = \|r\| \cdot \|x\|$.

In this paper, we will only consider normed spaces over $\mathbb{R}$, which is a normed field with its norm given by the absolute value. A *Banach space* is a complete normed space. In Lean we do not explicitly define Banach spaces, but instead talk about complete normed spaces.

3 Measure Theory in mathlib

In this section we describe the background measure theory library used in this formalization.

A basic notion in measure theory is a σ-*algebra* on X, which is a nonempty collection of subsets of X that is closed under complements and countable unions. In mathlib this is formulated as a type-class, with name `measurable_space`:

```
class measurable_space (X : Type*) :=
(is_measurable : set X → Prop)
(is_measurable_empty : is_measurable ∅)
(is_measurable_compl : ∀ A, is_measurable A → is_measurable Aᶜ)
(is_measurable_Union : ∀ A : ℕ → set X, (∀ i, is_measurable (A i)) →
  is_measurable (⋃ i, A i))
```

We say that a function is *measurable* if the preimage of every measurable set is measurable:

```
def measurable [measurable_space X] [measurable_space Y] (f : X → Y) :=
∀ {B : set Y}, is_measurable B → is_measurable (f ⁻¹' B)
```

Note that we use `is_measurable` for sets and `measurable` for functions.

In mathlib, measures are defined as special case of outer measures. An outer measure on X is a monotone function $m : \mathcal{P}(X) \to [0,\infty]$ such that $m(\emptyset) = 0$ and m is countably subadditive:

```
structure outer_measure (X : Type*) :=
(measure_of : set X → ennreal)
(empty : measure_of ∅ = 0)
(mono : ∀ {A₁ A₂}, A₁ ⊆ A₂ → measure_of A₁ ≤ measure_of A₂)
(Union_nat : ∀ (A : ℕ → set X),
  measure_of (⋃ i, A i) ≤ (∑' i, measure_of (A i)))
```

A measure on a measurable space is an outer measure with two extra properties: it is countably additive on measurable sets, and given the value on measurable sets, the outer measure is the maximal one that is compatible with these values (which is called `trim` in the snippet below).

```
structure measure (X : Type*) [measurable_space X]
  extends outer_measure X :=
(m_Union : ∀ {A : ℕ → set X},
  (∀ i, is_measurable (A i)) → pairwise (disjoint on A) →
  measure_of (⋃ i, A i) = ∑' i, measure_of (A i))
(trimmed : to_outer_measure.trim = to_outer_measure)
```

This definition has two very convenient properties:

1. We can apply measures to any set, without having to provide a proof that the set is measurable.
2. Two measures are equal when they are equal on measurable sets.

Given a measure μ on X and a measurable map $f : X \to Y$, we can define the *pushforward* $f_*\mu$, which is a measure on Y, defined on measurable sets A by $(f_*\mu)(A) = \mu(f^{-1}(A))$.

We proceed to define lower Lebesgue integration $\int^-$ for functions $f : X \to [0, \infty]$, where X is a measurable space. We annotate this integral with a minus sign to distinguish it from the Bochner integral. This is done first for *simple functions*, i.e. functions with finite range with the property that the preimage of all singletons are measurable. The integral of a simple function g is simply

$$\int^- g \,\mathrm{d}\mu = \int^- g(x) \,\mathrm{d}\mu(x) = \sum_{y \in g(X)} \mu(g^{-1}\{y\}) \cdot y \in [0, \infty].$$

If $f : X \to [0, \infty]$ is any function, we can define the *(lower) Lebesgue integral* of f to be the supremum of $\int^- g \,\mathrm{d}\mu(x)$ for all simple functions g that satisfy $g \leq f$ (pointwise). In mathlib we denote the Lebesgue integral of `f` by `∫⁻ x, f x ∂μ`. We prove the standard properties of the Lebesgue integral for nonnegative functions, such as the monotone convergence theorem:

```
theorem lintegral_supr {f : ℕ → X → ennreal}
  (hf : ∀ n, measurable (f n)) (h_mono : monotone f) :
  (∫⁻ x, ⨆ n, f n x ∂μ) = (⨆ n, ∫⁻ x, f n x ∂μ)
```

After defining the Lebesgue integral for nonnegative functions, most books and formal libraries continue to define the Lebesgue integral for (signed) functions $f : X \to \mathbb{R}$. This is defined as the difference of the integrals of the positive and negative parts of f. However, this is not very general: separate definitions are needed for integrals for functions with codomain $\mathbb{C}$ or $\mathbb{R}^n$. Instead, we opt for the more general definition of the *Bochner integral*, which is defined for functions that map into any second-countable real Banach space.

If X is a topological space, we equip it with the *Borel* σ-algebra, which is the smallest σ-algebra that contains all open sets (or equivalently, all closed sets). The fact that X is equipped with the Borel σ-algebra is a mixin, and is specified by providing the arguments `[topological_space X] [measurable_space X] [borel_space X]`.

Let X be a measurable space and E be a second-countable real Banach space. We define two quotients on functions from X to E.

Let $X \to_\mu E$ be the measurable maps $X \to E$ modulo μ-a.e. (almost everywhere) equivalence, i.e. modulo the relation

$$f =_\mu g \iff \mu(\{x \mid f(x) \neq g(x)\}) = 0.$$

Note that we are using the fact that Lean is based on dependent type theory here, since $X \to_\mu E$ is a type that depends on μ. Here our formalization differs from the one in Isabelle, since in Isabelle this type cannot be formed. Instead, a similar argument is given purely on functions, without passing to the quotient.

If $f : X \to E$ we say that f is *integrable* if

- f is μ-a.e. measurable, i.e. there is a measurable function $g : X \to E$ with $f =_\mu g$;[1] and
- $\int^- \|f(x)\| \,\mathrm{d}\mu(x) < \infty$.

Being integrable is preserved by μ-a.e. equivalence, and we define the L^1-space $L^1(X, \mu; E)$ as those equivalence classes of functions in $X \to_\mu E$ that are integrable. $L^1(X, \mu; E)$ is a normed space, with the norm given by

$$\|f\| := \int^- \|f(x)\| \,\mathrm{d}\mu(x).$$

We define the Bochner integral first for simple functions, similar to the definition for the Lebesgue integral. If $g : X \to E$ is a simple function then its Bochner integral is

$$\int g \,\mathrm{d}\mu = \int g(x) \,\mathrm{d}\mu(x) = \sum_{\substack{y \in g(X) \text{ s.t.} \\ \mu(g^{-1}\{y\}) < \infty}} \mu(g^{-1}\{y\}) \cdot y \in E.$$

The symbol $\cdot$ denotes the scalar multiplication in the Banach space E. Since this definition respects μ-a.e. equivalence, we can also define the Bochner integral on the simple L^1 functions. On the simple L^1 functions this definition is continuous and the simple L^1 functions are dense in all L^1 functions, and E is complete, so we can extend the Bochner integral to all L^1 functions.

Finally, for an arbitrary function $f : X \to E$ we define its Bochner integral to be 0 if f is not integrable and the integral of $[f] \in L^1(X, \mu; E)$ otherwise, where $[f]$ is the equivalence class of f. The two notions of integrals agree on integrable functions $f : X \to \mathbb{R}_{\geq 0}$:

$$\int^- f \,\mathrm{d}\mu = \int f \,\mathrm{d}\mu.$$

However, when f is has infinite Lebesgue integral, then the equality does not hold: the LHS is ∞, while the RHS is 0. We then prove useful properties about the Bochner integral, such as the dominated convergence theorem:

[1] Until recently, the definition of "integrable" in mathlib required f to be measurable. The definition was modified to increase generality. The motivating reason was that we would regularly take integrals of a function f with a measure $\mu|_A$ defined by $\mu|_A(B) = \mu(A \cap B)$. A function that is (for example) continuous on A is also $\mu|_A$-a.e. measurable, and we usually do not need any information about the behavior of f outside A. In the previous definition of "integrable" we required f to be measurable everywhere, which would be an unnecessary additional assumption.

```
theorem tendsto_integral_of_dominated_convergence
  {F : ℕ → X → E} {f : X → E} (bound : X → ℝ)
  (F_measurable : ∀ n, ae_measurable (F n) μ)
  (f_measurable : ae_measurable f μ)
  (bound_integrable : integrable bound μ)
  (h_bound : ∀ n, ∀ᵐ x ∂μ, ‖F n x‖ ≤ bound x)
  (h_lim : ∀ᵐ x ∂μ, tendsto (λ n, F n x) at_top (𝓝 (f x))) :
  tendsto (λ n, ∫ x, F n x ∂μ) at_top (𝓝 (∫ x, f x ∂μ))
```

In this statement, `∀ᵐ x ∂μ, P x` means that `P x` holds for μ-almost all `x`, in other words that `μ { x | ¬ P x } = 0`, and `tendsto (λ n, g n) at_top (𝓝 x)` means that `g n` tends to `x` as `n` tends to ∞.

Other results about integration include part 1 and 2 of the fundamental theorem of calculus, but we will not use these results in this paper.

4 Products of Measures

To prove that the Haar measure is essentially unique, we need to work with product measures and iterated integrals. In this section we will define the product measure and prove Tonelli's and Fubini's theorem. (Sometimes both theorems are called Fubini's theorem, but in this paper we will distinguish them.) Tonelli's theorem characterizes the Lebesgue integral for nonnegative functions in the product space, and Fubini's theorem does the same (under stronger conditions) for the Bochner integral. By symmetry, both theorems also give sufficient conditions for swapping the order of integration when working with iterated integrals.

The contents of this section have been formalized before in Isabelle. Product measures, Tonelli's theorem and Fubini's theorem for the Lebesgue integral are discussed in the paper [12]. Fubini's theorem for the Bochner integral is significantly harder to prove, but has also been formalized in Isabelle after the appearance of that paper. We read through the proof in the Isabelle formalization for key ideas in various intermediate lemmas.

In this section we will restrict our attention to σ-finite measures, since product measures are much nicer for σ-finite measures, and most of the results do not hold without this condition. We say that a measure μ on a space X is σ-finite if X can be written as a union of countably many subsets $(A_i)_i$ with $\mu(A_i) < \infty$ for all i.

First we need to know that limits of measurable functions are measurable.

▶ **Lemma 1.** *Let X be a measurable space.*

1. *Given a sequence of measurable functions $f_n : X \to [0, \infty]$ for $n \in \mathbb{N}$, the pointwise suprema and infima $f(x) := \sup\{f_n(x)\}_n$ and $g(x) := \inf\{f_n(x)\}_n$ are measurable.*
2. *If a sequence of measurable functions $f_n : X \to [0, \infty]$ for $n \in \mathbb{N}$ converges pointwise to a function f, then f is measurable.*
3. *Let Y be a metric space equipped with the Borel σ-algebra. If a sequence of measurable functions $f_n : X \to Y$ for $n \in \mathbb{N}$ converges pointwise to a function f, then f is measurable.*

Proof. For part 1, note that the collection of intervals $[0, x)$ generate the σ-algebra on $[0, \infty]$, as do the intervals $(x, \infty]$. The conclusion follows from the fact that measurable sets are closed under countable unions and observing that $f^{-1}([0, x)) = \bigcup_n f_n^{-1}([0, x))$ and $g^{-1}((x, \infty]) = \bigcup_n f_n^{-1}((x, \infty])$.

Part 2 follows by observing that $f(x) = \liminf_n f_n(x) = \sup_n \inf_{m \geq n} f_m(x)$, and applying both claims in part 1.

For part 3, let C be a closed set in Y. The function $d(-,C) : Y \to [0,\infty]$ that assigns to each point the (minimal) distance to C is continuous. Therefore, we know that $h(x) := d(f(x), C)$ is the limit of the measurable functions $d(f_n(x), C)$ as $n \to \infty$. By part 2 we know that h is measurable. Therefore $f^{-1}(C) = h^{-1}(\{0\})$ is measurable (the equality holds because C is closed). ◀

If X and Y are measurable spaces, the σ-algebra on $X \times Y$ is the smallest σ-algebra that makes the projection $X \times Y \to X$ and $X \times Y \to Y$ measurable. Alternatively, it can be characterized as the σ-algebra generated by sets of the form $A \times B$ for measurable $A \subseteq X$ and $B \subseteq Y$.

For the rest of this section, μ is a σ-finite measure on X and ν is a σ-finite measure on Y. For a set $A \subseteq X \times Y$ we write $A_x = \{y \in Y \mid (x,y) \in A\}$ for a slice of Y. In Lean, this is written as `prod.mk x ⁻¹' A`. We want to define the product measure evaluated at A as an integral over the volume of the slices of A. To make sure that this makes sense, we need to prove that this is a measurable function.

```
lemma measurable_measure_prod_mk_left [sigma_finite ν] {A : set (X × Y)}
  (hA : is_measurable A) : measurable (λ x, ν (prod.mk x ⁻¹' A))
```

This lemma crucially depends on the fact that ν is σ-finite, but we omit the proof here. We can now define the product measure $\mu \times \nu$ on $X \times Y$ as

$$(\mu \times \nu)(A) = \int^{-} \nu(A_x)\, \mathrm{d}\mu(x). \tag{1}$$

It is not hard to see that $(\mu \times \nu)(A \times B) = \mu(A)\nu(B)$ for measurable $A \subseteq X$ and $B \subseteq Y$ and that $\mu \times \nu$ is itself σ-finite. When we pushforward the measure $\mu \times \nu$ across the map $X \times Y \to Y \times X$ that swaps the coordinates, we get the measure $\nu \times \mu$ (this equality is easily checked because both measures are equal on rectangles). This immediately gives the symmetric version of (1), using the notation $A^y = \{x \in X \mid (x,y) \in A\}$:

$$(\mu \times \nu)(A) = \int^{-} \mu(A^y)\, \mathrm{d}\nu(y).$$

For a function $f : X \times Y \to Z$ define $f_x : Y \to Z$ by $f_x(y) := f(x,y)$. If $Z = [0,\infty]$ let $I_f^- : X \to [0,\infty]$ be defined by $I_f^-(x) := \int_Y^- f_x\, \mathrm{d}\nu = \int_Y^- f(x,y)\, \mathrm{d}\nu(y)$.

We can now formulate Tonelli's theorem.

▶ **Theorem 2** (Tonelli's theorem). *Let $f : X \times Y \to [0,\infty]$ be a measurable function. Then*

$$\int_{X \times Y}^{-} f\, \mathrm{d}(\mu \times \nu) = \int_X^- \int_Y^- f(x,y)\, \mathrm{d}\nu(y) \mathrm{d}\mu(x) = \int_Y^- \int_X^- f(x,y)\, \mathrm{d}\mu(x) \mathrm{d}\nu(y),$$

and all the functions in the integrals above are measurable.

Proof. We check the first equality and the measurability of I_f^-. The other claims follow by symmetry. For both these statements we use *induction on measurable functions into* $[0,\infty]$, which states that to prove a statement $P(f)$ for all measurable functions f into $[0,\infty]$, it is sufficient to show

1. if χ_A is the characteristic function of a measurable set and $c \in [0,\infty]$, we have $P(c\chi_A)$;
2. if f_1 and f_2 are measurable functions with $P(f_1)$ and $P(f_2)$, then $P(f_1 + f_2)$;
3. if $(f_i)_i$ is a monotone sequence of measurable functions such that $P(f_i)$ for all i, we have $P(\sup_i f_i)$.

We first prove that I^{-}_{f} is measurable by induction on f.

In the induction base, if $f = c\chi_A$, then $\int^{-}_{Y} f(x,y)\,\mathrm{d}\nu(y) = c\nu(A_x)$, so the measurability follows from `measurable_measure_prod_mk_left`. The two induction steps follow from the additivity of the integral and the monotone convergence theorem.

We now prove the first equality in Theorem 2, also by induction on f. In the induction base, if $f = c\chi_A$, then it is not hard to see that both sides reduce to $c(\mu \times \nu)(A)$. The first induction step follows by applying additivity thrice, and the second induction step follows by applying the monotone convergence theorem thrice. ◀

Fubini's theorem (sometimes called Fubini–Tonelli's theorem) states something similar for the Bochner integral. However, it is a bit harder to state and much harder to prove. For a function $f : X \times Y \to E$ define $I_f : X \to E$ by $I_f(x) := \int_Y f_x\,\mathrm{d}\nu = \int_Y f(x,y)\,\mathrm{d}\nu(y)$.

▶ **Theorem 3** (Fubini's theorem for the Bochner integral)**.** *Let E be a second-countable Banach space and $f : X \times Y \to E$ be a function.*

1. *If f is $\mu \times \nu$-a.e. measurable then f is integrable iff the following two conditions hold:*
   - *for almost all $x \in X$ the function f_x is ν-integrable;*
   - *the function $I_{\|f\|}$ is μ-integrable.*
2. *If f is integrable, then*

$$\int_{X\times Y} f\,\mathrm{d}(\mu\times\nu) = \int_X\int_Y f(x,y)\,\mathrm{d}\nu(y)\mathrm{d}\mu(x) = \int_Y\int_X f(x,y)\,\mathrm{d}\mu(x)\mathrm{d}\nu(y).$$

Moreover, all the functions in the integrals above are a.e. measurable.

Note that Part 1 also has a symmetric counterpart, which we omit here. Note that the middle term in Part 2 is just $\int_X I_f\,\mathrm{d}\mu(x)$.

Proof.

Measurability. First suppose that f is measurable. In this case, we show that the function I_f is measurable. We can approximate any measurable function f into E by a sequence $(s_n)_n$ of simple functions such that for all z we have $\|s_n(z)\| \le 2\|f(z)\|$ and $s_n(z) \to f(z)$ as $n \to \infty$. Now define $g_n : X \to E$ by

$$g_n(x) = \begin{cases} I_{s_n}(x) & \text{if } f_x \text{ is } \nu\text{-integrable} \\ 0 & \text{otherwise.} \end{cases}$$

Note that g_n is similar to I_{s_n}, except that we set it to 0 whenever f_x is not integrable. This is required to ensure that g_n converges to I_f (see below). We first check that g_n is measurable. Note that the set

$$\{x \mid f_x \text{ is } \nu\text{-integrable}\} = \{x \mid \textstyle\int_Y \|f(x,y)\|\,\mathrm{d}\nu(y) < \infty\}$$

is measurable, using Theorem 2. Therefore, to show that g_n is measurable, it suffices to show that I_{s_n} is measurable, which is true since it is a finite sum of functions of the form $x \mapsto c\nu(A_x)$ (using that s_n is simple).

Secondly, we check that g_n converges to I_f pointwise. Let $x \in X$. If f_x is not integrable, it is trivially true, so assume that f_x is integrable. Then $(s_n)_x$ is integrable for all n, and by the dominated convergence theorem $I_{s_n}(x)$ converges to $I_f(x)$.

We conclude that I_f is measurable by Lemma 1.

Finally, if f is $\mu\times\nu$-a.e. measurable, then it is not hard to show that I_f is μ-a.e. measurable.

Integrability. We now prove Part 1, so suppose that f is $\mu \times \nu$-a.e. measurable. By Tonelli's theorem, we know that f is $\mu \times \nu$-integrable iff $I^{-}_{\|f\|}$ is μ-integrable. Note that $I^{-}_{\|f\|}$ is similar to $I_{\|f\|}$. The difference is that when f_x is not integrable, $I_{\|f\|}(x) = 0$ but $I^{-}_{\|f\|} = \infty$. This means that if $I^{-}_{\|f\|}$ is μ-integrable, then f_x must almost always be ν-integrable (otherwise the integrand is infinite on a set of positive measure) and $I_{\|f\|}$ must be μ-integrable. For the other direction in Part 1, if f_x is almost always integrable, then $I_{\|f\|} = I^{-}_{\|f\|}$ almost everywhere, so if the former is μ-integrable, then so is the latter.

Equality. To prove the first equality in Part 2, we use the following *induction principle for integrable functions.* It states that to prove a statement $P(f)$ for all τ-integrable functions $f : Z \to E$, it is sufficient to show

- if A is measurable with $\tau(A) < \infty$ and $c \in E$, then $P(c\chi_A)$;
- if f_1 and f_2 are τ-integrable functions with $P(f_1)$ and $P(f_2)$, then $P(f_1 + f_2)$;
- If f_1 is integrable with $P(f_1)$ and $f_1 = f_2$ τ-a.e. then $P(f_2)$.
- The set $\{f \in L^1(Z, \tau; E) \mid P(f)\}$ is closed.

We now prove by induction on f that

$$\int_{X \times Y} f \, \mathrm{d}(\mu \times \nu) = \int_X \int_Y f(x, y) \, \mathrm{d}\nu(y)\mathrm{d}\mu(x). \tag{2}$$

- If $f = c\chi_A$ then it is easy to see that both sides of (2) reduce to $c \cdot (\mu \times \nu)(A)$.
- It is clear that if (2) holds for f_1 and f_2, then it also holds for $f_1 + f_2$.
- Suppose that (2) holds for f_1 and that $f_1 = f_2$ almost everywhere. Since integrals are defined up to a.e. equality, we know that

  $$\int_{X \times Y} f_1 \, \mathrm{d}(\mu \times \nu) = \int_{X \times Y} f_2 \, \mathrm{d}(\mu \times \nu).$$

  It therefore suffices to show that

  $$\int_X \int_Y f_1(x, y) \, \mathrm{d}\nu(y)\mathrm{d}\mu(x) = \int_X \int_Y f_2(x, y) \, \mathrm{d}\nu(y)\mathrm{d}\mu(x).$$

  For this it is sufficient to show that $I_{f_1}(x) = I_{f_2}(x)$ for almost all x. Since $f_1(z) = f_2(z)$ for almost all z, it is not hard to see that for almost all x we have that for almost all y the equality $f_1(x, y) = f_2(x, y)$ holds. From this we easily get that $I_{f_1}(x) = I_{f_2}(x)$ for almost all x.
- We need to show that the set of $f \in L^1(Z, \tau; E)$ for which (2) holds is closed. We know that taking the integral of an L^1 function is a continuous operation, which means that both sides of (2) are continuous in f, which means that the set where these functions are equal is closed. ◀

5 Existence of the Haar Measure

In this section, we define the left Haar measure, and discuss some design decisions when formalizing these definitions. We also discuss how to obtain a right Haar measure.

Throughout this section, we assume that `G` is a locally compact group, equipped with the Borel σ-algebra. We will write the group operation multiplicatively. We will define the Haar measure and show that it is a left invariant regular measure on `G`. In the formalization, all intermediate definitions and results are given with weaker conditions on `G`, following the design in `mathlib` that all results should be in the most general form (within reason).

In Lean, the precise conditions on `G` are written as follows.

```
{G : Type*} [topological_space G] [t2_space G] [locally_compact_space G]
  [group G] [measurable_space G] [borel_space G] [topological_group G]
```

The concept of a left invariant measure is defined as follows in mathlib.

```
def is_left_invariant (μ : set G → ennreal) : Prop :=
∀ (g : G) {A : set G} (h : is_measurable A), μ ((λ h, g * h) ⁻¹' A) = μ A
```

Note that the preimage `(λ h, g * h) ⁻¹' A` is equal to the image `(λ h, g⁻¹ * h) '' A`. We use preimages for translations, since preimages are nicer to work with. For example, the fact that `(λ h, g * h) ⁻¹' A` is measurable follows directly from the fact that multiplication (on the left) is a measurable function.

Next we give the definition of a regular measure

```
structure regular (μ : measure X) : Prop :=
(lt_top_of_is_compact : ∀ {K : set X}, is_compact K → μ K < ⊤)
(outer_regular : ∀ {A : set X}, is_measurable A →
  (⨅ (U : set X) (h : is_open U) (h2 : A ⊆ U), μ U) ≤ μ A)
(inner_regular : ∀ {U : set X}, is_open U →
  μ U ≤ ⨆ (K : set X) (h : is_compact K) (h2 : K ⊆ U), μ K)
```

If μ is a regular measure, then the inequalities in the last two fields are always equalities. This means that a measure μ is regular if it is finite on compact sets, its value on a measurable set A is equal to the infimum (⨅) of its value on all open sets containing A, and finally its value on an open set U is the supremum (⨆) of its values on all compacts subsets of U.

We are now ready to start the definition of the Haar measure. Given a compact set K in G, we start by giving a rough approximation of the "size" of K by comparing it to a reference set V, which is an open neighborhood of `(1 : G)`. We can consider all left translates `(λ h, g * h) ⁻¹' V` of V, which is an open covering of K. Since K is compact, we only need finitely many left translates of V to cover K. Let `index K V` be the smallest number of left-translates of V needed to cover K. This is often denoted $(K : V)$. We did not use this in Lean code, since it conflicts with the typing-notation in Lean. In Lean, this notion is defined for arbitrary sets K and V, and it is defined to be 0 if there is no finite number of left translates covering K:

```
def index (K V : set G) : ℕ :=
Inf (finset.card '' {t : finset G | K ⊆ ⋃ g ∈ t, (λ h, g * h) ⁻¹' V })
```

For the rest of this section, we fix K₀ as an arbitrary compact set with nonempty interior (in mathlib this is denoted `K₀ : positive_compacts G`). We then define a weighted version of the index:

```
def prehaar (K₀ U : set G) (K : compacts G) : ℝ :=
(index K U : ℝ) / index K₀ U
```

This definition satisfies the following properties.

▶ **Lemma 4.** *Denote `prehaar K₀ U K` by $h_U(K)$ (recall that K_0 is fixed). Then we have*
- $(K : U) \leq (K : K_0) \cdot (K_0 : U)$;
- $0 \leq h_U(K) \leq (K : K_0)$;
- $h_U(K_0) = 1$;
- $h_U(xK) = h_U(K)$;
- *if* $K \subseteq K'$ *then* $h_U(K) \leq h_U(K')$;
- $h_U(K \cup K') \leq h_U(K) + h_U(K')$ *and equality holds if* $KU^{-1} \cap K'U^{-1} = \emptyset$.

To define the left Haar measure, we next want to define the "limit" of this quotient as U becomes a smaller and smaller open neighborhood of 1. This is not an actual limit, but we will emulate it by constructing a product space that contains all these functions, and then use a compactness argument to find a "limit" in a large product space.

Consider the product of closed intervals

$$\prod_{K \subseteq G \text{ compact}} [0, (K : K_0)].$$

In Lean this is defined as a subspace of the topological space `compacts G → ℝ`:

```
def haar_product (K₀ : set G) : set (compacts G → ℝ) :=
pi univ (λ K : compacts G, Icc 0 (index K K₀))
```

Here `pi univ` is the product of sets and `Icc` is an interval that is closed on both sides.

Note that by Tychonoff's theorem `haar_product K₀` is compact, and that every function `prehaar K₀ U : compacts G → ℝ` determines a point in `haar_product K₀`.

Given an open neighborhood `V` of 1, we can define the collection of points determined by `prehaar K₀ U` for all `U ⊆ V` and take its closure:

```
def cl_prehaar (K₀ : set G) (V : open_nhds_of (1 : G)) :
  set (compacts G → ℝ) :=
closure (prehaar K₀ '' { U : set G | U ⊆ V ∧ is_open U ∧ (1 : G) ∈ U })
```

Now we claim that the intersection of `cl_prehaar K₀ V` is nonempty for all open neighborhoods `V` of 1.

```
lemma nonempty_Inter_cl_prehaar (K₀ : positive_compacts G) : nonempty
  (haar_product K₀ ∩ ⋂ (V : open_nhds_of (1 : G)), cl_prehaar K₀ V)
```

Proof. Since `haar_product K₀` is compact, it is sufficient to show that if we have a finite collection of open neighborhoods `t : finset (open_nhds_of 1)` the set `haar_product K₀ ∩ ⋂ (V ∈ t), cl_prehaar K₀ V` is non-empty. In this case we can explicitly give an element, namely `prehaar K₀ V₀`, where `V₀ = ⋂ (V ∈ t), V`. ◀

Finally, we can choose an arbitrary element in the set of the previous lemma, which we call `chaar K₀`, since it measures the size of the compact sets of `G`:

```
def chaar (K₀ : positive_compacts G) (K : compacts G) : ℝ :=
classical.some (nonempty_Inter_cl_prehaar K₀) K
```

▶ **Lemma 5.** *Denote* `chaar K₀ K` *by* $h(K)$. *Then we have*
- $0 \leq h(K)$ *and* $h(\emptyset) = 0$ *and* $h(K_0) = 1$;
- $h(xK) = h(K)$;
- *if* $K \subseteq K'$ *then* $h(K) \leq h(K')$ *(monotonicity);*
- $h(K \cup K') \leq h(K) + h(K')$ *(subadditivity) and equality holds if* $K \cap K' = \emptyset$ *(additivity).*

The idea in the proof is that the projections $f \mapsto f(K)$ are continuous functions. Since these statements in this lemma hold for all h_U by Lemma 4, they also hold for each point in each set `cl_prehaar K₀ V`, and therefore for h.

This function `chaar K₀` measures the size of the compact sets of `G`. It has the properties of a *content* on the compact sets, which is an additive, subadditive and monotone function into the nonnegative reals [10, §53]. From this we can define the Haar measure in three steps.

First, given a content, we can obtain its associated *inner content* on the open sets,

```
def inner_content (h : compacts G → ennreal) (U : opens G) : ennreal :=
⨆ (K : compacts G) (hK : K ⊆ U), h K
```

This inner content is monotone and countably subadditive. Also note that for every set `K` that is both compact and open, we have `inner_content h K = h K`.

Second, given an arbitrary inner content ν, we can obtain its associated outer measure μ which is given by `μ A = ⨅ (U : set G) (hU : is_open U) (h : A ⊆ U), ν U`.

For open sets `U` we have that `μ U = ν U`. Suppose that the inner content ν comes from the content `h`. We have observed that μ agrees with ν on their common domain (the open sets), and that ν agrees with `h` on their common domain (the compact open sets). One might be tempted to conclude that μ agrees with `h` on their common domain (the compact sets). This is implicitly claimed in Gleason [8], since the same name is used for these three functions. However, this is not necessarily the case [10, p. 233]. The best we can say in general is that for compact `K` we have `μ (interior K) ≤ h K ≤ μ K`

If we apply these two steps to the function `h = chaar K₀` we obtain the Haar outer measure `haar_outer_measure K₀` on `G`. Next, we show that all Borel measurable sets on `G` are Carathéodory measurable w.r.t. `haar_outer_measure K₀`. This gives the *left Haar measure.* We also scale the Haar measure so that it assigns measure 1 to `K₀`.

▶ **Theorem 6.** *Let μ be the left Haar measure on G, defined above. Then μ is a left invariant and regular measure satisfying $\mu(K_0) = 1$.*

We will show in Section 6 that the Haar measure is unique, up to multiplication determined by a constant. The following theorem states some properties of left invariant regular measures.

▶ **Theorem 7.** *Let μ be a nonzero left invariant regular measure on G.*
- *If U is a nonempty open set, then $\mu(U) > 0$*
- *If $f : G \to \mathbb{R}$ is a nonnegative continuous function, not identically equal to 0, then $\int^{-} f \, d\mu > 0$.*

In this section we have only discussed the left Haar measure, which is left invariant. One can similarly define the notion of right invariance for a measure. We can easily translate between these two notions. For a measure μ define the measure $\check{\mu}$ by $\check{\mu}(A) = \mu(A^{-1})$. The following Theorem shows that if μ is the left Haar measure on G, then $\check{\mu}$ is a right Haar measure on G (i.e. a nonzero regular right invariant measure).

▶ **Theorem 8.**
- *The operation $\mu \mapsto \check{\mu}$ is involutive.*
- *μ is left invariant iff $\check{\mu}$ is right invariant.*
- *μ is regular iff $\check{\mu}$ is regular.*

6 Uniqueness of the Haar measure

In this section we will show that the Haar measure is unique, up to multiplication by a constant. For this proof, we followed the proof in Halmos [10, §59-60] up to some differences given at the end of the section.

Let G be a second-countable locally compact group. The main ingredient in the proof is that if we have two left invariant measures μ and ν on G, we can transform expressions involving μ into expressions involving ν by applying a transformation to $G \times G$ that preserves the measure $\mu \times \nu$. We need Tonelli's theorem to compute the integrals in this product. Because G is second countable, the product σ-algebra on $G \times G$ coincides with the Borel σ-algebra.

Suppose μ and ν are left invariant and regular measures on G. Since G is second-countable, this also means that μ and ν are σ-finite. We first show that $(x, y) \mapsto (x, xy)$ and $(x, y) \mapsto (yx, x^{-1})$ are measure-preserving transformations.

▶ **Lemma 9.** *Let μ and ν be left invariant regular measures on G, then the transformations $S, T : G \times G \to G \times G$ given by $S(x, y) := (x, xy)$ and $T(x, y) := (yx, x^{-1})$ preserve the measure $\mu \times \nu$, i.e. $S_*(\mu \times \nu) = \mu \times \nu = T_*(\mu \times \nu)$.*

Proof. It suffices to show that the measures are equal on measurable rectangles, since these generate the σ-algebra on $G \times G$. Let $A \times B \subseteq G \times G$ be a measurable rectangle. For S we compute:

$$\begin{aligned} S_*(\mu \times \nu)(A \times B) &= (\mu \times \nu)(S^{-1}(A \times B)) \\ &= \int^- \nu(S(x, -)^{-1}(A \times B))\,\mathrm{d}\mu(x) = \int^- \chi_A(x)\nu(x^{-1}B)\,\mathrm{d}\mu(x) \\ &= \int^- \chi_A(x)\nu(B)\,\mathrm{d}\mu(x) = \mu(A)\nu(B) = (\mu \times \nu)(A \times B). \end{aligned}$$

Note that S is an equivalence with inverse $S^{-1}(x, y) = (x, x^{-1}y)$. If we define $R(x, y) := (y, x)$, we saw in Section 4 that $R_*(\mu \times \nu) = \nu \times \mu$. The claim that T preserves measure now follows from the observation that $T = S^{-1}RSR$, which is easy to check. ◀

For measurable sets A its left translates have the same measure: $\mu(xA) = \mu(A)$. This might not be true for right translates $\mu(Ax)$, but we can say the following [10, §59, Th. D].

▶ **Lemma 10.** *Let μ be a left invariant regular measure on G and suppose that $\mu(A) > 0$ for a measurable set $A \subseteq G$. Then $\mu(A^{-1}) > 0$ and $\mu(Ax) > 0$ for $x \in G$. Furthermore, the map $f(x) := \mu(Ax)$ is measurable.*

Proof. Let S and T be as in Lemma 9. For the first claim, we will show that $\mu(A^{-1}) = 0$ implies that $\mu(A) = 0$. It suffices to show that $(\mu \times \mu)(A \times A) = 0$, which we can compute using Lemma 9:

$$\begin{aligned} (\mu \times \mu)(A \times A) = (\mu \times \mu)(T^{-1}(A \times A)) &= \int^- \mu((T^{-1}(A \times A))^y)\,\mathrm{d}\mu(y) \\ &= \int^- \mu(y^{-1}A \cap A^{-1})\,\mathrm{d}\mu(y) \le \int^- \mu(A^{-1})\,\mathrm{d}\mu(y) = 0. \end{aligned}$$

The second claim now easily follows, since $\mu(A) > 0$ implies that $\mu(Ax) = \mu((x^{-1}A^{-1})^{-1}) > 0$.

For the final claim, we can see that $\mu(Ax) = \mu((S^{-1}(G \times A))^x)$, which is measurable by the symmetric version of `measurable_measure_prod_mk_left` from Section 4. ◀

The following is a technical lemma that allows us to rewrite ν-integrals into μ-integrals. This is the key lemma to show the uniqueness of the Haar measure. [10, §60, Th. A].

▶ **Lemma 11.** *Let μ and ν be left invariant regular measures on G, let $K \subseteq G$ be compact with $\nu(K) > 0$, and suppose that $f : G \to [0, \infty]$ is measurable. Then*

$$\mu(K) \cdot \int^- \frac{f(y^{-1})}{\nu(Ky)}\,\mathrm{d}\nu(y) = \int^- f\,\mathrm{d}\mu.$$

Proof. First note that $0 < \nu(Ky) < \infty$ for any y. The first inequality follows from Lemma 10 and the second inequality from the fact that ν is regular and Ky is compact. Let $g(y) := \frac{f(y^{-1})}{\nu(Ky)}$, then by Lemma 10, g is measurable. Now we compute

$$\mu(K) \cdot \int^{-} g(y)\,\mathrm{d}\nu(y) = \int^{-} \chi_K(x)\,\mathrm{d}\mu(x) \cdot \int^{-} g(y)\,\mathrm{d}\nu(y)$$
$$= \int^{-}\!\!\int^{-} \chi_K(x)g(y)\,\mathrm{d}\nu(y)\,\mathrm{d}\mu(x) = \int^{-}\!\!\int^{-} \chi_K(yx)g(x^{-1})\,\mathrm{d}\nu(y)\,\mathrm{d}\mu(x) \quad \text{(by Lemma 9)}$$
$$= \int^{-} \nu(Kx^{-1})g(x^{-1})\,\mathrm{d}\mu(x) = \int^{-} f(x)\,\mathrm{d}\mu(x).$$

Note that in the third step we also use Tonelli's theorem, since Lemma 9 shows that the corresponding integrals over the product are equal. ◀

As a consequence, we get the uniqueness of the Haar measure: every left invariant measure is a multiple of the Haar measure.

▶ **Theorem 12.** *Let ν be a left invariant regular measure and K_0 be a compact set with non-empty interior. If μ is the Haar measure on G with $\mu(K_0) = 1$, then $\nu = \nu(K_0) \cdot \mu$.*

Proof. Let A be any measurable set. The result follows from the following computation, where we apply Lemma 11 twice, once with measures (ν, μ) and once with measures (μ, μ):

$$\nu(A) = \int^{-} \chi_A\,\mathrm{d}\nu = \nu(K_0) \cdot \int^{-} \frac{\chi_A(y^{-1})}{\mu(K_0 y)}\,\mathrm{d}\mu(y)$$
$$= \nu(K_0) \cdot \mu(K_0) \cdot \int^{-} \frac{\chi_A(y^{-1})}{\mu(K_0 y)}\,\mathrm{d}\mu(y) = \nu(K_0) \cdot \int^{-} \chi_A\,\mathrm{d}\mu = \nu(K_0) \cdot \mu(A).$$ ◀

For this proof we followed Halmos [10, §59-60], but there are some differences. Halmos gives a more general version of Lemma 11, for an arbitrary measurable group, while we did it for a second-countable locally compact group equipped with the Borel σ-algebra (which forms a measurable group). There are other proofs, like in Cohn [3, Theorem 9.2.6], that do not require G to be second countable. Another difference between our and Halmos' proof is that we assume that K is compact, while Halmos assumes that K is measurable with $\nu(K) < \infty$. However, in the proof he implicitly uses that also $\nu(Ky) < \infty$, a fact that is not justified in the text, and that I was unable to prove from the assumption that $\nu(K) < \infty$. At this point I do not know whether this claim is true or not. I asked this question on Stack Exchange, which has not been answered as of writing this paper. Lemma 10 claims something similar, namely that if $\nu(K) > 0$ then $\nu(Ky) > 0$, but the proof of this claim does not work to show finiteness of $\nu(Ky)$. We patched this gap by assuming that K is compact and ν is regular, which implies that $\nu(Ky) < \infty$ because Ky is also compact.

7 Concluding Thoughts

The formalization of product measures (and their basic properties) is about 900 lines of code, and the formalization of Haar measure (and its basic properties) is about 1300 lines of code (including blank lines and comments). The development integrates smoothly into mathlib, and preliminaries of this work are placed in their appropriate files. The formalization was integrated into mathlib via more than a dozen pull requests that added new material or modified and expanded existing developments. Each pull request was carefully reviewed by one of mathlib's maintainers.

The proofs in this formalization use a wide variety of techniques, from the reasoning in Banach spaces for Fubini's theorem, to topological compactness arguments in the existence proof of the Haar measure, to the calculational proofs in the uniqueness of the Haar measure. This formalization shows that these different proof techniques can be used and combined effectively in mathlib to formalize graduate level mathematics.

This work can be extended in various ways. One interesting project would be to prove the uniqueness for groups that need not be second countable. This should not be too difficult, but requires a version of Fubini's theorem for compactly supported functions on locally compact Hausdorff spaces. Another project would be to explore the *modular function*, which describes how the left and the right Haar measures differ on a locally compact group. Furthermore, the Haar measure is used as a tool in many areas of mathematics, some of which would now be feasible to formalize with the Haar measure as a new tool in our toolkit.

One such area is abstract harmonic analysis. Abstract harmonic analysis is the area of analysis on topological groups, usually locally compact groups. One of the core ideas is to generalize the Fourier transform to an arbitrary locally compact abelian group. The special case of the continuous Fourier transform has been formalized in HOL4 [9], but the abstract Fourier transform has never been formalized. One specific goal could be to formalize *Pontryagin duality*. For a locally compact abelian group G we can define the *Pontryagin dual* $\widehat{G} := \mathrm{Hom}(G, T)$ consisting of the continuous group homomorphisms from G to the circle group T. Then Pontryagin duality states that the canonical map $G \to \widehat{\widehat{G}}$ (given by the evaluation function) is an isomorphism. Other targets include Peter–Weyl theorem and more ambitiously the representation theory of compact groups and Weyl's integral formula for compact Lie groups.

In mathlib we have not just defined binary product measures, but also finitary product measures, which can be used to define integrals over $\mathbb{R}^n$. In mathlib this is done by defining a measure on the product space `Π i : ι, X i`.

```
def measure.pi {ι : Type*} [fintype ι] {X : ι → Type*}
  [∀ i, measurable_space (X i)] (μ : ∀ i, measure (X i)) :
  measure (Π i, X i)
```

The definition is conceptually very simple, since you just iterate the construction of binary product measures, but in practice it is quite tricky to do it in a way that is convenient to use. For example, in a complicated proof, you do not want to worry about the fact that the spaces $X \times (Y \times Z)$ and $(X \times Y) \times Z$ are not exactly the same, just equivalent. One open question is whether we can formulate Fubini's theorem for finitary product measures in a way that is convenient to use. It would be inconvenient to rewrite a finitary product of measures into a binary product in the desired way, and then apply Fubini's theorem for binary product measures. We have an idea that we expect to be more promising, which is to define a "marginal integral" that only integrates some variables in the input of a function, corresponding to the marginal distribution of a random variable in probability theory. It takes a function `f : (Π i, X i) → E` and integrates away the variables in a subset `I`. The result is another function `(Π i, X i) → E` that is constant on all variables in `I`:

```
def marginal (μ : ∀ i, measure (X i)) (I : finset ι)
  (f : (Π i, X i) → E) : (Π i, X i) → E
```

This has not been incorporated in mathlib, but it is a promising experiment on a separate branch. If we denote `marginal μ I f` as `∫⋯∫_I, f ∂μ`, then the following is a convenient way to state Fubini's theorem for finitary product measures.

```
lemma marginal_union (f : (Π i, X i) → E) (I J : finset ι) :
  disjoint I J → ∫⋯∫_I ∪ J, f ∂μ = ∫⋯∫_ I, ∫⋯∫_J, f ∂μ ∂μ
```

References

1 Jeremy Avigad, Johannes Hölzl, and Luke Serafin. A formally verified proof of the central limit theorem. *Journal of Automated Reasoning*, 59(4):389–423, 2017.

2 Aaron R. Coble. Anonymity, information, and machine-assisted proof. Technical Report UCAM-CL-TR-785, University of Cambridge, Computer Laboratory, 2010. URL: `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-785.pdf`.

3 Donald L. Cohn. *Measure Theory.* Birkhäuser Basel, 2 edition, 2013.

4 The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 367–381, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3372885.3373824`.

5 Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (system description). *CADE-25*, pages 378–388, 2015.

6 Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *Proc. ACM Program. Lang.*, 1(ICFP):34:1–34:29, August 2017. `doi:10.1145/3110278`.

7 Noboru Endou, Keiko Narita, and Yasunari Shidama. The Lebesgue monotone convergence theorem. *Formalized Mathematics*, 16(2):167–175, 2008.

8 Jonathan Gleason. Existence and uniqueness of Haar measure. preprint, 2010.

9 Yong Guan, Jie Zhang, Zhiping Shi, Yi Wang, and Yongdong Li. Formalization of continuous Fourier transform in verifying applications for dependable cyber-physical systems. *Journal of Systems Architecture*, 106:101707, 2020. `doi:10.1016/j.sysarc.2020.101707`.

10 Paul R. Halmos. *Measure theory.* Springer-Verlag New York, 1950. `doi:10.1007/978-1-4684-9440-2`.

11 John Harrison. The HOL Light theory of Euclidean space. *Journal of Automated Reasoning*, 50(2):173–190, 2013. `doi:10.1007/s10817-012-9250-9`.

12 Johannes Hölzl and Armin Heller. Three chapters of measure theory in Isabelle/HOL. In *Interactive Theorem Proving*, pages 135–151, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

13 Joe Hurd. Formal verification of probabilistic algorithms. Technical Report UCAM-CL-TR-566, University of Cambridge, Computer Laboratory, 2003. URL: `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-566.pdf`.

14 David R Lester. Topology in PVS: Continuous mathematics with applications. In *Proceedings of the Second Workshop on Automated Formal Methods*, AFM '07, page 11–20, New York, NY, USA, 2007. Association for Computing Machinery. `doi:10.1145/1345169.1345171`.

15 Tarek Mhamdi, Osman Hasan, and Sofiène Tahar. Formalization of measure theory and Lebesgue integration for probabilistic analysis in HOL. *ACM Trans. Embed. Comput. Syst.*, 12(1), January 2013. `doi:10.1145/2406336.2406349`.

16 Jan Mikusiński. The bochner integral. In *The Bochner Integral*, pages 15–22. Springer, 1978.

17 Stefan Richter. Formalizing integration theory with an application to probabilistic algorithms. In *Theorem Proving in Higher Order Logics*, pages 271–286, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

18 B.A.W. Spitters and E.E. van der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21:795–825, 2011.

19 Joseph Tassarotti. coq-proba probability library, 2019. URL: `https://github.com/jtassarotti/coq-proba`.

20 P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, page 60–76, New York, NY, USA, 1989. Association for Computing Machinery. `doi:10.1145/75277.75283`.

A Mechanised Proof of the Time Invariance Thesis for the Weak Call-By-Value λ-Calculus

Yannick Forster
Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

Fabian Kunze
Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

Gert Smolka
Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

Maximilian Wuttke
Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

Abstract

The weak call-by-value λ-calculus L and Turing machines can simulate each other with a polynomial overhead in time. This time invariance thesis for L, where the number of β-reductions of a computation is taken as its time complexity, is the culmination of a 25-years line of research, combining work by Blelloch, Greiner, Dal Lago, Martini, Accattoli, Forster, Kunze, Roth, and Smolka. The present paper presents a mechanised proof of the time invariance thesis for L, constituting the first mechanised equivalence proof between two standard models of computation covering time complexity.

The mechanisation builds on an existing framework for the extraction of Coq functions to L and contributes a novel Hoare logic framework for the verification of Turing machines.

The mechanised proof of the time invariance thesis establishes L as model for future developments of mechanised computational complexity theory regarding time. It can also be seen as a non-trivial but elementary case study of time-complexity-preserving translations between a functional language and a sequential machine model. As a by-product, we obtain a mechanised many-one equivalence proof of the halting problems for L and Turing machines, which we contribute to the Coq Library of Undecidability Proofs.

2012 ACM Subject Classification Theory of computation → Computability; Theory of computation → Type theory

Keywords and phrases formalizations of computational models, computability theory, Coq, time complexity, Turing machines, lambda calculus, Hoare logic

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.19

Supplementary Material *Software (Code Repository)*: `https://github.com/uds-psl/time-invariance-thesis-for-L`; archived at `swh:1:dir:cc13c3c6279985ae15903a24e23bd5d356b6f435`

Acknowledgements We want to thank Lennard Gäher and Dominik Kirst for valuable feedback on drafts of this paper.

1 Introduction

Computability theory – i.e. the study of which problems can be solved computationally – is invariant under the chosen model of computation: Any Turing-complete model does the job. Similarly, but less generally, computational complexity theory – i.e. the study of how efficiently problems can be solved computationally – is invariant under the chosen model: Both Turing machines and RAM machines are used and can be exchanged as long as only complexity classes closed under polynomial time reductions are of interest. This practice is based on the *invariance thesis*, introduced by Slot and van Emde Boas as “all reasonable models of computation simulate each other with a polynomially bounded overhead in time and a constant factor overhead in space” [23]. For the present paper, we dub the first half concerning time complexity the *time invariance thesis*.

While RAM machines and Turing machines are reasonable in this sense, the question how and whether the (untyped) λ-calculus [4] is reasonable was a long-standing open question. This may sound especially surprising given the fact that the λ-calculus was one of the first models for computability to be proven Turing-complete, by Turing himself [27]. On the other hand, the λ-calculus is indeed more complicated than sequential models: Terms are trees and contain binders, computation is non-local, multiple potentially non-equivalent reduction strategies can be chosen, etc. Maybe most crucially, it is not obvious what a reasonable time complexity measure is: The number of β-steps in a computation? Or does one have to account for (the size of) β-redexes? And even more unclear: What is the space complexity of a λ-calculus computation?

More or less independently of these questions, one direction of the time invariance thesis is easy to prove: The λ-calculus can simulate Turing machines.

In this paper, we focus on the weak call-by-value λ-calculus introduced by Plotkin [21], in the concrete variant L introduced by Forster and Smolka [12]. In L, only abstractions are values, and we take the number of β-steps as time complexity measure of a computation [7]. L is similar to an ML-like functional programming language: Reductions in abstractions are not allowed, and a β-reduction only applies when both sides of an application are a value.

A short timeline of time complexity for L and the full λ-calculus looks as follows:

1995 Blelloch and Greiner [3] prove the time invariance thesis for the weak call-by-value λ-calculus w.r.t. RAM machines, with the number of β-steps as time complexity measure.

2008 Dal Lago and Martini [17] prove the time invariance thesis for the weak call-by-value λ-calculus w.r.t. Turing machines, with the sum over all differences of the size of the redex and the size of the reduct as time complexity measure.

2015 Accattoli and Dal Lago [1] prove the time invariance thesis for the full λ-calculus w.r.t. RAM machines, with the number of β-steps as time complexity measure.

2020 Forster, Kunze, and Roth [7] prove the invariance thesis for L w.r.t. Turing machines, with the number of β-steps as time complexity measure and the maximum of the sizes of all intermediate terms as space complexity measure.

All of these results come with different levels of formality in their proofs. Blelloch and Greiner [3] only *state* their result, but do not give any details of a proof, much less provide details how the (RAM)-machine carrying out the simulation might look like. Dal Lago and Martini [17] give a *proof sketch*: They informally describe what a Turing machine simulating L has to do, and for instance how many tapes it has, but do not give invariants for inductions. The other, easier simulation direction is discussed in similar detail. In a later note, Dal Lago and Accattoli [16] give a *formal* proof on paper that the λ-calculus – independent from the concrete variant used – can simulate Turing machines with linear overhead. The proof of the time invariance thesis for the full λ-calculus by Accattoli and Dal Lago [1] is based on a careful and *formal* study of explicit sharing, but again the implementation in terms of machines is left out. This technique is omnipresent in theoretical computer science and mathematics: The folklore parts of results are only sketched or are entirely left out, while the interesting parts are formalised and detailed proofs are given.

When mechanising a result in an interactive theorem prover, both aspects form their own challenges: For folklore results, first a proof has to be found, then formalised, then mechanised, and each individual step can prove challenging. For a formal proof, only missing details of an argument have to be recovered, which can still impose challenges.

The proof of the invariance thesis for L [7] is accompanied by a mechanisation for one part of the novel contribution, namely two stack machine semantics for L. Since it is folklore that concrete algorithms can be implemented on Turing machines, this part is only sketched.

In the present paper, we give a mechanised proof of the time invariance thesis for L in Coq [26], providing all details left out in existing proofs. We mechanise two variants of the time invariance thesis for L. The first provides a Turing machine M_{sim}, simulating L based on a stack machine for L with a heap, and vice versa a term s_{sim} simulating Turing machines:

▶ **Theorem 1** (Time Invariance Thesis for L w.r.t. Simulation). *There are a Turing machine M_{sim}, an L-term s_{sim}, and encoding relations of L-terms and heaps on tapes, and Turing machines and tapes as L-terms s.t.*

1. *M_{sim} simulates L with polynomial overhead, i.e. there is a polynomial p s.t. for all closed s:*
   a. *Whenever s terminates in i steps with value v, M_{sim} run on an input tape encoding s terminates in $p(i, \|s\|)$ steps with an encoding of a heap containing v on its output tape, where $\|s\|$ is the size of s.*
   b. *If M_{sim} terminates on the encoding s, then s terminates.*
2. *s_{sim} simulates TMs with linear overhead, i.e. there is a constant c s.t. for all M:*
   a. *Whenever $M : \mathsf{TM}^n_\Sigma$ terminates on tapes $\boldsymbol{t}$ in i steps with result tapes $\boldsymbol{t'}$, s_{sim} run on an encoding of both M and $\boldsymbol{t}$ evaluates to the encoding of $\boldsymbol{t'}$ in $c \cdot i \cdot \|M\| + c$ steps.*
   b. *If s_{sim} terminates on the encodings M and $\boldsymbol{t}$, then M terminates on $\boldsymbol{t}$.*

As a by-product, this theorem yields the first mechanised proof that the halting problems for L and Turing machines are many-one equivalent, which we contribute as a corner stone to the Coq Library of Undecidability Proofs [11].

Two subtleties are important to point out: First, an L-computation does not have explicit input and output. Any L-term s_0 can compute on its own. Input can be realised by application to an (encoded) value $(s_0\overline{n})$ and the result of the computation can be considered its output. In contrast, a Turing machine M can not compute without the value of its tapes specified, unless one explicitly defines the canonical value of tapes to be e.g. empty. Secondly, the theorem depends on notions of encoding an L-term on a tape, of encoding heaps on a tape, and of encoding a Turing machine and tapes as L-terms. The choice of such encodings is not canonical, and they have to fulfil certain properties for the theorem to be meaningful. For instance, the unfolding of a heap to an L-term on a Turing machine has to be (at most) polynomial in time, otherwise the theorem is meaningless.

As is well-known, computation in L and the λ-calculus in general is potentially subject to so-called "size explosion", i.e. the size of a result of a computation can be exponentially larger than both the size of the input and the number of steps. Thus, in the above theorem, unfolding the heap containing v will be polynomial in the size of v and i, but the size of v might be exponential in both the size of s and i. The following version of the time invariance thesis abstracts away from such subtleties by only considering the computability of k-ary relations on boolean strings, while still being as transparent as possible:

▶ **Theorem 2** (Time Invariance Thesis for L w.r.t Computability). *Let $R \subseteq (\mathbb{LB})^k \times \mathbb{LB}$. Then*

1. *If R is L-computable with time complexity function τ_R, then there is a polynomial p s.t. R is TM-computable with time complexity function $(m, n_1, \ldots, n_k) \mapsto p(m, n_1, \ldots, n_k, \tau_R(n_1, \ldots, n_k))$.*
2. *If R is TM-computable with time complexity function τ_R, then there is a constant c s.t. R is L-computable with time complexity function $(m, n_1, \ldots, n_k) \mapsto c \cdot m \cdot n_1 \cdots n_k \cdot \tau_R(n_1, \ldots, n_k) + c$.*

We accomodate for size explosion by making the time complexity function of a relation R also depend on the size m of the output. Of course, for Turing machines m is always bounded linearly by $n_1, \ldots, n_k$ and i.

Two corollaries are immediate: If the output of a relation R is polynomially bounded by its input, R is polynomially TM-computable if and only if it is polynomially L-computable. In particular, the complexity class P (i.e. PTIME) agrees for both L and Turing machines.

Note that the theorem still depends on encodings, namely on the precise definitions of a relation R being TM- and L-computable. However, the two notions can be assessed independently: L-computability only depends on how to encode boolean strings ($\mathbb{LB}$), and similar for TM-computability. Unsurprisingly, both encodings are very simple, and it is obvious that e.g. equality checking works in linear time.

The theorems are furthermore equivalent, in the sense that one can prove one from the other without additional complex simulations: We obtain our mechanised proof of the time invariance thesis w.r.t. computability from the time invariance thesis w.r.t. simulation, by defining terms and machines dynamically exchanging between the encodings of $\mathbb{LB}$ on Turing machines, $\mathbb{LB}$ in L, L-terms on Turing machines, and Turing machines as L-terms. The other direction would be possible by verifying a universal Turing machine with polynomial overhead in time for Turing machine computation, and similarly a universal L-term.

We avoid non-polynomial overhead for terms exhibiting size explosion by relying on heaps. However, this means that the space overhead is linear-logarithmic rather than constant factor, due to terms exhibiting pointer explosion. A mechanised proof of the time and space invariance thesis for L is left for future work.

The present work is based on the certifying extraction framework for L by Forster and Kunze [6] and the Turing machine verification framework by Forster, Kunze, and Wuttke [9].

The certifying extraction framework allows extracting (by definition total) Coq functions on first-order types to L and automatically proves correctness. The user can provide a time complexity function, which is then automatically verified as well. We use an L-computability proof of Turing machine transitions, which is already a case study of the framework.

The Turing machine verification framework allows giving algorithms in the style of a register-based while-language, and a corresponding machine is automatically constructed behind the scenes. Separate correctness and verification proofs are then inclusion proofs between the automatically derived and the user-given relations for the constructed machine.

Contribution. The main contribution of the paper are mechanised proofs of the time invariance thesis for L w.r.t. both simulation and computability. As a by-product, we obtain that the halting problems for Turing machines and L are many-one equivalent, a result we contribute to the Coq Library of Undecidability Proofs [11]. We also contribute a Hoare logic framework for the verification of correctness, time, and space complexity of Turing machines.

Outline. The paper is split into four parts: First, Section 3 introduces the weak call-by-value λ-calculus L with small-step, big-step, and stack machine semantics, and Section 4 introduces Turing machines and the Turing machine verification framework from [9]. Secondly, we explain the simulation of L computations on Turing machines with polynomial overhead in Section 5, and how to obtain a simulator s_{sim} from the L-computability of Turing machine steps [6] in Section 6, which yields a mechanised proof of the time invariance thesis for L w.r.t. simulation. Thirdly, we explain how to prove that TM-computable relations $R \subseteq (\mathbb{LB})^k \times \mathbb{LB}$ are L-computable with polynomial overhead in Section 7. Lastly, we introduce a novel Hoare logic verification framework for Turing machines in Section 8, which we use to prove that L-computable relations $R \subseteq (\mathbb{LB})^k \times \mathbb{LB}$ are TM-computable with polynomial overhead in Section 9, which yields a mechanised proof of the time invariance thesis for L w.r.t. computability.

Related work. Considering only the simulation of models of computation without time complexity, we are aware of Xu et al. [29] mechanising the equivalence of register machines and Turing machines, Forster and Larchey-Wendling [10] with a mechanised compilation of register machines to binary stack machines, and Larchey-Wendling and Forster [18], proving that the halting problems of register machines and μ-recursive functions, and the solvability of diophantine equations are many-one equivalent. In unpublished work, Pous [22] mechanises an equivalence proof between counter machines and partial recursive functions in Coq.

2 Notations and Definitions

We use the following inductive types:

$\mathbb{1} ::= ()$	(unit)	$o : \mathbb{O}A ::= \mathsf{None} \mid \mathsf{Some}\, a$	(options)
$n : \mathbb{N} ::= 0 \mid \mathsf{S}\, n$	(natural numbers)	$l : \mathbb{L}A ::= [] \mid a :: l$	(lists)
$b : \mathbb{B} ::= \mathsf{true} \mid \mathsf{false}$	(booleans)	$A + B := \mathsf{inl}\, a \mid \mathsf{inr}\, b$	(sums)
		$A \times B := (a, b)$	(pairs)

We use the notation **if** a **is** s **then** b_1 **else** b_2 for inline case analysis, which evaluates to b_1 if a is of the shape s (e.g. for a non-zero number $s := \mathsf{S}n$ or for a list $s := []$ or $s := x :: l$), and to b_2 otherwise.

We use the functions $\mathsf{map} : (A \to B) \to \mathbb{L}A \to \mathbb{L}B$ and $\mathsf{map}_2 : (A \to B \to C) \to \mathbb{L}A \to \mathbb{L}B \to \mathbb{L}C$, defined as follows

$$\mathsf{map}\, f\, (a :: l) := fa :: \mathsf{map}\, f\, l \qquad \mathsf{map}_2\, f\, (a :: l_1)\, (b :: l_2) := fab :: \mathsf{map}_2\, f\, l_1\, l_2$$
$$\mathsf{map}\, f\, [] := [] \qquad \mathsf{map}_2\, f\, l_1\, l_2 := []$$

We use i, n, k, and m as letters for numbers, but try to be consistent in there use: numbers of steps are i, number of tapes are n, the arity of input and relations is k, the size of inputs are $n_1, \dots, n_k$ and the size of the output is m.

We write $\mathbb{P}$ for the type of propositions. $R \subseteq A \times B$ is short for $R : A \to B \to \mathbb{P}$, and $P \subseteq B$ is short for $P : B \to \mathbb{P}$. In particular, if $R \subseteq A \times B$ and $a : A$, then $Ra \subseteq B$.

A^k is the type of vectors of length k with elements in A. We use bold letters ($\boldsymbol{t}, \boldsymbol{u}, \dots$) for vectors and reuse list functions such as map_2 for vectors. The type Fin_n is inductively defined to have exactly n elements. We write the elements of the type $\mathsf{Fin}_{\mathsf{S}n}$ as $0, \dots, n$.

A retraction $X \hookrightarrow Y$ consists of a function $I : X \to Y$ and an inverse function $R : Y \to \mathbb{O}X$ s.t. $\forall x.\ R(Ix) = \mathsf{Some}\, x$.

3 The call-by-value λ-calculus L

The call-by-value λ-calculus was introduced by Plotkin [21] as variant of Church's λ-calculus [5]. The concrete variant of the call-by-value λ-calculus we present here is called L [12]. We define syntax and semantics for L in Section 3.1 and introduce a stack machine with a heap in 3.2.

3.1 Syntax, small-step, and big-step semantics

We define the syntax of L using de Bruijn indices as the syntax of the full λ-calculus, i.e. as variables, applications, or abstractions. The size $\|s\|$ counts the number of constructors in s with unary encoded de Bruijn indices.

$$s, t, u : \mathsf{tm_L} ::= n : \mathbb{N} \mid st \mid \lambda s \qquad \|n\| := n + 1 \qquad \|st\| := \|s\| + \|t\| + 1 \qquad \|\lambda s\| := 1 + \|s\|$$

We use names for concrete terms on paper, e.g. write $(\lambda xy.xx)(\lambda z.z)$ for $(\lambda\lambda 11)(\lambda 0)$.

We define a simple substitution operation s^n_t, agreeing with a more standard parallel substitution operation when the term substituted with is closed:

$$n^m_u := \textbf{if } n = m \textbf{ then } u \textbf{ else } n \qquad (st)^n_u := s^n_u t^n_u \qquad (\lambda s)^n_u := \lambda(s^{\mathsf{S}n}_u)$$

Formally, we say that a term s is a *closed term* if $\forall nu.\ s^n_u = s$.

We now define a small-step relation $\succ$, its n-step repetition $\succ^n$, and an inductive characterisation of weak, call-by-value big-step evaluation $s \rhd v$:

$$\frac{}{(\lambda s)(\lambda t) \succ s^0_{\lambda t}} \quad \frac{s \succ s'}{st \succ s't} \quad \frac{t \succ t'}{st \succ st'} \quad \frac{}{s \succ^0 s} \quad \frac{s \succ s' \quad s' \succ^n t}{s \succ^{\mathsf{S}i} t} \quad \frac{}{\lambda s \rhd^0 \lambda s} \quad \frac{s \rhd^{i_1} \lambda u \quad t \rhd^{i_2} t' \quad u^0_{t'} \rhd^{i_3} v}{st \rhd^{1+i_1+i_2+i_3} v}$$

Note that we have for example $(\lambda xy.xx)(\lambda z.z) \rhd^1 \lambda y.(\lambda z.z)(\lambda z.z)$. Evaluation is called weak because the bodies of abstractions are not evaluated and call-by-value because arguments are evaluated before a function is called. Evaluation agrees with evaluation in Plotkin's calculus [21] on closed terms, but does not treat variables as values.

▶ **Lemma 3.** *If s is closed, $s \rhd^i t$ if and only if $s \succ^i t$ and t is an abstraction.*

Weak call-by-value reduction is uniformly confluent [20], meaning for a terminating term s, we can talk about *the* time complexity of s without fixing a reduction path.

▶ **Fact 4.** *If $s \succ t_1$ and $s \succ t_2$, then $t_1 = t_2$ or $\exists u. t_1 \succ u \land t_2 \succ u$.*

▶ **Corollary 5.** *If $s \rhd^{n_1} t_1$ and $s \rhd^{n_2} t_2$, then $n_1 = n_2$ and $t_1 = t_2$.*

The L *halting problem* is defined as $\mathsf{Halt_L}(s : \mathsf{tm_L}, H : \mathsf{closed} s) := \exists t.\ s \rhd t$.

To define L-computability we introduce so-called Scott encodings [14,19] for $\mathbb{B}$, $\mathbb{N}$, and $\mathbb{L}$, which internalise the case-analysis behaviour of the respective types.

$$\begin{aligned} \overline{\mathsf{true}} &:= \lambda xy.x & \overline{0} &:= \lambda xy.x & \overline{[]} &:= \lambda xy.x \\ \overline{\mathsf{false}} &:= \lambda xy.y & \overline{\mathsf{S}n} &:= \lambda xy.y\overline{n} & \overline{b :: l} &:= \lambda xy.y\,\overline{b}\,\overline{l} \end{aligned}$$

A relation $R \subseteq \mathbb{LB}^k \times \mathbb{LB}$ is L-*computable* with time complexity relation $\tau \subseteq \mathbb{N}^{1+k} \times \mathbb{N}$ if

$$\begin{aligned} &\exists s : \mathsf{tm_L}. \forall l_1 \dots l_k. (\forall l.\ R(l_1, \dots, l_k)l \to \exists c \le \tau(|l|, |l_1|, \dots, |l_k|).\ s\,\overline{l_1} \dots \overline{l_k} \rhd^c \overline{l}) \land \\ &\qquad\qquad \forall t.\ s\overline{l_1} \dots \overline{l_k} \rhd t \to \exists l.\ R(l_1, \dots, l_k)\overline{l} \end{aligned}$$

Note that time complexity is a relation rather than a function. We will only consider functional time complexity relations, and write them on paper as if they were functions. However, for undecidable relations R (e.g. expressing halting problems necessary for deducing a proof of the time invariance thesis w.r.t. simulation from the one w.r.t. computability), it is crucial that τ is a relation: one could use a complexity *function* $\tau : \mathbb{N}^{1+k} \to \mathbb{N}$ to write a Coq decision function checking whether $\exists l.\ R(l_1, \dots, l_k)l$. Since we know that such a decision is impossible in the constructive type theory of Coq, time complexity functions can not be defined for undecidable problems.

3.2 Stack machine semantics

The big-step semantics allows for a compact definition, but is not ideal for implementations of L. To prepare for a simulation of L on Turing machines we introduce a stack machine for L, utilising references to a heap instead of substitution, similar to the heap machine by Kunze et al. [15]. In contrast to the results there, we give a direct correctness proof instead of a step-wise refinement via several machines. Our machine is also similar to the heap machine by Forster et al. [7], but with fewer reduction rules simplifying verification.

Instead of terms, we will work with programs $P, Q : \mathsf{Pro} := \mathbb{L}\,\mathsf{Com}$, which are lists of commands. Commands are reference, application, abstraction, or return tokens:

$$c : \mathsf{Com} ::= \mathsf{ref}\ n \mid \mathsf{app} \mid \mathsf{lam} \mid \mathsf{ret}$$

The *compilation function* $\gamma : \mathsf{Ter} \to \mathsf{Pro}$ compiles terms to programs:

$$\gamma n \ := \ [\mathsf{ref}\ n] \qquad \gamma(st) \ := \ \gamma s +\!\!+ \gamma t +\!\!+ [\mathsf{app}] \qquad \gamma(\lambda s) \ := \ \mathsf{lam} :: \gamma s +\!\!+ [\mathsf{ret}]$$

We have $\gamma((\lambda xy.xx)(\lambda z.z)) = [\mathsf{lam}; \mathsf{lam}; \mathsf{ref}\ 1; \mathsf{ref}\ 1; \mathsf{app}; \mathsf{ret}; \mathsf{ret}; \mathsf{lam}; \mathsf{ref}\ 0; \mathsf{ret}; \mathsf{app}]$.

Compiled abstractions start with the token lam and end with ret. We can thus define a function $\phi P : \mathbb{O}(\mathsf{Pro} \times \mathsf{Pro})$ that extracts the body of an abstraction by matching the tokens like parentheses. We define $\phi P := \phi_{0,[]}\, P$, where $\phi_{k,Q}\, P$ is an auxiliary function storing the number k of unmatched lam and the processed prefix Q:

$$\phi_{k,Q}\,[] := \mathsf{None} \qquad \phi_{0,Q}\,(\mathsf{ret} :: P) := \mathsf{Some}\,(Q, P) \qquad \phi_{\mathsf{S}k,Q}\,(\mathsf{ret} :: P) := \phi_{k,Q +\!\!+ [\mathsf{ret}]}\, P$$

$$\phi_{k,Q}\,(\mathsf{lam} :: P) := \phi_{\mathsf{S}k,Q +\!\!+ [\mathsf{lam}]}\, P \qquad \phi_{k,Q}\,(c :: P) := \phi_{k,Q +\!\!+ [c]}\, P \ \text{ if } c = \mathsf{ref}\ n \text{ or } \mathsf{app}$$

The *states of the heap machine* are tuples T, V, H. The *control stack* T and the *value stack* V are lists of *closures* $g : \mathsf{Clos} := \mathsf{Pro} \times \mathbb{N}$. A closure (P, a) denotes an open program, where the reference 0 in P has to be looked up at address $a : \mathbb{N}$ in the heap when evaluating.

The *heap* H is a linked list of heap entries $e : \mathsf{Entry} := \mathbb{O}(\mathsf{Clos} \times \mathbb{N})$, i.e. an entry is either empty, or contains the head of the list and the address of its tail. Given a heap H and an address a, $H[a] : \mathbb{O}\,\mathsf{Entry}$ denotes the a-th element of H. We define $H[a, n]$ to be the n-th entry on the heap starting at address a as follows:

$$H[a, n] := \textbf{if } H[a] \textbf{ is } \mathsf{Some}\,(\mathsf{Some}\,(g, b)) \textbf{ then if } n \textbf{ is } \mathsf{S}n \textbf{ then } H[b, n] \textbf{ else } \mathsf{Some}\, g \textbf{ else } \mathsf{None}$$

We can now define the small-step semantics of the stack machine for L:

$$\begin{aligned} (\mathsf{lam} :: P, a) :: T, V, H &\rightsquigarrow (P', a) ::_{\mathsf{tc}} T, (Q, a) :: V, H && \textit{if } \phi P = \mathsf{Some}\,(Q, P') \\ (\mathsf{ref}\, n :: P, a) :: T, V, H &\rightsquigarrow (P, a) ::_{\mathsf{tc}} T, g :: V, H && \textit{if } H[a, n] = \mathsf{Some}\, g \\ (\mathsf{app} :: P, a) :: T, g :: (Q, b) :: V, H &\rightsquigarrow (Q, |H|) :: (P, a) ::_{\mathsf{tc}} T, V, H +\!\!+ [\mathsf{Some}\,(g, b)] \end{aligned}$$

Here, $(P, a) ::_{\mathsf{tc}} T := \textbf{if } P \textbf{ is } [] \textbf{ then } T \textbf{ else } (P, a) :: T$.

In the abstraction rule, the complete abstraction is parsed via ϕ and its body put on the value stack. In principle $::$ instead of $::_{\mathsf{tc}}$ could be used to obtain a correct machine, however the time complexity of this machine is easier to verify using this optimising operation.

Similarly, in the reference rule, the body of the abstraction corresponding to the variable n is looked up in the heap starting at address a and the result is put on the value stack.

In the application rule, the machine takes the closure of the called function (Q, b) and its argument g from the value stack. The address b is bound to g in the heap, the entry being appended to H, thus obtaining address $|H|$. The machine continues evaluating the body Q, where the value for reference 0 can be looked up at address $|H|$, where it was just placed.

Given a closed term s, the initial state of the machine is $([(\gamma s, 0)], [], [])$, i.e. an empty value stack, an empty heap, and the closure $(\gamma s, 0)$ on the task stack. In fact, for a closed term, 0 can be replaced by any address since there are no free references. An example run of the machine for $(\lambda xy.xx)(\lambda z.z)$ can be found in Figure 1, using a as start address.

To state the correctness of the stack machine we need to define an unfolding operation $\mathsf{unf}_H(P, a)$. We will use functional notation for unfolding on paper, but define it as a functional relation in Coq, since the function is not structurally recursive, and not even terminating on cyclic heaps. The function $\mathsf{unf} : \mathsf{Pro} \to \mathsf{tm}_{\mathsf{L}}$ unfolds programs from the value

$$\begin{aligned}
&[([\mathsf{lam}; \mathsf{lam}; \mathsf{ref}\ 1; \mathsf{ref}\ 1; \mathsf{app}; \mathsf{ret}\,; \mathsf{ret}\,; \mathsf{lam}; \mathsf{ref}\ 0; \mathsf{ret}\,; \mathsf{app}], a)], [], [] \\
\rightsquigarrow\ &[([\mathsf{lam}; \mathsf{ref}\ 0; \mathsf{ret}\,; \mathsf{app}], a)], [([\mathsf{lam}; \mathsf{ref}\ 1; \mathsf{ref}\ 1; \mathsf{app}; \mathsf{ret}\,], a)], [] \\
\rightsquigarrow\ &[([\mathsf{app}], a)], [([\mathsf{ref}\ 0], a), ([\mathsf{lam}; \mathsf{ref}\ 1; \mathsf{ref}\ 1; \mathsf{app}; \mathsf{ret}\,], a)], [] \\
\rightsquigarrow\ &[([\mathsf{lam}; \mathsf{ref}\ 1; \mathsf{ref}\ 1; \mathsf{app}; \mathsf{ret}\,], 0)], [], [\mathsf{Some}\,(([\mathsf{ref}\ 0], a), a)] \\
\rightsquigarrow\ &[], [([\mathsf{ref}\ 1; \mathsf{ref}\ 1; \mathsf{app}], 0)], [\mathsf{Some}\,(([\mathsf{ref}\ 0], a), a)]
\end{aligned}$$

$$\begin{aligned}
&\mathsf{unf}_{[\mathsf{Some}\,(([\mathsf{ref}\ 0],a),a)]}([\mathsf{ref}\ 1; \mathsf{ref}\ 1; \mathsf{app}], 0) \\
=\ &\mathsf{unf}_{[\mathsf{Some}\,(([\mathsf{ref}\ 0],a),a)],0,0}(\mathsf{unf}[\mathsf{ref}\ 1; \mathsf{ref}\ 1; \mathsf{app}]) \\
=\ &\mathsf{unf}_{[\mathsf{Some}\,(([\mathsf{ref}\ 0],a),a)],0,0}(\lambda 11) \\
=\ &\lambda\mathsf{unf}_{[\mathsf{Some}\,(([\mathsf{ref}\ 0],a),a)],0,1}(11) \\
=\ &\lambda(\mathsf{unf}_{[\mathsf{Some}(([\mathsf{ref}\ 0],a),a)],0,1}1)(\mathsf{unf}_{[\mathsf{Some}(([\mathsf{ref}\ 0],a),a)],0,1}1) \\
=\ &\lambda(\mathsf{unf}_{[\mathsf{Some}\,(([\mathsf{ref}\ 0],a),a)],0,0} \\
&\qquad(\mathsf{unf}[\mathsf{ref}\ 0]))\ (\mathsf{unf}_{[\mathsf{Some}\,(([\mathsf{ref}\ 0],a),a)],0,0}(\mathsf{unf}[\mathsf{ref}\ 0])) \\
=\ &\lambda(\lambda 0)(\lambda 0) = \lambda y.(\lambda z.z)(\lambda z.z)
\end{aligned}$$

Figure 1 Example of the execution for term $(\lambda xy.\ xx)\ (\lambda z.z)$.

stack into a term by inversing γ. It adds λ to the result, since only the bodies of abstractions are saved on the value stack. The function $\mathsf{unf}_{H,a,k} : \mathsf{tm}_\mathsf{L} \to \mathsf{tm}_\mathsf{L}$ substitutes free variables $n \geq k$ in a term by $H[a, n-k]$. Finally, $\mathsf{unf}_H : \mathsf{Pro} \times \mathbb{N} \to \mathsf{tm}_\mathsf{L}$ unfolds a result using the two previous functions. The example from above is continued in Figure 1.

$$\mathsf{unf}P := \lambda t \quad (\textit{if } \gamma t = P) \qquad\qquad \mathsf{unf}_{H,a,k}n := n \quad (\textit{if } n < k)$$

$$\mathsf{unf}_{H,a,k}n := \mathsf{unf}_{H,b,0}(\mathsf{unf}P) \quad (\textit{if } n \geq k \textit{ and } H[a, n-k] = \mathsf{Some}\,(P, b))$$

$$\mathsf{unf}_{H,a,k}(st) := (\mathsf{unf}_{H,a,k}s)(\mathsf{unf}_{H,a,k}s) \qquad \mathsf{unf}_{H,a,k}(\lambda s) := \lambda(\mathsf{unf}_{H,a,\mathsf{S}k}s) \qquad \mathsf{unf}_H(P, a) := \mathsf{unf}_{H,a,0}(\mathsf{unf}P)$$

We define the size of the components of a stack machine as follows:

$$\|\mathsf{ref}\ n\| := \|n\| + 1 \qquad \|\mathsf{app}\| := \|\mathsf{lam}\| := \|\mathsf{ret}\,\| := 1 \qquad \|n : \mathbb{N}\| := n + 1$$

$$\|(a, b)\| := \|a\| + \|b\| + 1 \qquad \|[]\| := 1 \qquad \|x :: l\| := \|x\| + \|l\|$$

The final correctness theorem then reads:

▶ **Theorem 6.** *Let s be closed.*

1. *If $s \rhd^i v$ then $([(\gamma s, 0)], [], []) \rightsquigarrow^{3\cdot i+1} ([], [(P, a)], H)$ for some P, a and H s.t. $\mathsf{unf}_H(P, a) = v$.*
2. *If $([(\gamma s, 0)], [], []) \rightsquigarrow^i (T, V, H)$ then $\|(T, V, H)\| \leq c \cdot (i + 1) \cdot (i + \|s\| + 1)$ for some c and if furthermore $\neg\exists\sigma.(T, V, H) \rightsquigarrow \sigma$, then $T = []$, $V = [(P, a)]$, and $s \rhd v$ for some P, a, and v s.t. $\mathsf{unf}_H(P, a) = v$ is defined.*

3.3 Mechanisation in Coq

The weak call-by-value λ-calculus L is a sweet spot for the mechanisation of computability and complexity theory – but only since it is engineered to be one. In principle, there is an abundance of options which λ-calculus to choose: call-by-value or call-by-name, weak or strong, variables are values or not, de Bruijn encoding with simple substitution, or with parallel substitution, or locally nameless, or parametric higher-order abstract syntax.

For the implementation of L on Turing machines it is mainly the choice of a simple substitution operation based on de Bruijn indices that is crucial. Parallel substitution is considerably more complicated to define, since it is not structurally recursive *and* a priori uses functions, i.e. uncountable types, to represent substitutions. In contrast, simple substitutions are structurally recursive and only require natural numbers for their definition.

To simulate Turing machines in L, it is crucial that also a small step semantics is available: the non-determinism of $\succ$, i.e. that it applies to both sides of an application, allows (directed) equational reasoning in correctness proofs without any overhead.

4 Turing machines

Turing machines [27] are widely used in books on computability theory, are the standard model of computation for complexity theory, and are still considered by many to be the model of computation most convincincly capturing all "effectively calculable" functions. Despite their universal use, there is however no consensus on how to formally define Turing machines in the literature. There are a multitude of definitions which can all be proved equivalent. In Appendix A, we compare the definition we use [9] to the one by Hopcroft et al. [13].[1]

In Section 4.2, we give an overview of the Turing machine verification framework from [9].

4.1 Definition

We start by defining a tape over type Σ inductively using four constructors [2]:

$$\mathsf{tp}_\Sigma ::= \mathsf{niltp} \mid \mathsf{leftof}\ r\ rs \mid \mathsf{midtp}\ ls\ m\ rs \mid \mathsf{rightof}\ l\ ls \qquad \textit{where } m, l, r : \Sigma \textit{ and } ls, rs : \mathbb{L}\Sigma$$

The representation does enforce that a tape contains a continous sequence of symbols, and that either a symbol or one of the ends of the tape is distinguished as head position. There are no explicit blank symbols, which allows for a unique representation of every tape and no well-formedness predicate is needed.

We define a type of moves Move, a function $\mathsf{mv} : \mathsf{Move} \to \mathsf{tp} \to \mathsf{tp}$ applying a move to a tape, a function $\mathsf{wr} : \mathbb{O}\Sigma \to \mathsf{tp} \to \mathsf{tp}$ writing to a tape, and a function $\mathsf{curr} : \mathsf{tp} \to \mathbb{O}\Sigma$ obtaining the current symbol of a tape in Figure 2.

We define multi-tape *Turing machines* $M : \mathsf{TM}^n_\Sigma$, where $n : \mathbb{N}$ is the number of tapes and the finite type Σ is the alphabet, as dependent pairs $(Q, \delta, q_0, \mathsf{halt})$ where Q is a finite type, $\delta : Q \times (\mathbb{O}\Sigma)^n \to Q \times (\mathbb{O}\Sigma \times \mathsf{Move})^n$, $q_0 : Q$ is the starting state, and $\mathsf{halt} : Q \to \mathbb{B}$ indicates halting states. The definition of Turing machine evaluation $M(q,t) \rhd (q', t')$ and the *Turing machine halting problem* $\mathsf{Halt}_{\mathsf{TM}}$ are defined in Figure 2.

A relation $R \subseteq (\mathbb{L}\mathbb{B})^k \times (\mathbb{L}\mathbb{B})$ is TM-*computable* with time-complexity $\tau \subseteq \mathbb{N}^{1+k} \times \mathbb{N}$ if

$$\begin{aligned}&\exists n : \mathbb{N}.\ \exists \Sigma.\ \exists \mathsf{s}\,\mathsf{bl} : \Sigma.\ \mathsf{s} \neq \mathsf{bl} \land \exists M : \mathsf{TM}^{1+k+n}_\Sigma . \forall l_1 \dots l_k .\\ &\quad (\forall l. R\,(l_1, \dots, l_k)\, l \to \exists i \le \tau(|l|, |l_1|, \dots, |l_k|).\ \exists q\, \boldsymbol{t}.\ M(q_0, [\mathsf{niltp}, \overline{l_1}, \dots, \overline{l_k}, \mathsf{niltp}, \dots, \mathsf{niltp}]) \rhd^i (q, \boldsymbol{t}) \land \boldsymbol{t}[0] = \overline{l}) \land \\ &\quad \forall q\, \boldsymbol{t}\, i.\ M(q_0, [\mathsf{niltp}, \overline{n_1}, \dots, \overline{n_k}, \mathsf{niltp}, \dots, \mathsf{niltp}]) \rhd^i (q, \boldsymbol{t}) \to \exists l.\ R\,(l_1, \dots, l_k)\, l\end{aligned}$$

with $\overline{[x_1, \dots, x_n]} := \mathsf{midtp}\ []\ \mathsf{bl}\ [\overline{x_1}, \dots, \overline{x_n}]$ and $\overline{\mathsf{true}} := \mathsf{s}$, $\overline{\mathsf{false}} := \mathsf{bl}$.

4.2 Verified programming of Turing machines

As presented, Turing machines are not compositional: There is no canonical way how to execute a 5-tape machine over alphabet $\mathbb{B}$ after a 3-tape machine over $\mathbb{O}(\mathbb{B} \times \mathbb{B})$.

To allow for the composition of Turing machines and their verification, we first introduce labellings in order to abstract away from the state space. A *labelled Turing machine* over a type L, written $M : \mathsf{TM}^n_\Sigma(L)$, is a dependent pair (M', lab_M) of a machine $M' : \mathsf{TM}^n_\Sigma$ and a labelling function $\mathsf{lab}_M : Q_{M'} \to L$.

To prove the soundness of machines, we introduce *realisation*. A Turing machine M : $\mathsf{TM}^n_\Sigma(L)$ realises a relation $R \subseteq \mathsf{tp}^n_\Sigma \times (L \times \mathsf{tp}^n_\Sigma)$ if

$$M \vDash R := \forall \boldsymbol{t}\ q\ \boldsymbol{t'}.\ M(q_0, \boldsymbol{t}) \rhd (q, \boldsymbol{t'}) \to R\ \boldsymbol{t}\ (\mathsf{lab}_M\ q, \boldsymbol{t'})$$

[1] chosen for instance by Wikipedia as reference definition

$$\mathsf{Move} ::= \mathsf{L} \mid \mathsf{N} \mid \mathsf{R}$$

$$\mathsf{mv} : \mathsf{Move} \to \mathsf{tp} \to \mathsf{tp}$$

$$\mathsf{mv}\,\mathsf{L}\,(\mathsf{rightof}\; l\; ls) := \mathsf{midtp}\; ls\; l\; [] \qquad \mathsf{mv}\,\mathsf{R}\,(\mathsf{leftof}\; r\; rs) := \mathsf{midtp}\; []\; r\; rs$$

$$\mathsf{mv}\,\mathsf{L}\,(\mathsf{midtp}\; []\; m\; rs) := \mathsf{leftof}\; m\;\; rs \qquad \mathsf{mv}\,\mathsf{R}\,(\mathsf{midtp}\; ls\; a\; []) := \mathsf{rightof}\; a\;\; ls$$

$$\mathsf{mv}\,\mathsf{L}\,(\mathsf{midtp}\; (l :: ls)\; a\; rs) := \mathsf{midtp}\; ls\; l\; (a :: rs) \qquad \mathsf{mv}\,\mathsf{R}\,(\mathsf{midtp}\; ls\; a\; (r :: rs)) := \mathsf{midtp}\; (a :: ls)\; r\; rs$$

$$\mathsf{mv}\; m\; t := t \qquad \textit{in all other cases}$$

$$\mathsf{wr} : \mathbb{O}\Sigma \to \mathsf{tp} \to \mathsf{tp}$$

$$\mathsf{wr}\;\mathsf{None}\; t := t \qquad \mathsf{wr}\;(\mathsf{Some}\; a)\;\mathsf{niltp} := \mathsf{midtp}\; []\; a\; [] \qquad \mathsf{wr}\;(\mathsf{Some}\; a)\;(\mathsf{midtp}\; ls\; b\; rs) := \mathsf{midtp}\; ls\; a\; rs$$

$$\mathsf{wr}\;(\mathsf{Some}\; a)\;(\mathsf{leftof}\; r\; rs) := \mathsf{midtp}\; []\; a\; (r :: rs) \qquad \mathsf{wr}\;(\mathsf{Some}\; a)\;(\mathsf{rightof}\; l\; ls) := \mathsf{midtp}\; (l :: ls)\; a\; []$$

$$\mathsf{curr} : \mathsf{tp} \to \mathbb{O}\Sigma$$

$$\mathsf{curr}(\mathsf{midtp}\; ls\; a\; rs) := \mathsf{Some}\; a \qquad \mathsf{curr} t := \mathsf{None} \quad \textit{otherwise}$$

$$\frac{\mathsf{halt}\; q = \mathsf{true}}{M(q, \boldsymbol{t}) \rhd^{i} (q, \boldsymbol{t})} \qquad \frac{\mathsf{halt}\; q = \mathsf{false} \qquad \delta(q, \mathsf{map}\,\mathsf{curr}\,\boldsymbol{t}) = (q', a) \qquad M(q', \mathsf{map}_2(\lambda(c, m)\, t.\mathsf{mv}\; m\; (\mathsf{wr}\; c\; t))\; a\; t) \rhd^{i} (q'', \boldsymbol{t}')}{M(q, \boldsymbol{t}) \rhd^{\mathsf{S} i} (q'', \boldsymbol{t}')}$$

$$\mathsf{Halt}_{\mathsf{TM}^n_\Sigma}(M : \mathsf{TM}^n_\Sigma, \boldsymbol{t} : \mathsf{tp}^n_\Sigma) := \exists i q'\, \boldsymbol{t}'.\; M(q_0, \boldsymbol{t}) \rhd^{i} (q', \boldsymbol{t}')$$

$$\mathsf{Halt}_{\mathsf{TM}}(n : \mathbb{N}, \Sigma, M : \mathsf{TM}^n_\Sigma, \boldsymbol{t} : \mathsf{tp}^n_\Sigma) := \mathsf{Halt}_{\mathsf{TM}^n_\Sigma}(M, \boldsymbol{t})$$

Figure 2 Definitions for Turing machines.

Dually, we introduce *termination*. $M : \mathsf{TM}^n_\Sigma(L)$ terminates in $T \subseteq \mathsf{tp}^n_\Sigma \times \mathbb{N}$ if

$M \downarrow T := \forall \boldsymbol{t}\, i.\; T\; \boldsymbol{t}\; i \to \exists q\, \boldsymbol{t}'.\; M(t) \rhd^{i} (q, \boldsymbol{t}').$

We call a machine *total* if $\exists c.\; M \downarrow \lambda t i. i \geq c$, i.e. if it halts on any tape in at most c steps.

▶ **Fact 7.** *The introduced predicates are (anti-)monotonic:*

1. *If $M \vDash R'$ and $\forall \boldsymbol{t}\, \ell\, \boldsymbol{t}'.\; R'\; \boldsymbol{t}\; (\ell, \boldsymbol{t}') \to R\; \boldsymbol{t}\; (\ell, \boldsymbol{t}')$, then $M \vDash R$.*
2. *If $M \downarrow T'$ and $\forall \boldsymbol{t} i.\; T\, \boldsymbol{t}\, i \to T'\, \boldsymbol{t}\, i$, then $M \downarrow T$.*

We will use the following total machines we call *primitive machines*:

$$\mathsf{Read} : \mathsf{TM}^1_\Sigma(\mathbb{O}(\Sigma)) \vDash \lambda\; \boldsymbol{t}\; (\ell, \boldsymbol{t}').\; \ell = \mathsf{curr}\; \boldsymbol{t}[0] \wedge \boldsymbol{t}' = \boldsymbol{t} \qquad \mathsf{Write}\; s : \mathsf{TM}^1_\Sigma(\mathbb{1}) \vDash \lambda\; \boldsymbol{t}\; \boldsymbol{t}'.\; \boldsymbol{t}'[0] = \mathsf{wr}\; s\; \boldsymbol{t}[0]$$

$$\mathsf{Move}\; d : \mathsf{TM}^1_\Sigma(\mathbb{1}) \vDash \lambda\; \boldsymbol{t}\; \boldsymbol{t}'.\; \boldsymbol{t}'[0] = \mathsf{mv}\; d\; \boldsymbol{t}[0] \qquad \mathsf{Return}\, \ell : \mathsf{TM}^n_\Sigma(L) \vDash \lambda\; \boldsymbol{t}\; (\ell', \boldsymbol{t}').\; \boldsymbol{t}' = \boldsymbol{t} \wedge \ell' = \ell$$

The last necessary tool now are *combinators* to compose machines. Given $M : \mathsf{TM}^n_\Sigma(L)$ and $M'_\ell : \mathsf{TM}^n_\Sigma(L')$ (for $\ell : L$), we introduce the combinator $\mathsf{Switch}\; M\; M' : \mathsf{TM}^n_\Sigma(L')$, which executes M and, depending on the label ℓ returned by M, executes M'_ℓ.

$$\frac{M \vDash R \qquad \forall (\ell : L).\; M'_\ell \vDash R'_\ell}{\mathsf{Switch}\; M\; M' \vDash \lambda \boldsymbol{t}_0\; (\ell', \boldsymbol{t}').\; \exists \boldsymbol{t}\; (\ell : L).\; R\; \boldsymbol{t}_0\; (\ell, \boldsymbol{t}) \wedge R'_\ell\; \boldsymbol{t}\; (\ell', \boldsymbol{t}')} \qquad \frac{M \downarrow T \quad M \vDash R \quad \forall (\ell : L). M'_\ell \downarrow T'_\ell}{\mathsf{Switch}\; M\; M' \downarrow \lambda\; \boldsymbol{t}\; k.\; \exists\; k_1\; k_2.\; T\; \boldsymbol{t}\; k_1 \wedge 1 + k_1 + k_2 \leq k \wedge \forall\; \ell\; \boldsymbol{t}'.\; R\; \boldsymbol{t}\; (\ell, \boldsymbol{t}') \to T'_\ell\; \boldsymbol{t}'\; k_2}$$

Given a machine $M : \mathsf{TM}^n_\Sigma(\mathbb{O}(L))$, we introduce the combinator $\mathsf{While}\; M : \mathsf{TM}^n_\Sigma(L)$, which loops M until the result label is $\mathsf{Some}\, \ell$. The realisation relation for While is defined inductively, whereas the termination relation is the co-inductively defined accessibility relation (the dashed line indicates coinduction).

$$\frac{M \vDash R}{\mathsf{While}\ M \vDash \mathsf{While}R\ R} \qquad \frac{R\ \boldsymbol{t}\ (\mathsf{Some}\,\ell, \boldsymbol{t'})}{\mathsf{While}R\ R\ \boldsymbol{t}\ (\ell, \boldsymbol{t'})} \qquad \frac{R\ \boldsymbol{t}\ (\mathsf{None}, \boldsymbol{t'}) \qquad \mathsf{While}R\ R\ \boldsymbol{t'}\ (\ell, \boldsymbol{t''})}{\mathsf{While}R\ R\ \boldsymbol{t}\ (\ell, \boldsymbol{t''})}$$

$$\frac{M \downarrow T \qquad M \vDash R}{\mathsf{While}\ M \downarrow \mathsf{While}T\ R\ T} \qquad \begin{array}{c} T\ \boldsymbol{t}\ k_1 \quad \forall \boldsymbol{t'}\ l.\ R\ \boldsymbol{t}\ (\mathsf{Some}\,l, \boldsymbol{t'}) \rightarrow k_1 \leq k \\ \forall \boldsymbol{t'}.\ R\ \boldsymbol{t}\ (\mathsf{None}, \boldsymbol{t'}) \rightarrow \exists k_2.\ \mathsf{While}T\ R\ T\ \boldsymbol{t'}\ k_2 \wedge 1 + k_1 + k_2 \leq k \\ \hdashline \mathsf{While}T\ R\ T\ \boldsymbol{t}\ k \end{array}$$

Composing machines with the operator Switch only works for machines over the same alphabet and the same number of tapes. To remedy this situation we introduce lifting operations for alphabets and tapes, and also a relabelling operation Relabel.

Given a retraction $f : \Sigma \rightarrow \Gamma$, a default symbol $d : \Sigma$, and a machine $M : \mathsf{TM}_\Sigma^n$, the *alphabet lift* $\Uparrow_{(f,d)} M : \mathsf{TM}_\Gamma^n$ translates every read symbol via f^{-1}, passes it to M, and translates the symbol M writes via f. In case f^{-1} returns None, d is passed to M.

Given a retraction $I : \mathsf{Fin}_m \rightarrow \mathsf{Fin}_n$ and a machine $M : \mathsf{TM}_\Sigma^m$, the *tape lift* $\Uparrow_I M : \mathsf{TM}_\Sigma^n$ replicates the behavior of M on tape i on tape Ii, and leaves all other tapes untouched.

We only show the canonical realisation and termination relations for the tape lift here:

$$\Uparrow_I R := \lambda t\ (\ell, t').\ R\ (\mathsf{select}\ I\ \boldsymbol{t})\ (\ell, \mathsf{select}\ I\ \boldsymbol{t'}) \wedge \forall (i : \mathsf{Fin}_n).\ i \notin I \rightarrow \boldsymbol{t'}[i] = t[i]$$

$$\Uparrow_I T := \lambda t\ k.\ T\ (\mathsf{select}\ I\ \boldsymbol{t})\ k \qquad \textit{where}\ \mathsf{select}\ I\ \boldsymbol{t} := \mathsf{map}\ (\lambda i.\ t[I(i)])\ [0, \ldots, m-1]$$

$$\frac{M : \mathsf{TM}_\Sigma^m(L) \vDash R \qquad I : \mathsf{Fin}_m \hookrightarrow \mathsf{Fin}_n}{\Uparrow_I M : \mathsf{TM}_\Sigma^n \vDash\ \Uparrow_I R} \qquad \frac{M \downarrow T \qquad I : \mathsf{Fin}_m \hookrightarrow \mathsf{Fin}_n}{\Uparrow_I M \downarrow\ \Uparrow_I T}$$

Given a function $r : L_1 \rightarrow L_2$ and a machine $M : \mathsf{TM}_\Sigma^n(L_1)$, Relabel $M\ r : \mathsf{TM}_\Sigma^n(L_2)$ behaves like M, but returns label $r\ell$ where M returned ℓ.

The last important layer of abstraction introduces the treatment of tapes as registers, based on the notion $t[i] \simeq v$ expressing that tape $t[i]$ contains an encoded value $v : V$. A type V is a TM-*encodable type* on alphabet Σ if there is a (designated) injective function $\varepsilon : V \rightarrow \mathbb{L}\Sigma$. We define such designated encoding functions for several data types, e.g. booleans, tuples, and lists of encodable types. In Coq, the implementation of encodable types relies on type classes, such that users can define their own encoding functions.

We then define *tape containment* $t \simeq_f v$, where V is TM-encodable on alphabet Σ, $v : V$, $t : \mathsf{tp}_{\Gamma^+}$, and $f : \Sigma \rightarrow \Gamma$.

$$\Gamma^+ ::= \mathsf{START} \mid \mathsf{STOP} \mid \mathsf{UNKNOWN} \mid (s : \Gamma)$$

$$t \simeq_f v := \exists ls. t = \mathsf{midtp}\ ls\ \mathsf{START}\ (\mathsf{map}\ f\ (\varepsilon v) +\!\!+ [\mathsf{STOP}])$$

Note that the position of the head is fixed in the definition of tape containment: The head must be located on the start symbol. By extending the alphabet Σ with the delimiting symbols START and END, values can effectively be copied from one tape to another tape. The symbol UNKNOWN is used as the canonical default symbol for the alphabet lift. Let $M : \mathsf{TM}_{\Sigma^+}^n$ and $f : \Sigma \rightarrow \Gamma$. Then $\Uparrow_{f^+} M : \mathsf{TM}_{\Gamma^+}^n$ (with the canonically inferrable injection $f^+ : \Sigma^+ \rightarrow \Gamma^+$) is an alphabet lift of M. In case the lifted machine reads a symbol that is not in the image of f, $\Uparrow_{f^+} M$ behaves like if M reads UNKNOWN. However, this will by design not happen if the head is under a symbol of the encoding of a value.

Void tapes (written $\mathsf{isVoid}\, t$) do not contain values. The head of the tape is located at the right-most symbol:

$$\mathsf{isVoid}\, t := \exists m\ ls.\ t = \mathsf{midtp}\ ls\ m\ []$$

A void tape can be initialised with a value by writing the encoding of a value delimited by START and END.

5 Simulating L on Turing machines

To simulate L on Turing machines, we use the stack machine semantics, meaning we have to implement the relation $\rightsquigarrow$ from Section 3.2 as multi-tape Turing machine $\mathsf{Step} : \mathsf{TM}^n_\Sigma$. The central components of Step are machines implementing the heap lookup operation $H[a, n]$ and the parsing operation ϕ. We will omit concrete implementations, but show the correctness and termination relations and briefly discuss the proof goals for such verifications. The machines will share an alphabet Σ consisting of 30 symbols, allowing to encode commands, programs, addresses, closures, task and value stack, heap entries, and heap.

The relations we display here are simplified in comparison to the actual relations in Coq w.r.t. two aspects: First, we omit the retractions $f_X : \Sigma_X \to \Sigma$ when writing $\simeq$. Since as long as concrete f_X are fixed for every type X encodable on type Σ_X, their concrete definitions do not matter. Secondly, we omit the condition $\mathsf{isVoid}\, t$ both in the premise and conclusion of rules: Any unspecified tape is always implicitly void.

▶ **Fact 8.** *There is a machine* $\mathsf{Lookup} : \mathsf{TM}^5_\Sigma(\mathbb{B})$ *and a* $c : \mathbb{N}$ *s.t.*

1. $\mathsf{Lookup} \vDash \lambda \boldsymbol{t}(\ell, \boldsymbol{t'}).\ \forall H\, a\, n.\ \boldsymbol{t}[0] \simeq H \to \boldsymbol{t}[1] \simeq a \to \boldsymbol{t}[2] \simeq n \to$
   $\textbf{if } \ell \textbf{ then } \exists g.\ H[a, b] = \mathsf{Some}\, g \wedge \boldsymbol{t'}[0] \simeq H \wedge \boldsymbol{t'}[3] \simeq g \textbf{ else } H[a, b] = \mathsf{None}$
2. $\mathsf{Lookup} \downarrow \lambda \boldsymbol{t} i.\ \exists H\, a\, n.\ \boldsymbol{t}[0] \simeq H \wedge \boldsymbol{t}[1] \simeq a \wedge \boldsymbol{t}[2] \simeq n \wedge i \geq c \cdot (n+1) \cdot (\|H\| + \max \|H\| \|a\|)$

Proof. The machine Lookup can be defined by using building blocks like While and Switch. Once the machine is defined, an inductive relation R s.t. $\mathsf{Lookup} \vDash R$ can be automatically inferred from the relations of the building blocks.

By Fact 7 it then suffices to prove $R\, t\, (\ell, t') \to \forall H\, a\, n.\ \boldsymbol{t}[0] \simeq H \to \boldsymbol{t}[1] \simeq a \to \boldsymbol{t}[2] \simeq n \to$ $\textbf{if } \ell \textbf{ then } \exists g.\ H[a, b] = \mathsf{Some}\, g \wedge \boldsymbol{t'}[0] \simeq H \wedge \boldsymbol{t'}[3] \simeq g \textbf{ else } H[a, b] = \mathsf{None}$ by induction on R.

The termination proof is dual. ◀

▶ **Fact 9.** *There is a machine* $\mathsf{Parse} : \mathsf{TM}^5_\Sigma(\mathbb{B})$ *and a* $c : \mathbb{N}$ *s.t.*

1. $\mathsf{Parse} \vDash \lambda \boldsymbol{t}\, (\ell, t').\forall P.\boldsymbol{t}[0] \simeq P \to$
   $\textbf{if } \ell \textbf{ then } \exists Q\, P'.\ \phi P = (Q, P') \wedge \boldsymbol{t'}[0] \simeq P' \wedge \boldsymbol{t'}[1] \simeq Q \textbf{ else } \phi P = \mathsf{None}$
2. $\mathsf{Parse} \downarrow \lambda \boldsymbol{t} i.\ \exists P.\ \boldsymbol{t}[0] \simeq P \wedge i \geq \|c\| \cdot P^2$

▶ **Fact 10.** *There is a machine* $\mathsf{Step} : \mathsf{TM}^{11}_\Sigma(\mathbb{B})$ *and a* $c : \mathbb{N}$ *s.t.*

1. $\mathsf{Step} \vDash \lambda \boldsymbol{t}\, (\ell, t').\ \forall T\, V\, H.\ \boldsymbol{t}[0] \simeq T \to \boldsymbol{t}[1] \simeq V \to \boldsymbol{t}[2] \simeq H \to$
   $\textbf{if } \ell \textbf{ then } \exists T'\, V'\, H'.\ (T, V, H) \rightsquigarrow (T', V', H') \wedge \boldsymbol{t'}[0] \simeq T' \wedge \boldsymbol{t'}[1] \simeq V' \wedge \boldsymbol{t'}[2] \simeq H'$
   $\textbf{else } (\neg \exists \sigma.\ (T, V, H) \rightsquigarrow \sigma) \wedge T = [] \to \boldsymbol{t'}[0] \simeq [] \wedge \boldsymbol{t'}[1] \simeq V \wedge \boldsymbol{t'}[2] \simeq H$
2. $\mathsf{Step} \vDash \lambda \boldsymbol{t}\, i.\ \boldsymbol{t}[0] \simeq T \wedge \boldsymbol{t}[1] \simeq V \wedge \boldsymbol{t}[2] \simeq H \wedge i \geq 1 + c \cdot \textbf{if } T \textbf{ is } (a, P) :: \_ \textbf{ then}$
   $\|a\| + \|H\| + \|V\| + \|P\| \cdot (1 + \|H\| + \max a \|H\|_{\mathsf{addr}} + \|P\|) \textbf{ else } 0$

This suffices to prove one direction of the time invariance thesis w.r.t. simulation:

▶ **Theorem 11.** *There is* $M_{\mathsf{sim}} : \mathsf{TM}^{11}_\Sigma$ *and a polynomial* $p : \mathbb{N} \to \mathbb{N}$ *s.t. for closed terms* s

1. *If* $s \rhd^i v$, *then there exist* $\boldsymbol{t}$, H, P, *and* a *s.t.* $M_{\mathsf{sim}}(\overline{[(\gamma s, 0)]}, \mathsf{niltp}, \dots, \mathsf{niltp}) \rhd^{p(i, \|s\|)} \boldsymbol{t}$ *with* $\boldsymbol{t}[1] = \overline{[]}$, $\boldsymbol{t}[2] = \overline{(P, a)}$, $\boldsymbol{t}[3] = \overline{H}$, *and* $\mathsf{unf}_H(P, a) = v$.
2. *If* $M_{\mathsf{sim}}(\overline{[(\gamma s, 0)]}, \mathsf{niltp}, \dots, \mathsf{niltp})$ *terminates, so does* s.

Proof. Define $M_{\mathsf{sim}} := \mathsf{While\ Step}$ and $p\, i\, m := c \cdot (3i+2)^2 \cdot (3i + 1 + m) \cdot m$ for some c. ◀

▶ **Corollary 12.** $\mathsf{Halt}_{\mathsf{L}} \preceq \mathsf{Halt}_{\mathsf{TM}}$

A first version of the Coq verification of M_{sim} was also discussed in Wuttke's bachelor's thesis [28].

6 Simulating Turing machines in L

To simulate Turing machines in L, we first give an alternative, executable semantics for Turing machines based on iteration of a step-function. We then recap the central ingredients of the certifying extraction framework [6], which we use to extract the step-function to L. And lastly, we implement a (potentially non-terminating) iteration combinator in L.

We define a function $\mathsf{nxt}_M : Q_M \times \mathsf{tp}^n_\Sigma \to Q_M \times \mathsf{tp}^n_\Sigma + \mathsf{tp}^n_\Sigma$ and a polymorphic function $\mathsf{loop} : (X \to X + Y) \to X \to \mathbb{N} \to \mathbb{O}Y$ as follows:

$$\mathsf{nxt}_M(q, \boldsymbol{t}) := \textbf{if } \mathsf{halt}\, q \textbf{ then } \boldsymbol{t} \textbf{ else let } (q', \boldsymbol{a}) := \delta_M(q, \mathsf{curr}\, \boldsymbol{t}) \textbf{ in } (q', \mathsf{map}_2(\lambda(c,m)t.\mathsf{mv}\, m\, (\mathsf{wr}\, c\, t))\, \boldsymbol{a}\, \boldsymbol{t})$$

$$\mathsf{loop}\, f\, x\, 0 := \mathsf{None} \qquad \mathsf{loop}\, f\, x\, (\mathsf{S}n) := \mathsf{loop}\, f\, x'\, n \ \ (\text{if } fx = \mathsf{inl}\, x')$$

$$\mathsf{loop}\, f\, x\, (\mathsf{S}n) := \mathsf{Some}\, y \ \ (\text{if } fx = \mathsf{inr}\, y)$$

▶ **Fact 13.** $\mathsf{loop}\, \mathsf{nxt}_M\, (q_0, \boldsymbol{t})\, (\mathsf{S}i) = \mathsf{Some}\, \boldsymbol{t'} \leftrightarrow \exists q'.\ M(q, \boldsymbol{t}) \rhd^i (q', \boldsymbol{t'})$

The certifying extraction framework [6] automatically extracts L-terms s_f for functions f and proves that s_f computes f. Additionally, one can pass a time complexity function τ_f and is then presented with proving certain recurrence equations for τ_f. Since we do not implement higher-order functions, we can give a simplified account of the framework here.

Central in the framework are Scott encodings, which are used to encode elements of arbitrary, first-order types as L-terms. The idea behind Scott encodings is that case analysis is by application: For instance, **if** b **then** a_1 **else** a_2 corresponds to the L-term $\overline{b}\ s_{a_1}\ s_{a_2}$, where s_{a_1} and s_{a_2} compute a_1 and a_2 respectively.

In general, for types $A_1, \ldots, A_n, B$ with a Scott encoding, a function $f : A_1 \to \ldots A_n \to B$, is computed by a term s_f with time complexity function $\tau_f : A_1 \to \cdots \to A_n \to \mathbb{N}$ if

$$\forall a_1, \ldots, a_n.\ \exists k \le \tau_f\ a_1 \ldots a_n.\ s_f\ \overline{a_1} \ldots \overline{a_n} \rhd^k \overline{f\, a_1 \ldots a_n}$$

The framework also supports higher-order functions and currying, but we omit those features complicating e.g. the definition of time complexity since we do not rely on them.

The certifying extraction framework comes with a library of computability proofs including time complexity, covering natural numbers, list, and vectors. Furthermore, one can give a general computability proof for functions with listable domain. I.e. if there is $[x_1, \ldots, x_n] : \mathbb{L}X$ s.t. $\forall x : X. x \in [x_1, \ldots, x_n]$ and $f : X \to Y$, then there is s_f and a constant c s.t. $\forall x. \exists i \le c \cdot n. s_f\ \overline{x} \rhd^i \overline{fx}$.

Since δ_M has a listable domain, we can use the certifying extraction framework to extract nxt for every M:

▶ **Fact 14.** *Let* $M : \mathsf{TM}^n_\Sigma$. *There is* $\mathsf{nxt}_M : \mathsf{tm}_\mathsf{L}$ *and* $C_\mathsf{nxt} : \mathbb{N}$ *s.t.* $s_{\mathsf{nxt}_M} \overline{(q, \boldsymbol{t})} \rhd^{C_\mathsf{nxt}} \overline{\mathsf{nxt}(q, \boldsymbol{t})}$.

Proof. The framework generates the proof obligations $52 \cdot |Q_M|^2 + 56 \le C_\mathsf{nxt}$, and $52 \cdot |Q_M|^2 + 130 \cdot n + 216 \le C_\mathsf{nxt}$, where $|Q_M|$ is the number of states of M. If C is picked large enough before running extraction, the obligations can be discharged automatically by the tactic `solverec` provided by the framework. ◀

We now define a term s_loop which expects f and x and loops f until a value y is found, or indefinitely if not. Since the extraction framework only covers total functions, we have to manually implement s_loop. To do so, we rely on a recursion combinator ρ [12], also employed by the framework to use recursion:

▶ **Fact 15.** *If s and s' are closed abstractions, $\rho s s' \succ^3 s(\rho s)s'$.*

We then define s_{loop} with time complexity function $\tau_{\mathsf{loop}} : (X \to \mathbb{N}) \to X \to \mathbb{N} \to \mathbb{N}$:

$$s_{\mathsf{loop}} := \rho(\lambda r f x.\, f x(\lambda x' z. r f x')(\lambda y z. y)(\lambda z.\, z))$$

$$\tau_{\mathsf{loop}}\, \tau_f\, x\, 0 := 0 \qquad \tau_{\mathsf{loop}}\, \tau_f\, x\, (\mathsf{S} i) := \tau_f x + 11 + \textbf{if } f x \textbf{ is } \mathsf{inl}\, x' \textbf{ then } \tau_{\mathsf{loop}}\, \tau_f\, x\, i \textbf{ else } 0$$

▶ **Lemma 16.** *Let f be computable by s_f with time complexity function τ_f, i.e. $\forall x. \exists i \leq \tau_f x.\ s_f \overline{x} \triangleright^i \overline{f x}$. Then*
1. *If $\mathsf{loop}\, f\, x\, i = \mathsf{Some}\, y$, then $\exists k \leq \tau_{\mathsf{loop}}\, \tau_f\, x\, i.\ s_{\mathsf{loop}}\, s_f\, \overline{x}\, \overline{i} \triangleright^k \overline{y}$.*
2. *If $s_{\mathsf{loop}}\, s_f\, \overline{x}$ terminates, there exist i and y s.t. $\mathsf{loop}\, f\, x\, i = \mathsf{Some}\, y$.*

This suffices to prove one direction of the time invariance thesis w.r.t. simulation:

▶ **Theorem 17.** *There is $s_{\mathsf{sim}} : \mathsf{tm_L}$ s.t. for all $M : \mathsf{TM}^n_\Sigma$ there is $C : \mathbb{N}$ s.t. for all $\boldsymbol{t} : \mathsf{tp}^n_\Sigma$*
1. *If $M(q_0, \boldsymbol{t}) \triangleright^i (q, \boldsymbol{t'})$, then $\exists j \leq C \cdot i + C.\ s_{\mathsf{sim}}\ s_{\mathsf{nxt}_M}\ \overline{\boldsymbol{t}} \triangleright^j \overline{\boldsymbol{t'}}$.*
2. *If $s_{\mathsf{sim}}\ s_{\mathsf{nxt}_M}\ \overline{\boldsymbol{t}} \triangleright v$, then $\exists q \boldsymbol{t'}.\ M(q_0, \boldsymbol{t}) \triangleright (q, \boldsymbol{t'})$.*

7 TM-computable relations over $\mathbb{LB}$ are L-computable

Let $R \subseteq (\mathbb{LB})^k \times \mathbb{LB}$ be computable by $M : \mathsf{TM}^n_\Sigma$. We define an L-term computing R by taking $l_1, \ldots, l_k$ as input, converting them to their respective TM-encoding, and then running M with the help of s_{sim}. Step-by-step, s has to:
1. Expect input in the form $s\, \overline{l_1} \ldots \overline{l_k}$,
2. for every $1 \leq i \leq k$ compute $\overline{\mathsf{midtp}\, []\, \mathsf{bl}\, l_i}$, i.e. the L-encoding of the TM-encoding of l_i.
3. run the simulation $s_{\mathsf{sim}}\ s_{\mathsf{nxt}_M}\ \overline{[\mathsf{niltp}, t_1, \ldots, t_k, \mathsf{niltp}, \ldots, \mathsf{niltp}]}$.
4. this computation will (if it terminates) terminate with a value $\overline{(\mathsf{midtp}\, []\, b\, l, t'_2 \ldots, t'_n)}$,
5. meaning s has to output $\overline{l}$.

Three challenges arise: the term s has to be defined parametric in k, the L-encoding of the lists $l_1, \ldots, l_k$ has to be converted to the L-encoding $\overline{[\mathsf{niltp}, t_1, \ldots, t_k, \mathsf{niltp}, \ldots, \mathsf{niltp}]}$, and the L encoding of a result $\boldsymbol{t'}$ has to be analysed, and the TM-encoding of a list l contained $\boldsymbol{t'}[0]$ has to be converted to the L-encoding of l.

For the first task, we implement k-ary substitutions and combinators.

▶ **Fact 18.** *One can define functions $s^n_{\boldsymbol{u}} : \mathsf{tm}_L$ where $s : \mathsf{tm_L}, n : \mathbb{N}, \boldsymbol{u} : \mathsf{tm}^k_\mathsf{L}$ and*

$$\lambda_k : \mathsf{tm_L} \to \mathsf{tm_L} \qquad\qquad \mathsf{app}_k : \mathsf{tm_L} \to \mathsf{tm}^k_\mathsf{L} \to \mathsf{tm_L} \qquad\qquad \mathsf{vars}_k : \mathsf{tm}^k_L$$

such that the following hold:
1. $\mathsf{vars}_{\mathsf{S}k} = k :: \mathsf{vars}_k$,
2. $(\mathsf{app}_k s(s_1, \ldots, s_k))^n_{\boldsymbol{u}} = \mathsf{app}_k(s^n_{\boldsymbol{u}})((t_1)^n_{\boldsymbol{u}}, \ldots, (t_k)^n_{\boldsymbol{u}})$,
3. *if all elements of $\boldsymbol{u}$ are closed abstractions, then $\mathsf{app}_k(\lambda_k s)\boldsymbol{u} \succ^k s^0_{\boldsymbol{u}}$.*

The second and third tasks can again be done by extraction.

▶ **Fact 19.** *There is a closed abstraction s_{prep} s.t. $s_{\mathsf{prep}}\, \overline{(l_1, \ldots, l_k)} \triangleright \overline{[\mathsf{niltp}, t_1, \ldots, t_k, \mathsf{niltp}, \ldots, \mathsf{niltp}]}$, where $t_i := \mathsf{midtp}\, []\, \mathsf{bl}\, [\overline{x_1}, \ldots, \overline{x_n}]$ for $l_i = [x_1, \ldots, x_n]$.*

▶ **Fact 20.** *There is a closed abstraction $s_{\mathsf{unenc_{TM}}}$ s.t. if $\boldsymbol{t}[0] = \mathsf{midtp}\, []\, \mathsf{bl}\, [\overline{x_1}, \ldots, \overline{x_m}]$ and $l = [x_1, \ldots, x_n]$ we have $s_{\mathsf{unenc_{TM}}} \overline{\boldsymbol{t}} \triangleright \overline{\mathsf{Some}\, l}$ and $s_{\mathsf{unenc_{TM}}} \overline{\boldsymbol{t}} \triangleright \overline{\mathsf{None}}$ otherwise.*

▶ **Theorem 21.** *If $R \subseteq (\mathbb{LB})^k \times \mathbb{LB}$ is TM-computable by a machine M with time complexity relation τ there are a term s_M and a constant c s.t. s_M computes R with time complexity relation $(m, n_1, \ldots, n_k) \mapsto c \cdot m \cdot n_1 \cdots n_k \cdot \tau(m, n_1, \ldots, n_k) + c$.*

Proof. Define $s_M := \lambda_k . s_{\mathsf{unenc_{TM}}}(s_{\mathsf{sim}}\ s_{\mathsf{nxt}_M}\ (s_{\mathsf{prep}}(s_{\mathsf{cons}} k(\ldots(s_{\mathsf{cons}}\, 0\, \overline{[]})))))\, (\lambda x.x)\, \overline{[]}$. ◀

8 Hoare logic verification framework for Turing machines

Although the relational verification approach is quite powerful, it suffers from two crucial problems. First, realisation and termination are inherently separated proof goals, even for total machines, which form the majority of machines we are interested in. The premises of the correctness and termination relations are usually almost the same, thus manipulation of premises is duplicated. Secondly, in an interactive proof the proof context can grow very large. When two machines with say 9 tapes are sequentially composed, the proof context contains 9 assumptions on the initial tapes, 9 assumptions for the intermediate tapes between the two machines, and 9 final propositions on the tapes. Many of these assumptions are equalities. The Turing machine verification framework has naive tactics to simplify such proof goals by substitution and rewriting, which becomes quadratically slower the longer a realisation proofs takes.

We propose a verification framework based on Hoare logic, solving both problems. Correctness and termination can be proved at once, eliminating the need for repeated manipulation of premises. Furthermore, at each step in the verification of an n tape TM, the user sees exactly n specifications S for the n tapes, plus optionally a custom invariant depending on both the tapes and the label.

We here give a high-level overview. More details are in the separate Appendix B [8].

The Hoare logic is built as a new layer of abstraction for the relational framework. Given $M : \mathsf{TM}^n_\Sigma(L)$, a predicate $P \subseteq \mathsf{tp}^n_\Sigma$ and a relation $Q \subseteq L \times \mathsf{tp}^n_\Sigma$, we write *weak* Hoare triples:

$$\vDash \{P\}\ M\ \{Q\} := M \vDash (\lambda \boldsymbol{t}\ (\ell, t').\ P\ \boldsymbol{t} \to Q\ (\ell, t'))$$

To state that the machine is functionally correct and terminates in a certain time, we use *total* Hoare triples. In addition to the relation of the weak Hoare triple, a total Hoare triple asserts that the machine terminates in (at most) i steps if the precondition is satisfied. Thus, we avoid spelling out the precondition again.

$$\vDash \{P\}^i\ M\ \{Q\} := (\vDash \{P\}\ M\ \{Q\})\ \wedge\ M \downarrow (\lambda \boldsymbol{t}\ i'.\ P\ \boldsymbol{t}\ \wedge\ i \le i')$$

Similar to the old verification framework, every building block like While and Switch comes with an associated Hoare triple, shown in Figure 1 in the separate Appendix B [8]. Using these rules, an interactive verification is akin to a symbolic execution of machines with explicitly annotated invariants.

The Hoare triples of user-defined machines $M : \mathsf{TM}^n_\Sigma$ exclusively have triples with both pre and post conditions of the form $S_1 \wedge \dots S_n \wedge I$ where S_i for $1 \le i \le n$ are either $\boldsymbol{t}[i] \simeq x$ for some x, $\mathsf{isVoid}\,\boldsymbol{t}[i]$, or $\boldsymbol{t}[i] = t_0$ for some fixed t_0, and I is a custom, user-chosen invariant. Thus, the following specification for a binary machine

$$\begin{aligned}
&M \vDash \lambda \boldsymbol{t}\,(\ell, \boldsymbol{t'}).\ \forall (x : X)\,(y : Y).\ P_X x \to P_Y y \to \\
&\qquad \boldsymbol{t}[0] \simeq x \to \boldsymbol{t}[1] \simeq y \to \boldsymbol{t'}[0] \simeq f(x,y) \wedge \boldsymbol{t'}[1] \simeq g(x,y) \\
&M \downarrow \lambda \boldsymbol{t}\, i.\ \exists (x : X)\,(y : Y).\ P_X x \wedge P_Y y \wedge \boldsymbol{t}[0] \simeq x \wedge \boldsymbol{t}[1] \simeq y \wedge i \ge \tau x y
\end{aligned}$$

can be compactly restated using the following triple:

$$\forall xy.\ P_X x \to P_Y x \to \vDash \{\lambda \boldsymbol{t}.\ \boldsymbol{t}[0] \simeq x \wedge \boldsymbol{t}[1] \simeq y\}^{\tau x y}\ M\ \{\lambda (\ell, \boldsymbol{t'}).\ \boldsymbol{t'}[0] \simeq f(x,y) \wedge \boldsymbol{t'}[1] \simeq g(x,y)\}$$

A typical workflow for the verification of total machines then looks as follows: First, the user defines Hoare triples for their machines, following the shape $S_1 \wedge \dots \wedge S_n \wedge I$. Secondly, they perform the correctness proof. Thirdly, they define the time complexity function of the machine, and add the running time to the triple in the proved lemma. Fourthly, they replay the proof script and in the very last step of the verification, show that the accumulated running time is indeed bounded by the time complexity function.

9 L-computable relations over $\mathbb{LB}$ are TM-computable

To convert a term s computing a relation R, we follow the same strategy as in Section 7. We exemplarily show the specification of the machine M_{conv} corresponding to s_{unencTM}:

▶ **Fact 22.** *There is a machine $M_{\mathsf{unencL}} : \mathsf{TM}^4_\Sigma$ and a constant c s.t. for all $l : \mathbb{LB}$ we have $\vDash \{P_l\}^{c\cdot|l|} M_{\mathsf{unencL}} \{Q_l\}$ where $P_l := \lambda \boldsymbol{t}.\ \boldsymbol{t}[0] \simeq \bar{l} \wedge \boldsymbol{t}[1] = \mathsf{niltp} \wedge \mathsf{isVoid}\, \boldsymbol{t}[2] \wedge \mathsf{isVoid}\, \boldsymbol{t}[3]$ and $Q_l := \lambda(\_, t').\ \boldsymbol{t}[1] = \mathsf{midtp}\, []\, \mathsf{bl}\, \bar{l} \wedge \mathsf{isVoid}\, \boldsymbol{t}[2] \wedge \mathsf{isVoid}\, \boldsymbol{t}[3]$.*

▶ **Theorem 23.** *If $R \subseteq (\mathbb{LB})^k \times \mathbb{LB}$ is* L*-computable by a closed term s with time complexity relation τ there is a polynomial p and a Turing machine M_s s.t. M_s computes R with time complexity relation $(m, n_1, \dots, n_k) \mapsto p(m, n_1, \dots, n_k, \tau_R(m, n_1, \dots, n_k))$.*

Proof. The machine M_s is defined parametric in s and by recursion on k, i.e. we define a different machine for each k. First, M_s reads its input and writes the task stack $[(\gamma(s\, \overline{n}_1\, \overline{n}_k), 0)]$ to an auxiliary tape. As a total subroutine, we verify this part using the Hoare framework. Then M_s runs M_{sim}. If the computation terminates, the resulting heap will contain a term $\bar{l}$ for a list $l : \mathbb{LB}$. Thus, M_s runs a machine implementing unf and lastly runs M_{unencL}.

The size of the heap after i steps is in $\mathcal{O}(i \cdot (i + N))$, where $N := n_1 + \dots + n_k$. For one reduction step, N variables might have to be looked up, resulting in a runtime of $\mathcal{O}(i \cdot (i + N + 1) \cdot (N + 1))$ per step. Unfolding a heap takes $\mathcal{O}((m+1) \cdot (H + N + 1))$, where H is the size of the heap. In total, we can thus define the polynomial p cubic in the number of steps i, quadratic in N and linear in the size of the output m as follows, where c is a constant:

$$(m, n_1, \dots, n_k, i) \mapsto c \cdot (i+1) \cdot (i + n_1 + \dots + n_k + 1) \cdot ((n_1 + \dots + n_k + 1) \cdot (i+1) + m). \quad \blacktriangleleft$$

10 Discussion

We have presented the first mechanised proof of an instance of the time invariance thesis, connecting L with Turing machines with a polynomial overhead in time. We prove two variants, respectively concerned with simulation of one model of computation on the other, and with the computability of relations on boolean strings. In total, including dependencies, our development consists of 30.000 LoC and takes about 18 minutes to compile on an Intel Core i7-6600U CPU @ 2.60GHz machine. The novel code for this paper still constitutes 7800 LoC, with about 40% specification and 60% proofs.

It is folklore that the two variants are equivalent. For instance, in [1,7,23] more emphasis is put on the simulation variant. In the interactive theorem proving community it is however well-known that "folklore" is not equivalent to "easy to mechanise". In the setting of computability and complexity theory, where one has to deal with models of computation, this fact is amplified: proofs on paper virtually never provide concrete implementations in a model of computation, but focus on the high-level invariants of a proof. This is based on the (again folklore) fact that algorithms can be implemented in models of computation provided enough time and strength to sustain the tedium to do so. However, the proof engineering time needed to show the correctness of an algorithm (as e.g. a Coq function) and to show the correctness of its implementation (as e.g. a Turing machine) are completely independent. Correctness proofs of algorithms depend on the intricacy of invariants. Correctness proofs of implementations depend on the length of the code of a Turing machine, and the size of the gap between specification language and Turing machines. If the specification is a first-order, tail-recursive Coq function, the length of the Turing machine is the main factor.

We however hope that future mechanised proofs of results in complexity theory and of the time invariance thesis for other models of computation can profit from our development.

For us, simulating Turing machines on L and simulating L on Turing machines was the only possible way. Now that L is shown reasonable for time complexity, future proofs of the time invariance thesis can choose which reasonable model to use for each direction of the proof. For the time invariance thesis for a model of computability C, one can show that C can simulate Turing machines, which are structurally simple and where computation is local. In the other direction, one can show that C can be simulated in L, which has rich structure and supports non-local computation on almost arbitrary (first-order) data-structures.

For our concrete cases of Turing machines and L, powerful abstraction layers and verification frameworks are necessary for mechanisation. We would assess that a "manual" approach to any invariance thesis involving Turing machines is completely unfeasible. The Turing machine verification framework [9] and the certifying extraction framework [6] proved very valuable in this regard. While having similar goals, they use different mathematical approaches to both verification and time complexity analysis.

First, the certifying L-extraction framework gives support to automatically prove the computability of a large subset of Coq functions. For time complexity, a user has to give a time complexity function as input, and the framework automatically generates equations the function has to fulfil and furthermore provides tactics to solve these equations. Finding a time complexity function can prove challenging, but an interactive approach where a wrong function is picked and a correct function is reverse-engineered from the equations works well. A priori, the framework does not support partial terms. The `Lsimpl` tactic used in the framework however can be used to normalise terms in manual verification.

Secondly, the Turing machine verification framework provides tools to verify the correctness, time and even space complexity of Turing machines, but the user has to implement those machines manually. For the implementation, the user can write Turing machines in the style of a register based while-language, for which a canonical realisation and termination relation are inferred automatically. Correctness and time complexity proofs w.r.t. user-defined relations are now simply inclusion proofs between the canonical and the user-defined relations. Due to the split of correctness and termination, the framework works well for the verification of partial machines. However, for total machines, termination and correctness are distinct proof goals and have to be proved separately, which leads to a mathematical duplication of similar proof goals, sometimes even to actual proof code duplication.

The novel Hoare logic verification framework we present remedies this situation: For total machines, only one proof subsuming both correctness and time complexity has to be carried out, while it still supports separate proof goals for partial machines. No canonical relations are used. The verification of Hoare triples is carried out directly on the implementation of the machine, using the proof rules for the used combinators and user-defined machines. We conjecture that the Hoare logic framework scales further and possibly far. Our simulation of L on Turing machines is reasonable on time, but if terms exhibit pointer explosion [7], the space overhead might not be constant factor. The proof of this stronger time and space invariance thesis for L [7] is considerably more complicated, but might now be in reach.

However, the assessment that "Turing machines as model of computation are inherently infeasible for the formalisation of any computability or complexity theoretic result" [9] still stands. Future developments of mechanised results in complexity theory can and should be based on L or similarly well-suited models. Our certifying extraction framework provides valuable help in doing so, but there are interesting extensions worth exploring in the future.

First, this framework does not cover space complexity, which would however be needed for a proof of the full invariance thesis for L. Secondly, exploring an automatic generation of time-complexity functions, where the user can then provide a (polynomial) upper bound for the function in a second step might be interesting. Lastly, the framework requires user input for every single recursive function used, also for auxiliary functions, and the automatic verification of these is based on `Ltac` tactics which sometimes fail, and sometimes are slow. Here, an automatic generation of correctness proofs based on a meta-programming tool for Coq like MetaCoq [24, 25] might be a possible solution.

References

1 Beniamino Accattoli and Ugo Dal Lago. (Leftmost-Outermost) Beta Reduction is Invariant, Indeed. *Logical Methods in Computer Science*, 12(1), 2016. `doi:10.2168/LMCS-12(1:4)2016`.

2 Andrea Asperti and Wilmer Ricciotti. Formalizing Turing Machines. In *Logic, Language, Information and Computation*, pages 1–25. Springer, 2012.

3 Guy E. Blelloch and John Greiner. Parallelism in Sequential Functional Languages. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*, pages 226–237, 1995. `doi:10.1145/224164.224210`.

4 Alonzo Church. A set of postulates for the foundation of logic. *Annals of mathematics*, pages 346–366, 1932.

5 Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

6 Yannick Forster and Fabian Kunze. A Certifying Extraction with Time Bounds from Coq to Call-By-Value Lambda Calculus. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:19, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.ITP.2019.17`.

7 Yannick Forster, Fabian Kunze, and Marc Roth. The weak call-by-value λ-calculus is reasonable for both time and space. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–23, 2019.

8 Yannick Forster, Fabian Kunze, Gert Smolka, and Maximilian Wuttke. Appendix b: A mechanised proof of the time invariance thesis for the weak call-by-value λ-calculus. Technical report, Saarland University, 2021. URL: `https://www.ps.uni-saarland.de/Publications/documents/AppendixB_ForsterEtAl_2021_Mechanised-Time-Invariance-L.pdf`.

9 Yannick Forster, Fabian Kunze, and Maximilian Wuttke. Verified programming of Turing machines in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 114–128, 2020.

10 Yannick Forster and Dominique Larchey-Wendling. Certified undecidability of intuitionistic linear logic via binary stack machines and minsky machines. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 104–117, 2019.

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 *The Sixth International Workshop on Coq for Programming Languages (CoqPL 2020).*, 2020. URL: `https://github.com/uds-psl/coq-library-undecidability`.

12 Yannick Forster and Gert Smolka. Weak call-by-value lambda calculus as a model of computation in Coq. In *International Conference on Interactive Theorem Proving*, pages 189–206. Springer, 2017.

13 John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

14 Jan Martin Jansen. *Programming in the λ-Calculus: From Church to Scott and Back*, pages 168–180. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-40355-2_12.

15 Fabian Kunze, Gert Smolka, and Yannick Forster. Formal small-step verification of a call-by-value lambda calculus machine. In *Asian Symposium on Programming Languages and Systems*, pages 264–283. Springer, 2018.

16 Ugo Dal Lago and Beniamino Accattoli. Encoding Turing machines into the deterministic lambda-calculus. *arXiv preprint arXiv:1711.10078*, 2017.

17 Ugo Dal Lago and Simone Martini. The weak lambda calculus as a reasonable machine. *Theor. Comput. Sci.*, 398(1-3):32–50, 2008. doi:10.1016/j.tcs.2008.01.044.

18 Dominique Larchey-Wendling and Yannick Forster. Hilbert's tenth problem in Coq. *arXiv preprint arXiv:2003.04604*, 2020.

19 Torben Æ. Mogensen. Efficient self-interpretation in lambda calculus. *JOURNAL OF FUNCTIONAL PROGRAMMING*, 2:345–364, 1994.

20 Joachim Niehren. Uniform confluence in concurrent computation. *Journal on Functional Programming*, 10(5):453–499, 2000.

21 Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical computer science*, 1(2):125–159, 1975.

22 Damien Pous. A certified compiler from recursive functions to minski machines. Technical report, Université Paris-Diderot, PPS, 2004. URL: http://perso.ens-lyon.fr/damien.pous/recursive-minski/.

23 Cees F. Slot and Peter van Emde Boas. On Tape Versus Core; An Application of Space Efficient Perfect Hash Functions to the Invariance of Space. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1984, Washington, DC, USA*, pages 391–400, 1984. doi:10.1145/800057.808705.

24 Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, 2020. doi:10.1007/s10817-019-09540-0.

25 Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–28, 2019.

26 The Coq Development Team. The coq proof assistant, version 8.12.0, July 2020. doi:10.5281/zenodo.4021912.

27 Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

28 Maximilian Wuttke. Verified programming of Turing machines in Coq. Bachelor's thesis, Saarland University, 2018.

29 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.

A Definition of Turing machines

We compare the definition of Turing machines we use [9] to the one by Hopcroft, Motwani, and Ullman [13].

1. The logical system in [13] is classical set theory, whereas we work in constructive type theory. We discuss the impact of this foundational choice below.
2. The alphabet in [13] is separated into a set of input symbols Σ and a superset of tape symbols Γ, where we unify both into a single type.

3. The blank symbol in [13] is an explicit part of Turing machines as an element of Γ, but not of Σ. We do not specify blank symbols explicitly and instead leave it to a user to specify a blank symbol or even various blank symbols.
4. Tapes in [13] are not formally defined. It is only stated that tapes 'extend infinitely to the left and right, initially hold[ing] [...] the input'. Instantaneous descriptions of Turing machines are formally defined as strings over Γ and Q, which contain 'the portion of the tape between the leftmost and the rightmost non-blank, unless the head is to the left of the leftmost non-blank or to the right of the rightmost non-blank.' In the latter case, the blanks between the head and the non-blank content are part of the string.
5. The transition function in [13] is a partial function, whereas ours is total. If the transition function is unspecified, the computation of the machine halts, whereas we have an explicit boolean halting function. In Coq's type theory one requires classical logic and $\mathsf{AUC}_{\mathbb{N},\mathbb{N}}$ to compile Turing machines with a partial transition function into an equivalent definition with total transition functions. In general, any compilation of partial functions on finite types to total functions is non-computable and thus not definable in Coq's type theory without axioms.
6. A machine in [13] always writes a symbol and always moves to the left or right. We allow to not write a symbol and to not move the head. The first is important regarding our definition of tapes (otherwise a fully empty niltp can never stay fully empty), whereas the second is a relatively arbitrary choice to allow more freedom in the definition of concrete machines.
7. Turing machines have an explicit set of accepting states in [13]. We do not add one, because our definition is not aimed at formalising computability theory directly. Instead, our more flexible definition of labels subsumes the binary notion of accepting states, but allows for more interesting constructions like the Switch and MemWhile machines.

Subtle difficulties might arise when defining notions of computability theory in classical set theory. For instance, defining Turing machines with a transition function which takes as input the whole tape is problematic: Nothing permits non-computable transition functions then, making arbitrary problems decidable by encoding the decision into the transition function. When imposing the transition function to be computable one obtains a circular dependency: The notion of computability is needed to define transition functions, transition functions are needed to define Turing machines, but Turing machines are needed to define the notion of computability. The well-known solution here is to define Turing machines as finite objects, i.e. let the transition function, although partial, work on a finite domain and codomain. In classical set theory one can then always show afterwards that this transition function is computable, since every function with finite domain and codomain is. In our type-theoretic setting, we can also show that the transition function is computable, provided it is, as we defined it, a total function. We did so in Section 6.

Mechanising Complexity Theory: The Cook-Levin Theorem in Coq

Lennard Gäher
Universität des Saarlandes, Saarland Informatics Campus, Saarbrücken, Germany

Fabian Kunze
Universität des Saarlandes, Saarland Informatics Campus, Saarbrücken, Germany

Abstract

We mechanise the Cook-Levin theorem, i.e. the NP-completeness of SAT, in the proof assistant Coq. We use the call-by-value λ-calculus L as the model of computation to formalise time complexity, the class NP, and polynomial-time reductions. The latter two notions agree with the usual characterisations via Turing machines (TMs), as L and TMs are polynomial-time equivalent.

The use of L as the computational model, as opposed to TMs, significantly eases program verification and the derivation of resource bounds. However, for showing the NP-hardness of SAT, computations of L need to be encoded in SAT, which is complicated by L's more complex computational structure. Thus, the polynomial-time reduction chain to SAT employs TMs as an intermediate *problem*, for which we neatly factor out a known textbook reduction from TMs to SAT. Still, all *reduction functions* are implemented and analysed in L.

To the best of our knowledge, this is the first result in *computational* complexity theory that has been mechanised with respect to any concrete computational model.

We discuss what makes this area of computer science hard to mechanise and highlight the design choices which enable our mechanisations.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness; Theory of computation → Complexity classes; Theory of computation → Lambda calculus; Theory of computation → Logic and verification; Theory of computation → Constructive mathematics

Keywords and phrases computational model, NP completeness, Coq, Cook, Levin

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.20

Supplementary Material *Software (Code Repository)*: `https://github.com/uds-psl/cook-levin` archived at `swh:1:dir:7d1c46883525acb8bbc1fc6164e1daaac1ec6e13`

Funding *Lennard Gäher*: was supported in part by the International Max-Planck Research School on Trusted Computing (IMPRS-TRUST).

1 Introduction

Computational complexity theory studies how efficiently problems can be solved. This subsumes the mere analysis of the resource usage of specific algorithms: Problems are classified according to their inherent (time and space) resource requirement, revealing a rich structure of relations between these complexity classes. Even practitioners make use of NP, the class of problems for which solution candidates can be verified in polynomial time: Showing that a problem p is NP-hard, i.e. at least as hard as the hardest problem in NP, is an established criterion for the infeasibility of solving p efficiently in the general case. And NP is even known outside the area of computer science due to one of the most famous[1] open questions of modern mathematics, whether $\mathsf{P} \overset{?}{=} \mathsf{NP}$, i.e. whether every problem whose solutions can be verified efficiently can also be solved efficiently.

[1] `https://www.claymath.org/millennium-problems/p-vs-np-problem`

Despite their relevance, complexity-theoretic results are seldom if ever proven in all detail, e.g. relying on the reader's intuition for the exact implementation or resource bounds of algorithms. While computability-theoretic results have been successfully mechanised in proof assistants [15, 6, 23], few mechanisations of even basic complexity theory are available. For computability theory, *synthetic* approaches [9, 4] have proved successful, by avoiding the use of an explicit computational model and relying on the fact that all functions definable in constructive theories such as Coq's are computable. Many results in *computational complexity theory*, on the other hand, inherently require proofs with respect to a concrete model of computation, for two reasons: First, the resource analysis of "programs" needs to be carried out with a *reasonable* resource model that is connected to the definitions of complexity classes, which rules out a synthetic approach. Secondly, many *structural* results involve universal quantifications over problems in certain complexity classes, necessitating the use of properties of an underlying computational model. For instance, showing that a problem p is NP-complete requires proving that *all* problems q verifiable in polynomial-time reduce to p in polynomial time. Thus, the essential requirements for mechanising complexity theory are to be able to program in a model of computation and to use properties of it.

Related Work. Past mechanisations of computational complexity theory have usually either assumed an abstract computational model satisfying certain conditions, but have not proved the existence of such a model, or have focused on merely proving functional correctness of constructions from complexity theory, without connecting the result to a cost model proved reasonable.

Asperti [2, 1] axiomatically proves the hierarchy theorems and Borodin's gap theorem in the setting of *reverse complexity theory* using the proof assistant Matita, in an effort to determine the minimal assumptions on a computational model needed to obtain the theorems.

Gamboa and Cowles [16] verify the construction of the Cook-Levin theorem for Turing machines in ACL2. They define functions counting the steps taken by their implementation, and, without any computational model, cannot establish any connection to the class NP.

Other attempts [12] have been made at mechanising basic complexity theory using Turing machines (TMs) as the computational model. Forster et al. [12] implement a multi-tape to single-tape Turing machine compiler and a universal Turing machine, both with polynomial time overhead and constant-factor space overhead. They conclude, after implementing a framework for programming TMs consisting of 19K lines of Coq code, that "TMs as model of computation are inherently infeasible for the formalisation of any computability or complexity theoretic result", owing to the non-compositionality and low-level structure of TMs. Instead, they conjecture that using a λ-calculus as model of computation may prove more successful.

Key Idea: L for Complexity Theory. We follow the idea of using a λ-calculus and choose the call-by-value λ-calculus L [15] as our model of computation. One big overhead of mechanised complexity theory is the verification and resources analysis of algorithms in the model of computation. It is simple to compositionally program in L since inductive datatypes and recursive functions can be encoded systematically, similar to functional programming languages. We use the certifying extraction mechanism from Coq to L by Forster and Kunze [10] to prove computability and semi-interactively deduce the running time of algorithms. This allows us to program and verify algorithms using the usual comforts of Coq, automatically establish computability in L, and derive resource bounds in Coq almost completely without descending into the computational model.

L has been shown to be a *reasonable* computational model [11], in the sense of the invariance thesis of Slot and van Emde Boas [26]: TMs and L can simulate each other with a polynomial overhead in time and a constant-factor overhead in space for decision problems. Thus, one obtains the same polynomial-time complexity classes for L as for TMs, which justifies using L for the mechanisation of complexity theory. The two key requirements for mechanising *structural* results are thus elegantly satisfied by L.

Overview. In this paper, we formalise the basics of polynomial-time complexity theory for L, namely the classes P and NP, as well as the notions of NP-hardness and NP-completeness.

As a first example of using L for complexity theory, we mechanise the popular Cook-Levin theorem in Coq. This theorem essentially founded the class of NP-complete problems when it was independently discovered by Cook [7] and Levin [21] in 1971. In the formulation by Cook, the satisfiability problem of Boolean formulas SAT is shown NP-complete.

The importance of the Cook-Levin theorem lies not in proving that *specifically* SAT is NP-complete, but that a *natural*, *machine-independent* problem is NP-complete. Once such a problem p has been shown to be NP-complete, it is relatively easy to derive NP-completeness of other problems by reduction from p.

In contrast, in any machine model one can usually define a *generic* NP*-complete problem* which is machine-dependent and usually straightforward to prove NP-complete. Essentially, the existence of an input accepted by a polynomial-time-verifier is reformulated as a problem on triples of a program (the verifier), a size bound, and a time bound.

Usually, the proof of the Cook-Levin theorem proceeds by reduction from a generic problem. Specifically, for L this would require us to encode the computation of L-terms using Boolean formulas. As λ-calculi have a very much non-local computational structure due to variable binding, formally verifying a direct encoding seems intractable.

On the other hand, encodings of TMs are conceptually simple (though still technically challenging): single-tape TMs can only locally change a constant amount of data in each step of computation. Therefore, we use TMs as an intermediate problem: L is encoded using TMs (employing previous related mechanisations [12, 13]) and in turn TMs are encoded as Boolean formulas. The expressivity of TMs seems better-suited than Boolean formulas for an encoding of L-computations. The full reduction is still computed and analysed in L. TMs serve only as an intermediate construct.

Definitions, lemmas, and theorems in this document are hyperlinked with the documentation of the Coq development; the links are marked by the symbol 🐓.

Contribution. In summary, our contributions are as follows:

1. We mechanise the theory of polynomial-time complexity for L, in particular the notions of polynomial-time computability, NP, NP-hardness, and NP-completeness,
2. and provide the first mechanisation of a *structural* result in complexity theory, namely the Cook-Levin theorem, that includes a running time analysis with respect to the reasonable computational model L, without any axioms.
3. Our reduction from Turing machines to SAT neatly factorises and formalises a textbook proof by Sipser [25].
4. We have developed techniques and automation to make time resource analyses on top of the L extraction framework [10] more feasible.

Structure. Section 2 introduces notation and type-theoretic preliminaries, as well as the most important aspects of TMs. We then continue with a detailed account of our basic complexity definitions in Section 3. Section 4 contains a high-level overview of the reduction and the involved problems. Section 5 and Section 6 elaborate on some details of the reductions. Section 7 gives a general overview of the mechanisation and in particular the mechanisation of resource analyses. Finally, we discuss our results and future work in Section 8.

2 Preliminaries

We work in a constructive type theory with inductive types and an impredicative universe of propositions $\mathbb{P}$ as implemented in Coq.

The basic inductive types we use are the Booleans $\mathbb{B} ::= \mathsf{true} \mid \mathsf{false}$, the unit type $\mathbb{1} ::= ()$, the natural numbers $\mathbb{N} ::= 0 \mid \mathsf{S}\,n$ for $n : \mathbb{N}$, product types $X \times Y$, and sum types $X + Y$. π_1 and π_2 denote the projections out of the product type. Given a type X, we further define options $\mathcal{O}(X) ::= \emptyset \mid \lfloor x \rfloor$ and lists (or strings) $X^* ::= [] \mid x :: A$ for $x : X$ and $A : X^*$. On lists we employ the standard notation for membership $x \in A$, inclusion $A \subseteq B$, concatenation $A +\!\!+ B$, and length $|A|$. The notation $A[i, j]$ is used to denote the sublist of A from positions i to j (inclusive), possibly having less than $j - i + 1$ elements if the indices are (partially) invalid. $[a, b]$ denotes the list of numbers between a and b (inclusive, possibly empty). $[\ f\ a \mid a \in A\ ]$ is the list obtained by mapping f over A.

We say that a type X is discrete if there is a function $X \to X \to \mathbb{B}$ deciding equality. We say that a type X is finite if it is discrete and there is an exhaustive list of elements $[x_0, \dots, x_n]$. In this case, we may write $\{x_1, \dots, x_n\}$ for X. In particular, we will use the letters Σ, Γ to denote finite types representing alphabets.

Turing Machines. We use the formalisation of deterministic, two-sided infinite TMs by [3] and the mechanisation by [12]. Unlike standard presentations, the tape alphabet Σ does not feature a distinguished blank symbol, so a tape contains one continuous sequence of non-blank symbols. Although we make use of multi-tape machines, for this exposition we restrict our attention to single-tape machines. We assume a type of tapes Tape_Σ with operations $\mathsf{left}, \mathsf{right} : \mathsf{Tape}_\Sigma \to \Sigma^*$ giving the parts of the tape left and right of the head, as well as $\mathsf{current} : \mathsf{Tape}_\Sigma \to \mathcal{O}(\Sigma)$ yielding an optional symbol under the head. $|tp|$ denotes the number of symbols on a tape tp.

A Turing Machine $M : \mathsf{TM}_\Sigma$ is a tuple $(Q, q_0, \delta, \mathsf{halt})$ consisting of a finite type of states Q, an initial state q_0, a transition function $\delta : Q \times \mathcal{O}(\Sigma) \to Q \times \mathcal{O}(\Sigma) \times \mathsf{Move}$, where $\mathsf{Move} ::= \mathsf{L} \mid \mathsf{N} \mid \mathsf{R}$, and a function determining the halting states. A configuration is a pair (q, tp) of a state and a tape. We use the notation $(q, tp) \succ (q', tp')$ for configuration changes according to δ and write $(q, tp) \rhd^{\leq t} (q', tp')$ for $\leq t$ steps where additionally $\mathsf{halt}(q') = \mathsf{true}$.

3 Polynomial-Time Complexity Theory in L

L [15], the underlying model of computation in this work, is a standard untyped λ-calculus with weak call-by-value reduction $\succ$. This section presents the relevant definitions for L and the definitions of standard notions of polynomial-time complexity theory, adapted to L as the computational model.

Its terms $s, t : \mathsf{Ter} ::= x \mid \lambda x.t \mid s\ t$ feature lambda abstractions and applications (the formalisation uses de-Bruijn indices instead of named variables). We use $\|s\|$ to denote the (syntactic) size of a term s. The reduction relation $\succ$ is uniformly confluent [15], meaning that each reduction path to a normal form has the same length[2]. This allows us to speak about *the* number of steps that a terminating term takes. Many types X can be encoded as L-terms by defining an encoding function $\ulcorner\cdot\urcorner : X \to \mathsf{L}$. Inductive first-order datatypes (like Peano numbers and lists) can be represented systematically using the Scott encoding [15, 22, 19], to which we will default from now on.

The following definitions that are parametric over types X, Y have the implicit requirement that these types are L-encodable. Note that whenever using a notion from this section, the precise encoding used is an important part of the statement, e.g. the addition of binary numbers can be performed more efficiently than the addition on a unary encoding.

As the time measure, we use the number of β-reduction steps to a normal form, which has been shown to be a reasonable computational model [5]. Forster and Kunze [10] have mechanised a notion of L-computability with running time for meta-level functions that can be encoded as L-terms. We refer to this work, and use this notion slightly informally here.

▶ **Definition 1** (Polynomial Boundedness). *We say that a function $f : \mathbb{N} \to \mathbb{N}$ is polynomial if there exists $k : \mathbb{N}$ such that for sufficiently large n, we have $f n \leq k \cdot n^k$. We say that a function $g : \mathbb{N} \to \mathbb{N}$ is polynomially bounded if there exists a monotonic, polynomial function f with $\forall x. g(x) \leq f(x)$.*

The monotonicity requirement in the definition eases formal reasoning by allowing rewriting in the function argument.

Definition 2 (Polynomial-time Computability). *A function $f : X \to Y$ is polynomial-time computable, if there exists $t : \mathbb{N} \to \mathbb{N}$ such that*

- *f is L-computable in time $\lambda x.t\ \|\ulcorner x\urcorner\|$,*
- *t is polynomially bounded, and*
- *the result size is polynomially bounded by the input size, i.e. there is a polynomial p with $\|\ulcorner f\ x\urcorner\| \leq p\|\ulcorner x\urcorner\|$.*

We use the term size of the (Scott) encoding, $\|\ulcorner\cdot\urcorner\|$, as the input size of computations. Note that the last requirement of Definition 2 is non-standard: For TMs, this condition holds automatically due to the known "time-bounds-space" lemma of TMs, while for L, a computation can produce terms of a size exponential in the number of reduction steps [11].

Definition 3 (P). *A decision problem $p : X \to \mathbb{P}$ can be decided in polynomial time, written $p \in \mathsf{P}$, if there exists $f : X \to \mathbb{B}$ such that f is polynomial-time computable and f decides p, i.e. $p\ x \leftrightarrow f\ x = \mathsf{true}$ for all x.*

We define NP using the well-known characterisation via deterministic certificate verifiers [24, p. 181], instead of e.g. extending L to support non-deterministic computations.

Definition 4 (NP). *A decision problem can be verified in polynomial time, written $p \in \mathsf{NP}$, if there exists a verifier relation $R : X \to \mathsf{Ter} \to \mathbb{P}$ and a certificate size bound $f : \mathbb{N} \to \mathbb{N}$ such that*

1. *$\lambda(x, y).R\ x\ y$ is polynomial-time decidable (we call this decider* verifier*),*
2. *$R\ x\ y$ implies $p\ x$,*
3. *$p\ x$ implies $R\ x\ y$ for some y with $\|\ulcorner y\urcorner\| \leq f\|\ulcorner y\urcorner\|$, and*
4. *f is polynomially bounded.*

[2] A single reduction step is not deterministic as there is no enforced order of evaluation in applications.

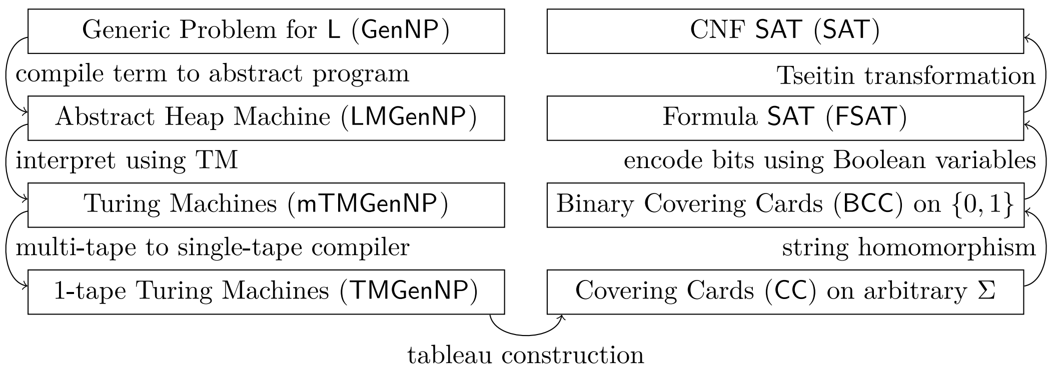

Generic Problem for L (GenNP)
compile term to abstract program
Abstract Heap Machine (LMGenNP)
interpret using TM
Turing Machines (mTMGenNP)
multi-tape to single-tape compiler
1-tape Turing Machines (TMGenNP)
tableau construction
Covering Cards (CC) on arbitrary Σ
string homomorphism
Binary Covering Cards (BCC) on {0, 1}
encode bits using Boolean variables
Formula SAT (FSAT)
Tseitin transformation
CNF SAT (SAT)

Figure 1 Chain of reductions from GenNP to SAT.

Note that this definition (unlike [24]) does not require the verifier to reject certificates which are too large, i.e. exceed the polynomial bound. This simplifies the verifier relation and implementation in formalised NP-containment proofs, as properties on the logical level of R are less tedious than implementing more tests in the function. Moreover, this definition only allows L-terms as certificates. This simplifies the formalisation in places where NP-containment is an assumption (e.g. when proving NP-hardness), as otherwise one would need to spell out requirements on the certificate type, like efficient enumerability, of the definition of NP.

In the other direction, when establishing NP-containment, a generalisation allows any type Y for certificates, as long as its encoding is injective and decodable in polynomial time.

Finally, the notion of polynomial-time reducibility enables us to define the two crucial notions relating problems to NP.

Definition 5 (Polynomial-time Many-one Reductions). *A problem $p : X \to \mathbb{P}$ reduces to $q : Y \to \mathbb{P}$ in polynomial-time, written $p \preceq_p q$, if there exists a polynomial-time computable function $f : X \to Y$ such that $p\ x \leftrightarrow q(f x)$.*

Definition 6 (Hardness and Completeness). *A problem $p : X \to \mathbb{P}$ is NP-hard if for all problems $q \in$ NP, $q \preceq_p p$. If additionally $p \in$ NP, then p is NP-complete.*

The standard closure properties hold, enabling reductions as a useful tool:

▶ **Lemma 7.**
1. *If $p \in$ NP and $q \preceq_p p$, then $q \in$ NP.*
2. *If p is NP-hard and $p \preceq_p q$, then q is NP-hard.*
3. *$\preceq_p$ is a pre-order.*

4 Overview of the Reduction

The hard part of proving the Cook-Levin theorem is the NP-hardness proof of SAT, on which we focus in this paper. For this, we construct a polynomial-time many-one reduction from a generic problem, which can be easily shown NP-hard, to SAT. Our reduction factorises in several intermediate problems, of which we give a brief overview in this section. For the most interesting parts, we give more technical details in Sections 5 and 6.

It turns out that for different models of computation, so-called *generic* NP*-hard problems* are of interest in this reduction chain: They ask the question whether, for the input triple $(\mathcal{P}, k, t) : \mathsf{Prog} \times \mathbb{N} \times \mathbb{N}$, there is a certificate c such that the program $\mathcal{P}$ halts on input c in no more than t time steps and such that k bounds the size of c. The exact notions of programs, inputs, size, and time differ per concrete model of computation. The generic problem for L can be shown NP-hard quite easily, as its definition lines up nicely with NP.

Our full reduction chain can be divided into two segments (Figure 1): First we reduce the generic NP-hard problem for L to a generic NP-hard problem for (single-tape) TMs. In a second step, single-tape TMs are encoded as Boolean formulas. While this second step closely follows the idea of a common textbook construction [25], our proof neatly separates different aspects of the construction into different reductions. One important detail is that L is used to *implement* all the reduction functions, as mentioned earlier. The reason for still having TMs as an intermediate problem is that their computational structure is much more local than that of L, simplifying the encoding as a formula.

Generic Problem for L. The generic problem for L, GenNP, asks, for the input triple (s, k, t), whether there exists a list of Booleans bs, the certificate, such that (1) the term s applied to the encoding $\ulcorner bs \urcorner$ halts in no more than t steps and (2) the encoding of bs is smaller than k. The definition of NP allows to easily establish NP-hardness of this problem, see Section 5.

Instead of directly reducing the generic problem for L to TMs, we employ an *abstract heap machine* as an intermediate step: It is a stack machine for L that has built-in structure sharing due to the use of closures and an explicitly modelled heap, similar to the one used by Forster et al. [11]. The reduction from GenNP to the generic problem for this abstract machine, $\mathsf{LMGenNP}$, uses the fact that the abstract machine implements L with a linear factor overhead in time.

Reducing to Turing Machines. The next step is to encode the heap machine using a multi-tape TM, for which we use the interpreter by Forster et al. [13]. The TM (i.e. state space and transition function) produced by the translation is fixed, i.e. this is an interpreter, not a compiler.

The most difficult part here is that the generic problem for multi-tape TMs, $\mathsf{mTMGenNP}$, uses an arbitrary tape as certificates, while the L-problems use a list of Booleans. Therefore, we need to define and verify a TM that performs some preprocessing: Each possible TM-encoding of an L-encoding of a list of Booleans must be the result of this TM on some tape, and this TM never produces something that does not encode a Boolean list.

We now use an existing translation from multi-tape to single-tape TMs [12] to reduce from $\mathsf{mTMGenNP}$ to the generic problem for single-tape TMs, $\mathsf{TMGenNP}$: Instances are dependent tuples $(\Sigma, M : \mathsf{TM}, in : \Sigma^*, k : \mathbb{N}, t : \mathbb{N})$, where Σ is a finite tape alphabet and M is a (two-sided) single-tape machine over Σ. An instance is a yes-instance if there exists a certificate $cert : \Sigma^*$ with $|cert| \le k$ such that M halts on $in \mathbin{+\!\!+} cert$ in $\le t$ steps.

Reducing to SAT using Covering Cards. For the reduction from $\mathsf{TMGenNP}$ to a Boolean formula, we follow the textbook proof by Sipser [25] at a high level. The key idea is to make use of the well-known "time-bounds-space" fact for TMs. From the fixed input part in, the maximum certificate size k, and the time bound t we can determine a maximum amount of space s the TM may use. All of the machine's computation can be layed out in a tableau of $t + 1$ lines and width s, with each line representing one configuration of the machine and each cell containing an encoding of a tape symbol or the state, see Figure 2. We need a special blank symbol $\sqcup$ filling the space not currently in use by the TM. The state symbol, initially $q_0^{\sqcup}$, not only contains the state q_0 but also marks the head's position and contains the current symbol under the head. Eventually, this tableau determines the layout of the Boolean formula.

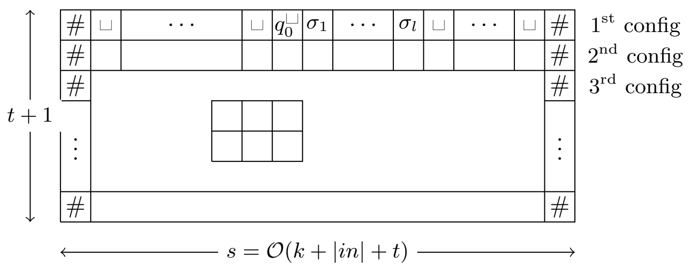


Figure 2 The tableau of configurations. Each line is delimited by #.

Exploiting the local computational structure of TMs, we constrain succeeding lines of the tableau locally with *covering cards* of 3 by 2 symbols to enforce valid configuration changes. At each possible offset of two succeeding lines, a card matching the symbols at that position needs to exist. Importantly, the cards need to overlap, which allows to express global constraints using only local cards. The set of cards available encodes the valid transitions the TM can take. Cards *can* be re-used.

Instead of directly encoding this tableau as a formula, as is done by Sipser, we introduce a string-based intermediate problem called *Covering Cards* (CC) making the tableau of configurations and covering cards explicit. This allows us to separately deal with the construction of the tableau and the TM encoding, reducing the special symbols introduced by the tableau encoding to a binary alphabet, and finally constructing a Boolean formula.

CC may use arbitrary finite alphabets for the contents of the tableau. This expressivity is needed for the reduction even if the original TM only uses a binary alphabet, as the tableau construction needs "control symbols". As a first step towards encoding this as a Boolean formula, we reduce to a binary alphabet $\{0, 1\}$ by replacing every element σ of the alphabet Σ by a unique binary string of length $|\Sigma|$. We call this variant *Binary Covering Cards* (BCC).

The next step of the reduction are general Boolean formulas φ featuring conjunction, disjunction, and negation. The general formula satisfiability problem (FSAT) has as instances formulas φ and asks whether there exists an assignment a such that φ is true under this assignment. BCC can be encoded as a Boolean formula $\varphi_{init} \land \varphi_{cards} \land \varphi_{final}$ of three gadgets for the first line of the tableau, the covering cards, and the final substring constraint. For these formulas, each cell of the tableau can be represented as one Boolean variable due to the binary alphabet.

However, usual statements of the Cook-Levin theorem reduce to the satisfiability problem of conjunctive normal forms (SAT). The formulas produced by the reduction to FSAT are not yet in CNF and so we transform formulas φ into their normal form. Since the naive way of implementing this transformation can incur an exponential blowup in the formula size, we use the well-known Tseitin transformation [27] to obtain a polynomial-time reduction.

5 From L to Turing Machines

In this section, we give some of the details of showing the generic problem for Turing machines TMGenNP NP-hard. We first show the problem GenNP to be NP-hard. We then give a path of polynomial-time reductions to TMGenNP.

The generic problem for L, GenNP, asks whether, for a triple consisting of a L-term s, and two unarily encoded natural numbers k (the size bound for certificates) and t (the time bound), there exists a list of Booleans bs with size no greater than k such that s, on the encoding of bs, halts in no more than k steps[3].

Instead of allowing all triples as possible input, we restrict the definition to a subset of *admissible* triples. Formally, the type of GenNP is a subtype, i.e. $\{x : \mathsf{Ter} \times \mathbb{N} \times \mathbb{N} \mid \mathrm{adm}\, x\} \to \mathbb{P}$ for some restriction $\mathrm{adm} : \mathsf{Ter} \times \mathbb{N} \times \mathbb{N} \to \mathbb{P}$. The reason for this is that later, when reducing from multi to single-tape TMs, this restriction simplifies the proof.

Definition 8 (Generic NP-complete problem for L)**.** *A triple* $(s, k, t) : \mathsf{Ter} \times \mathbb{N} \times \mathbb{N}$ *is a yes-instance for* GenNP *if there exists a certificate* $bs : \mathbb{B}^*$ *with* $\|\ulcorner bs \urcorner\| \leq k$ *such that* $s\ \ulcorner bs \urcorner$ *halts in no more than* t *steps.*

For GenNP*, the subtype of admissible input triples is restricted to closed abstractions* s *satisfying two constraints: If* s *halts when applied to some certificate, then there is a certificate of size* $\leq k$ *on which* s *halts. If* s *halts on a certificate, then it halts in no more than* t *steps.*

Theorem 9. GenNP *is* NP*-hard*

Proof. Assume a problem $p : Y \to \mathbb{P}$ is in NP with verifier $v : Y \times \mathsf{Ter} \to \mathbb{B}$. Let f be a polynomial-time surjection from $\mathbb{B}^*$ to Ter that in addition is non-wasteful, i.e. can produce each $c : \mathsf{Ter}$ from a boolean list that is not significantly larger than c.

The reduction function to GenNP now maps $y : Y$ to a triple: The first component is the term $s := \lambda x.\mathrm{HaltOnTrue}(s'x)$, where s' is the term computing (in polynomial time) the function $\lambda x.v(y, f x)$, and HaltOnTrue is a combinator that, applied to an encoded Boolean $\ulcorner b \urcorner$, halts only on true.

The second component, the size bound k, is chosen large enough so that there is a smaller certificate x iff the verifier v accepts some $f x$ as certificate for y. This can be achieved by concatenating the size bound for certificates (from the definition of NP) with the size-bound for f (from f being non-wasteful). The third component, the time bound, can be chosen large enough by determining the running time function of s', and then plugging in k and the size of y.

We skip over the actual correctness proof, which does not only have to account for two directions, but also the side conditions. We also skip over the polynomial-time computability proof in this presentation. ◀

We now reduce from GenNP to $\mathsf{mTMGenNP}_M$, a generic problem on multi-tape TMs [4]. Syntactically, for a fixed $n+1$ tape TM M, the inputs to $\mathsf{mTMGenNP}_M$ are triples consisting of a n-vector v of tapes, and two unarily encoded natural numbers k (the size bound for certificates) and t (the time bound). As for GenNP, only a subtype of those are admissible.

Definition 10 ($\mathsf{mTMGenNP}_M$)**.** *A triple* $(v, k, t) : \mathrm{tape}^n \times \mathbb{N} \times \mathbb{N}$ *is a yes-instance for* $\mathsf{mTMGenNP}_M$ *if there exists a tape* t_c *with* $|t_c| \leq k$ *such that* M *on input* $t_c :: v$ *halts in no more than* t *steps.*

For $\mathsf{mTMGenNP}_M$*, the subtype of admissible input triples is restricted by two constraints: (1) If* M *on* $t_c :: v$ *halts when applied to some certificate* v*, then there is a certificate shorter than* k *on which it halts. (2) If* M *halts on* $t_c :: v$*, then it halts in no more than* t *steps.*

[3] The mechanisation generalizes certificates to any type X that is polynomial-time surjectable to Ter.

[4] The mechanisation uses an abstract machine for L as intermediate step, which in hindsight is not the optimal factorisation, as nothing of interest happens in the first reduction. The better factorisation is to explicitly separate out a TM evaluating L. Since this focuses on the more interesting parts, we explain that approach here.

Compared to the definition of GenNP, we separate the program into two parts: A fixed $n+1$ tape TM M and, as part of the actual input, all-but-the-first tapes v. The reduction now needs two TMs as key ingredients: A TM CheckCert that transforms the guessed first tape, containing arbitrary symbols, into all possible encodings of Boolean lists, and a polynomial-time L-evaluator Eval_L, mechanised by Forster et al. [13].

Theorem 11. GenNP *reduces to* $\mathsf{mTMGenNP}_M$ *in polynomial time for some* M.

Proof. For a triple $(s, k, t) : \mathsf{Ter} \times \mathbb{N} \times \mathbb{N}$, the reduction function produces a triple (v, k', t') : $\text{tape}^n \times \mathbb{N} \times \mathbb{N}$ such that v contains an encoding of s. Then, k' and t' are chosen large enough such that the M we now construct satisfies all side conditions, which is tedious to mechanise and would be even more tedious to describe here.

The machine M is constructed as follows: It tests if the first tape contains a Boolean list and runs the simulator Eval_L on the application of s (read from v) to the encoding of the Boolean list. Otherwise, it diverges. ◀

The last step now is a reduction to the generic problem TMGenNP for single-tape TMs, for which we use the multi-to-single-tape translation by Forster et al. [12].

Definition 12 (TMGenNP). *A tuple* $(\Sigma, M : \mathsf{TM}, in : \Sigma^*, k : \mathbb{N}, t : \mathbb{N})$ *is a yes-instance for* TMGenNP *if there exists a tape* t_c *with* $|t_c| \le k$ *such that* M *halts on* $in +\!\!+ t_c$ *in* $\le t$ *steps.*

Note that the used tape is a combination of *in* from the input and the "guessed" certificate.

Theorem 13. *For every* M, $\mathsf{mTMGenNP}_M$ *reduces to* TMGenNP *in polynomial time.*

Proof. Given a $n+1$ tape machine M and an instance (v, k, t) of $\mathsf{mTMGenNP}_M$, we construct the instance (Σ, M', in, k', t') for TMGenNP as follows:

M' is a single tape machine that basically is the single-tape compilation of M, but we prepend an auxiliary machine that checks that the tape is the encoding of some $n+1$-vector of tapes, according to the single-to-multi-tape construction we use. M' can be hard-coded and does not have to be computed in L, as M is a parameter[5]. *in* is the proper encoding of v, for use by M'. k' is k plus some small constant due to overhead in the encoding of tapes. t' is chosen so that M' terminates, according to the running time analysis, and polynomial in k, t and v.

The two restrictions for admissible inputs to $\mathsf{mTMGenNP}_M$ are crucial for the direction where one obtains a multi-tape certificate from a terminating run of M': The construction we use only gives an upper bound for the running time of M' in terms of the running time of M. Therefore, we cannot be sure that only because M' halts in no more than k' steps, the multi-tape machine halts in k steps. Using the side conditions, just by the fact that the multi-to-single tape compiler preserves the halting behaviour, we get the existence of a certificate with the right bounds. ◀

We omit more details[6] and only note that the actual running time analysis of M', and the poly-time computability proofs of the reduction function are at least an order of magnitude larger than the parts sketched in this proof.

[5] A weaker definition of TMGenNP that is parameterised over Σ and M would suffice for our purposes, but our construction in Section 6 works for the actual, more general definition.

[6] For example that for technical reasons, we have another version of TMGenNP as intermediate step.

6 Reducing TMGenNP to SAT

In this section, we focus on the remaining steps to show SAT to be NP-complete, starting from the NP-hardness of TMGenNP.

6.1 Encoding of Turing Machines using Covering Cards

First, we present a more detailed account of encoding TMs using Covering Cards. The key steps of this proof are to find an encoding of configurations which is essentially unique and to construct the covering cards for the TM simulation, *such that* there is a one-to-one correspondence between valid CC-steps and TM steps, and to construct a *prelude* which, before the TM simulation runs, "guesses" a certificate string in a single CC-step.

Definition of CC. First, we specialise CC to a specific case we call 3-CC. 3-CC instances are dependent tuples (Γ, in, C, fin, t), where Γ is the finite alphabet for the cells of the tableau, $in : \Gamma^*$ is the first line of the tableau, and $t + 1$ is the number of lines. $C : C_\Gamma{}^*$ is a list of covering cards consisting of a premise and a conclusion, where $C_\Gamma := \Gamma^3 \times \Gamma^3$. Finally, $fin : \Gamma^*$ is a constraint on the tableau's final line s_{t+1} used for encoding that the TM has halted. An instance is a Yes-instance if there exists a sequence of strings $s_0 = in, \ldots, s_{t+1}$ such that $s \in s_{t+1}$ for an $s \in fin$, denoting that one of the symbols of the last line signals halting, and such that $s_i \rightsquigarrow_C s_{i+1}$, denoting that every succeeding line follows validly, at all positions, from the previous one according to the cards C. Formally, $s \rightsquigarrow_C t := \forall i \in [0, w-3]. \exists a \in C. s[i, i+2] = \pi_1 a \wedge t[i, i+2] = \pi_2 a$, where $w := |in|$. Note that CC does enforce that there is a matching rule for every position, in contrast to rewriting systems, which would apply a rule at a single position and leave other parts unchanged. We call this a CC-step from s to t.

The generalisation of 3-CC to CC (needed for the reduction to a binary alphabet) makes the following changes: the width of the cards, originally 3, becomes a variable ω, the offset at which cards need to hold, originally 1, becomes a variable o, and the list of final subsymbols is generalised to a list of final substrings. Abstractly, o symbols grouped together form a unit and covering cards need to hold only at positions which are multiples of o. Every 3-CC instance is a CC instance by an easy transformation.

Encoding Configurations. For the rest of this section, we fix a instance (Σ, M, in, k', t) of TMGenNP. The maximum length the fixed input in and the certificate can have is bounded by $k := |in| + k'$. We use the metavariables $\sigma : \Sigma$ and $m : \Sigma + \{\sqcup\}$. First, we consider how to *deterministically* simulate the machine on a given initial input and later deal with "guessing" a certificate.

A configuration consists of the current TM state, the tape contents, and the position of the single head. A configuration string $s : \Gamma^*$ over an extended alphabet Γ has a fixed width and encodes a configuration in an essentially unique way. Our encoding features the current state q and the symbol under the head m combined in a single symbol q^m *at the exact center* of the string. The tape halves $u : \Sigma^*$ to the left and to the right of the head are represented by the substrings $h : \Gamma^*$ left and right of the center symbol. Fixing the state symbol to the center simplifies the inductive invariants in the presence of two-sided infinite tape machines, but necessitates that, when the head is moved, the whole tape representation in the string is shifted.

Figure 3 shows an example of a configuration change, where the head is moved to the left and thus the tape needs to be shifted to the right. The outer-most character on both sides is the delimiter $\#$. We denote the amount of space maximally needed by the TM by

			$\longleftarrow z \longrightarrow$							$\longleftarrow z \longrightarrow$				
$\#$	$\sqcup$	$\sqcup$	$\cdots$	$\sqcup$	c	q_0^a	b	a	$\sqcup$	$\sqcup$	$\cdots$	$\sqcup$	$\sqcup$	$\#$
$\#$	$\overrightarrow{\sqcup}$	$\overrightarrow{\sqcup}$	$\cdots$	$\overrightarrow{\sqcup}$	$\overrightarrow{\sqcup}$	q_1^c	$\overrightarrow{b}$	$\overrightarrow{b}$	$\overrightarrow{a}$	$\overrightarrow{\sqcup}$	$\cdots$	$\overrightarrow{\sqcup}$	$\overrightarrow{\sqcup}$	$\#$

Figure 3 Two successive configuration strings, where $\Sigma = \{a, b, c\}$, q_0 is the initial state, and $\delta(q_0, \lfloor a \rfloor) = (q_1, \lfloor b \rfloor, \mathsf{L})$. The strings are delimited by $\#$.

$z := k + t$[7]. The tape symbols of the successor string are annotated with polarities p. They denote the direction in which the tape is shifted (left $\overleftarrow{m}$, stay $\overline{m}$, and right $\overrightarrow{m}$) and are added to every tape symbol. Polarities are needed in order to enforce a consistent tape shift across the whole tape string. A symbol with unknown polarity p is written as m^p. We omit the polarity of a symbol when it is irrelevant.

We define two relations $\sim_t$ and $\sim_c$ to capture the representation of tape halves and configurations. The relation $\sim_t$ is parameterised by the amount of space n available for the TM simulation and the polarity p. The empty tape half of length n is represented by $E\ p\ n$, defined as $E\ p\ 0 := [\#]$ and $E\ p\ Sn := \sqcup^p :: E\ p\ n$. A general tape half u is represented by u with an arbitrary polarity p added and trailing blank space, denoted $u \sim_t^{(n,p)} h$ (with two blanks added as noted above):

$$u \sim_t^{(n,p)} h := |u| \leq n \land h = [\, x^p \mid x \in u \,] +\!\!+\, E\ p\ (2 + n - |u|).$$

A configuration string is pieced together from three parts, where the left tape string must be reversed. The polarity is irrelevant, as long as it is consistent:

$$(q, tp) \sim_c s := \exists p, l, r. s = \mathsf{rev}\ (l) +\!\!+ [q^{\mathsf{current}\ tp}] +\!\!+ r \land \mathsf{left}\ tp \sim_t^{(z,p)} l \land \mathsf{right}\ tp \sim_t^{(z,p)} r$$

This relation determines the string representing a configuration uniquely, up to the polarity.

Simulation Cards. As a reminder, the covering cards used for the simulation encode valid configuration changes through overlaps. We distinguish three types of covering cards that encode valid configuration changes: cards that are used for shifting the tape, cards that affect the center state symbol, and cards that replicate a configuration string in case the machine halts in less than t steps. As most cards need to be applicable independently of the current tape contents or the polarity, we give *rules* for creating cards, containing metavariables σ_i and m_i. Moreover, for their premises we do not give the polarities. In order to obtain the actual cards, the rules are instantiated with all possible assignments to the metavariables. For instance, for transitions, three kinds of rules are needed, having the center state symbol in the left, middle, or right position. The middle position rules for $\delta(q_1, \lfloor \sigma_1 \rfloor) = (q_2, \lfloor \sigma_2 \rfloor, \mathsf{L})$ are:

$$\begin{array}{c|c|c} \sigma_3 & q_1^{\sigma_1} & m_1 \\ \hline \overrightarrow{m_2} & q_2^{\sigma_3} & \overrightarrow{\sigma_2} \end{array} \qquad \begin{array}{c|c|c} \sqcup & q_1^{\sigma_1} & m_1 \\ \hline \overrightarrow{\sqcup} & q_2^{\sqcup} & \overrightarrow{\sigma_2} \end{array}$$

One can see that a large number of rules is needed. The full set of rules can be found in the Coq development. We denote the set of covering cards obtained by instantiating the rules by R_{sim}.

[7] but we add two further blanks to each side as having at least three symbols in each tape string simplifies the proofs.

The next theorem is the main result needed to prove that TM runs are in one-to-one correspondence with CC-step sequences.

Theorem 14 (Step Simulation). *If $(q, tp) \sim_c s$, $(q, tp) \succ (q', tp')$ and $|tp| < z$, then there exists a unique s' such that $s \leadsto_{R_{sim}} s'$. Moreover, s' satisfies $(q', tp') \sim_c s'$.*

Proof. Given a configuration string, we split it into the state symbol q^m, the left tape half, and the right tape half.

The state symbol uniquely determines the transition the TM takes. By analysing the heads h_l and h_r of the tape halves, we can find out the cards to use at the center. These cards then uniquely determine the heads h'_l and h'_r of the successor tape halves. By an induction, the successor tape halves are therefore fully determined. The coverings can then be justified separately. ◀

A similar correspondence can be proved for halting configurations. As consequences, we get soundness and completeness for the simulation of deterministic TMs. Both proofs are by induction with some intermediate lemmas. The predicate haltingString s denotes that s contains a symbol q^m and halt q = true.

Theorem 15 (Completeness). *If $|tp| \leq k$, $(q, tp) \sim_c s$, and $(q, tp) \rhd^{\leq t} (q', tp')$, then there is s' with $s \leadsto^t s'$, $(q', tp') \sim_c s'$ and haltingString s'.*

Theorem 16 (Soundness). *If $(q, tp) \sim_c s$, $|tp| \leq k$, $s \leadsto^t s'$, and haltingString s', then there are q', tp' with $(q', tp') \sim_c s'$, $(q, tp) \rhd^{\leq t} (q', tp')$ and $|tp'| \leq z$.*

Nondeterministic Certificate. Finally, we need to generate an initial string that accounts for the certificate input of the TM. Since the 3-CC instance should be satisfiable iff there exists a valid certificate, this needs to be encoded using a CC-step. We prefix another line to the tableau and add additional cards that allow for a non-deterministic successor string that is used as the initial string of the TM simulation. By making these cards use a disjoint alphabet, they cannot interfere with the succeeding TM simulation.

Theorem 17. TMGenNP *reduces to* CC *in polynomial time.*

6.2 From 3-CC to SAT

For lack of space and since the constructions are conceptually simple, we do not give a more detailed account of the remaining steps from 3-CC to SAT but instead refer to Section 4 and our mechanisation. In total, we show the following reductions:

▶ **Theorem 18.**
1. CC *reduces to* BCC *in polynomial-time.*
2. BCC *reduces to* FSAT *in polynomial time.*
3. FSAT *reduces to* SAT *in polynomial time.*

6.3 NP Containment

For proving SAT NP-complete, we also need to show that SAT ∈ NP, which is far simpler than the chain of reductions. We construct a standard certificate verifier that takes assignments as certificates and verify it to run in polynomial-time. Further details are left to the Coq formalisation.

Lemma 19 (SAT is in NP)**.** SAT ∈ NP

In total, we now have proved the Cook-Levin theorem:

Theorem 20 (Cook-Levin)**.** SAT *is* NP*-complete*[8]*.*

Table 1 Line counts (grouped, counted with `coqwc`).

Component		Spec	Proof
Libraries $_{\text{definitions and frameworks for L and TM, ...}}$		17000+	17000+
Complexity Definitions		317	675
Problem Definitions	Normal Versions	444	541
	(Add.) Flat Versions	325	674
GenNP is NP-hard		43	230
GenNP to mTMGenNP$_{\text{excl. Eval}_\mathsf{L}}$		203	574
mTMGenNP to TMGenNP$_{\text{excl. multi-to-single-tape compiler}}$		264	889
TMGenNP to CC		3470	4442
CC to SAT		755	2312
SAT ∈ NP		108	297

7 Mechanisation in Coq

Our development compiles with Coq 8.12.1, without any axioms and with a code size as denoted in Table 1. We comment on a few challenges involved in the mechanisation.

Flat First-Order Encodings. One of the limitations of the extraction framework of [10] is that higher-order types and types having propositional components cannot be extracted directly. For instance, arbitrary finite types, dependent function types, and dependent pairs are hard to handle generically. We call a particular formulation of a problem $P : X \to \mathbb{P}$ *flat* if X is L-encodable and will also refer to the type of instances X as *flat*.

Directly formulating all problems discussed in this paper such that their instances are flat is unpleasant, as that strips away many of the amenities of Coq's dependently-typed language which are useful for stating problems. This does in particular affect the generic problems for TMs and the different variants of CC. Instead, we first formulate a problem P without paying attention to flatness and use this nice formulation to prove correctness statements. For the extraction, a separate flat version P' is defined, for instance by representing a finite type Σ by the number of its elements $|\Sigma|$ and its elements by the natural numbers up to $|\Sigma|$. The functions computing the reduction are defined on the flat versions. To prove their correctness, a natural notion of agreement to the non-flat definitions is proved.

For the variants of CC, this is straightforward. A flat definition of TMs was already available in the Coq Library of Undecidable Problems [14].

Encoding Turing machines in CC. A major difficulty in the reduction to CC is the handling of the covering cards. A large number of rules are needed to generate the cards and to handle all cases. This is in significant parts due to the fact that blank symbols ␣ are unknown to

[8] Actually, we mechanise that even 3-SAT, where clauses are restricted to have size 3, is NP-complete.

the TMs and have to be handled separately. Theorem 14 splits up into a large number of cases (100 are non-contradictory) due to cases analyses on the machine's behaviour and the local shape of the tape around the head (for selecting the right cards at the center). This is only feasible using custom Ltac automation (around 160LOC for this theorem), since the different cases are similar enough that the general strategy is the same, but different enough that at certain points, key decisions need to be taken depending on the current case and context. Further custom automation is used to invert the relations $\sim_t, \sim_c$ and to eliminate contradictions.

For easing automation, the cards are not directly formalised using lists, but instead using indexed inductive predicates. This enables an easy use of `eauto` to justify coverings and inversions to analyse coverings.

To make an automatic extraction into L and a resource analysis possible *with the extraction framework* [10], separate flat versions FlatCC of CC and FlatTMGenNP of TMGenNP are used, representing the finite type for the alphabet by natural numbers. In particular, a list of *flat* cards needs to be computed. In order to transfer the proof of correctness from CC to FlatCC, functions for computing the cards are parameterised and can be instantiated either with a finite type or with natural numbers. The proof then first shows that the cards determined by the indexed inductive predicates and the list-based cards using finite types agree and in a second step that both instantiations of the list-based cards are suitably equivalent.

Prop vs Type. Definitions like polynomial-time computability contain several components that are explicitly used in the constructions that we perform, like the explicit running time or the function bounding the result size. Therefore, our proofs get more concise by modelling definitions not as propositions, but in Type, which allows to define the projections.

7.1 Resource Analyses

A major part of complexity theory is making sure that the constructions satisfy the required resource bounds. While this aspect is uninteresting from a formalisation perspective, the mechanisation of resource bounds is work-intensive and benefits from the right abstractions. We need to derive not only running time bounds for L computations, but also for their result size, and similar bounds for the TMs constructed in Section 5.

We define a preorder $\leq_c$ on functions, used to give more readable upper bounds.

$$f \leq_c g := \Sigma c'. \forall x. f\ x \leq c' \cdot g\ x$$

This definition has many of the desired properties of O notation, like abstraction from constant factors and the ability to bound a (finite) sum by the largest summands. But in contrast to "full" O notation, which was mechanised and used by Armaël et al. [17], our definition is lightweight and covers functions with multiple (uncurried) arguments of arbitrary type, without the need of a notion of limits or ultrafilters.

We use two notational tricks in Coq: Using coercions and a record we can concisely express the existence of some function $\leq_c g$ in specifications. Using the projection to the c in the definition of $\leq_c$, we can employ the concrete bound $c \cdot g\ x$ whenever needed and still hide the exact value of c.

Intuitive Time Bounds for L. Since L is a low-level machine model, we inevitably need to connect our analyses to the notions of time for L. Formally, the running time function simply is a function that computes the number of steps needed for each input. A priori, the syntactic representation of this function can be very messy (e.g. recursive), mention many constants whose exact value is of no interest, and is not expressed in terms of the size of the input, but the input itself. Deriving concise, intuitive bounds is an important aspect of our mechanisation. As an example, we consider deriving time bounds for the evaluation of Boolean clauses (conjunctions of disjunctions of literals) used for the SAT verifier of Section 6.3. We represent variables by natural numbers, literals as pairs of variables and their Boolean sign, and clauses as lists of literals. Assignments are lists containing the variables which are assigned true, all other variables are implicitly false.

$$v : \mathsf{var} := \mathbb{N} \qquad l : \mathsf{lit} := \mathbb{B} \times \mathsf{var} \qquad C : \mathsf{cla} := \mathsf{lit}^* \qquad a : \mathsf{assgn} := \mathsf{var}^*$$

$$\begin{aligned} \mathsf{existsb}\ p\ [\,] &:= \mathsf{false} & \qquad \mathcal{E}_{\mathsf{lit}}\ a\ (s, v) &:= (v \overset{?}{\in} a) \overset{?}{=} s \\ \mathsf{existsb}\ p\ (x :: l) &:= p(x) \mid \mathsf{existsb}\ p\ l & \mathcal{E}_{\mathsf{cla}}\ a\ C &:= \mathsf{existsb}\ (\mathcal{E}_{\mathsf{lit}}\ a)\ C \end{aligned}$$

Let us assume that we have already analysed $\mathcal{E}_{\mathsf{lit}}$. For the time analysis of $\mathcal{E}_{\mathsf{cla}}$, we first analyse the higher-order function existsb used in its definition. We start with the (simplified) recurrences generated by the certifying extraction framework of Forster et al. [10], which are phrased in terms of the number of β-steps of the extracted L-term:

$$\begin{aligned} 8 &\le T_{\mathsf{existsb}}(T_f, []) \\ T_f\ h + T_{\mathsf{existsb}}(T_f, l) + 15 &\le T_{\mathsf{existsb}}(T_f, h :: l) \end{aligned}$$

where C is an L-encodable type and $T_f : X \to \mathbb{N}$ is the running time function of the argument function $f : X \to \mathbb{B}$. Instead of solving this recurrence explicitly for T_{existsb}, involving all the constants, we strive for more intuitive bounds in terms of abstract size functions, such as the length of the list. Therefore, we give a bounding solution $T_{\mathsf{existsb}} \le_c B_{\mathsf{existsb}}$ for $B_{\mathsf{existsb}} := \lambda(T_f, C).\ \sum_{a \in C} T_f a + |C| + 1$ which hides the constants.

With this, we derive bounds for $\mathcal{E}_{\mathsf{cla}}$. We make use of the abstract size function maxVar giving the (encoding size of the) maximum variable used in a clause. Assuming that $T_{\mathcal{E}_{\mathsf{lit}}} \le_c B_{\mathcal{E}_{\mathsf{lit}}} := \lambda a.(|a| + 1) \cdot (\mathsf{maxVar}(a) + 1)$, we compositionally obtain the bound $T_{\mathcal{E}_{\mathsf{cla}}} \le_c B_{\mathcal{E}_{\mathsf{cla}}} := \lambda(a, C).(|C| + 1) \cdot (|a| + 1) \cdot (\mathsf{maxVar}(a) + 1)$ by instantiating the bound for existsb.

To get back to the concrete resource measures of L in the end and obtain bounds in terms of the encoding size, we simply have to prove that the abstract size functions are bounded in terms of the encoding size, e.g. that $|a| \le \|\ulcorner a \urcorner\|$.

8 Discussion

To the best of our knowledge, this paper contains the first mechanisation of a complexity-theoretic result all the way down to a computational model. Our experiences seem to support the conjecture [12] that a λ-calculus is more amenable to mechanising complexity theory than TMs are. This is in large parts due to being able to automatically generate L-code from Coq-code, which makes programming and verifying in L simple, and having recurrences generated in the same process. Still, the resource analysis poses a major overhead over just verifying the functional correctness of reductions as is done in synthetic computability theory [9], since we need to derive flat first-order encodings of the problems and of course analyse the running time of the reduction functions.

It may seem ironic that we started out with a quest to move away from TMs, but ended up still using them as an intermediate problem in the reduction chain because of their local computational structure, and because of our experience with mechanising TMs [12, 14]. But the techniques we have for poly-time computability proofs in L are vastly superior to the hypothetical alternative of using TMs as underlying model of computation and verifying TM-computability of all constructions.

Future work. This work, the NP-hardness of SAT, now enables a variety of fully mechanised NP-completeness proofs, like the well known 21 NP-complete problems by Karp [20], without even knowing what a Turing machine is.

We conjecture that many interesting results in computational complexity theory can be mechanised when phrased in terms of L, as evident by a mechanisation of the time hierarchy theorem we are working on. We want to explore these possibilities in more detail.

Mechanising the agreement between our definition of NP and a TM-definition of NP should be straightforward by reusing simulations in L/TMs we already have.

We are interested in understanding how characterisations of P and NP that are independent from a computational model, like via Fagin's theorem [8] or certain type systems [18], can be shown equivalent to our characterisations.

References

1 Andrea Asperti. A formal proof of borodin-trakhtenbrot's gap theorem. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs*, pages 163–177, Cham, 2013. Springer International Publishing.

2 Andrea Asperti. Reverse complexity. *Journal of Automated Reasoning*, 55:373–388, 2015.

3 Andrea Asperti and Wilmer Ricciotti. A formalization of multi-tape Turing machines. *Theoretical Computer Science*, 603:23–42, 2015.

4 Andrej Bauer. First steps in synthetic computability theory. *Electron. Notes Theor. Comput. Sci.*, 155:5–31, 2006. doi:10.1016/j.entcs.2005.11.049.

5 Guy E. Blelloch and John Greiner. Parallelism in Sequential Functional Languages. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*, pages 226–237, 1995. doi:10.1145/224164.224210.

6 Mario M. 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, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*, pages 12:1–12:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ITP.2019.12.

7 Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery. doi:10.1145/800157.805047.

8 Ronald Fagin. Generalized first-order spectra, and polynomial. time recognizable sets. *SIAM-AMS Proc.*, 7, January 1974.

9 Yannick Forster, Dominik Kirst, and Gert Smolka. On synthetic undecidability in Coq, with an application to the Entscheidungsproblem. In *8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, New York, NY, USA, January 2019. ACM.

10 Yannick Forster and Fabian Kunze. A Certifying Extraction with Time Bounds from Coq to Call-By-Value Lambda Calculus. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:19, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

11 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.

12 Yannick Forster, Fabian Kunze, and Maximilian Wuttke. Verified programming of Turing machines in Coq. In *9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20–21, 2020*, New York, NY, USA, 2020. ACM.

13 Yannick Forster, Fabian Kunze, 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.

14 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 *The Sixth International Workshop on Coq for Programming Languages (CoqPL 2020)*, 2020. URL: https://github.com/uds-psl/coq-library-undecidability.

15 Yannick Forster and Gert Smolka. Weak Call-by-Value Lambda Calculus as a Model of Computation in Coq. In *International Conference on Interactive Theorem Proving*, pages 189–206. Springer, 2017.

16 Ruben Gamboa and John Cowles. A mechanical proof of the Cook-Levin theorem. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics*, pages 99–116, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

17 Armaël Guéneau, Arthur Charguéraud, and François Pottier. A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 533–560, Cham, 2018. Springer International Publishing.

18 Martin Hofmann. Linear types and non size-increasing polynomial time computation. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, LICS '99, page 464, USA, 1999. IEEE Computer Society.

19 Jan Martin Jansen. *Programming in the λ-Calculus: From Church to Scott and Back*, pages 168–180. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-40355-2_12.

20 Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972. doi:10.1007/978-1-4684-2001-2_9.

21 L. A. Levin. Universal sequential search problems. *Problems Inform. Transmission*, 9:265–266, 1973. [Probl. Peredachi Inf., **9**,3:115-116, 1973].

22 Torben Æ. Mogensen. Efficient self-interpretation in lambda calculus. *JOURNAL OF FUNCTIONAL PROGRAMMING*, 2:345–364, 1994.

23 Michael Norrish. Mechanised computability theory. In Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving*, pages 297–311, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

24 Christos H. Papadimitriou. *Computational Complexity*, page 260–265. John Wiley and Sons Ltd., GBR, 2003.

25 Michael Sipser. *Introduction to the Theory of Computation.* International Thomson Publishing, 1st edition, 1996.

26 C. Slot and P. van Emde Boas. On tape versus core an application of space efficient perfect hash functions to the invariance of space. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, STOC '84, page 391–400, New York, NY, USA, 1984. Association for Computing Machinery. doi:10.1145/800057.808705.

27 G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983. doi:10.1007/978-3-642-81955-1_28.

Proving Quantum Programs Correct

Kesha Hietala
University of Maryland, College Park, MD, USA

Robert Rand
University of Chicago, IL, USA

Shih-Han Hung
University of Maryland, College Park, MD, USA

Liyi Li
University of Maryland, College Park, MD, USA

Michael Hicks
University of Maryland, College Park, MD, USA

Abstract

As quantum computing progresses steadily from theory into practice, programmers will face a common problem: How can they be sure that their code does what they intend it to do? This paper presents encouraging results in the application of mechanized proof to the domain of quantum programming in the context of the SQIR development. It verifies the correctness of a range of a quantum algorithms including Grover's algorithm and quantum phase estimation, a key component of Shor's algorithm. In doing so, it aims to highlight both the successes and challenges of formal verification in the quantum context and motivate the theorem proving community to target quantum computing as an application domain.

2012 ACM Subject Classification Hardware → Quantum computation; Software and its engineering → Formal software verification

Keywords and phrases Formal Verification, Quantum Computing, Proof Engineering

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.21

Related Version *Extended Version*: `https://arxiv.org/abs/2010.01240`

Supplementary Material *Software*: `https://github.com/inQWIRE/SQIR`
archived at `swh:1:dir:2aed711638324c36b815c91d1f8b9ad7ca46a300`

Funding This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Quantum Testbed Pathfinder Program under Award Number DE-SC0019040 and the Air Force Office of Scientific Research under Grant No. FA95502110051.

Acknowledgements We thank Yuxiang Peng for ongoing contributions to the SQIR codebase and pointing out a bug in our original specification for QPE. We thank Xiaodi Wu for discussions about SQIR and follow-on projects.

1 Introduction

Quantum computers are fundamentally different from the "classical" computers we have been programming since the development of the ENIAC in 1945. This difference includes a layer of complexity introduced by quantum mechanics: Instead of a deterministic function from inputs to outputs, a quantum program is a function from inputs to a *superposition* of outputs, a notion that generalizes probabilities. As a result, quantum programs are strictly more expressive than probabilistic programs and even harder to get right. While we can test the output of a probabilistic program by comparing its observed distribution to the desired one, doing the same on a quantum computer can be prohibitively expensive and may not fully describe the underlying quantum state.

This challenge for quantum programming is an opportunity for formal methods. We can use formal methods to *prove*, in advance, that the code implementing a quantum algorithm does what it should for all possible inputs and configurations.

In prior work [17], we developed a formally verified optimizer for quantum programs (VOQC), implemented and proved correct in the Coq proof assistant [8]. VOQC transforms programs written in SQIR, a *small quantum intermediate representation*. While we designed SQIR to be a compiler intermediate representation, we quickly realized that it was not so different from languages used to write *source* quantum programs, and that the design choices that eased proving optimizations correct could ease proving source programs correct, too.

To date, we have proved the correctness of implementations of a number of quantum algorithms, including quantum teleportation, Greenberger–Horne–Zeilinger (GHZ) state preparation [15], the Deutsch-Jozsa algorithm [10], Simon's algorithm [32], the quantum Fourier transform (QFT), quantum phase estimation (QPE), and Grover's algorithm [16]. QPE is a key component of Shor's prime-factoring algorithm [31], today's best-known, most impactful quantum algorithm, with Grover's algorithm for unstructured search being the second. Our implementations can be extracted to code that can be executed on quantum hardware or simulated classically, depending on the problem size and hardware limitations.

While SQIR was first introduced as part of VOQC, this paper offers two new contributions. First, it presents a detailed discussion of how SQIR's design supports proofs of correctness. After presenting background on quantum computing (Section 2) and reviewing SQIR (Section 3), Section 4 discusses key elements of SQIR's design and compares and contrasts them to design decisions made in the related tools $\mathcal{Q}$WIRE [25], $\mathcal{Q}$BRICKS [7], and the Isabelle implementation of quantum Hoare logic [18]. SQIR's overall benefit over these tools is its flexibility, supporting multiple semantics and approaches to proof. As a second contribution, this paper presents the code, formal specification, and proof sketch of Grover's algorithm, QFT, and QPE, which are the most sophisticated algorithms that we have verified so far (Section 5). We comment on the proofs of simpler algorithms in Appendix B of the extended version of this paper. We believe there is ripe opportunity for further application of formal methods to quantum computing and we hope this paper, and our work on SQIR, paves the way for new research; we sketch open problems in Section 6.

SQIR is implemented in just over 3500 lines of Coq, with an additional 3700 lines of example SQIR programs and proofs; it is freely available on Github.[1]

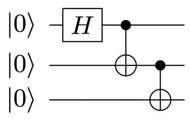
2 Background

We begin with a light background on quantum computing; for a full treatment we recommend the standard text on the subject [22].

2.1 Quantum States

A quantum state consists of one or more *quantum bits*. A quantum bit (or *qubit*) can be expressed as a two dimensional vector $\left(\begin{smallmatrix}\alpha\\ \beta\end{smallmatrix}\right)$ such that $|\alpha|^2 + |\beta|^2 = 1$. The α and β are called *amplitudes*. We frequently write this vector as $\alpha\,|0\rangle + \beta\,|1\rangle$ where $|0\rangle = \left(\begin{smallmatrix}1\\ 0\end{smallmatrix}\right)$ and $|1\rangle = \left(\begin{smallmatrix}0\\ 1\end{smallmatrix}\right)$ are *basis states*. When both α and β are non-zero, we can think of the qubit as being "both 0 and 1 at once," a.k.a. a *superposition*. For example, $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ is an equal superposition of $|0\rangle$ and $|1\rangle$.

[1] https://github.com/inQWIRE/SQIR



(a) Quantum circuit

```
H 0;
CNOT 0 1;
CNOT 1 2
```

(b) SQIR assembly

```
Fixpoint ghz (n : ℕ) : ucom base n :=
  match n with
  | 0 ⇒ I 0
  | 1 ⇒ H 0
  | S n' ⇒ ghz n'; CNOT (n'-1) n'
  end.
```

(c) SQIR meta-program

```
box (x,y,z) ⇒
  gate x     ← H x;
  gate (x,y) ← CNOT (x,y);
  gate (y,z) ← CNOT (y,z);
  output (x,y,z).
```

(d) QWIRE

```
SEQ(SEQ(PAR(H, PAR(I, I)),
        PAR(CNOT, I)),
    PAR(I, CNOT))
```

(e) QBRICKS-DSL

```
q1    := H q1;
q1,q2 := CNOT q1,q2;
q2,q3 := CNOT q2,q3
```

(f) QWhile

$$(I \otimes CNOT) \times (CNOT \otimes I) \times (H \otimes I \otimes I)$$

(g) Matrix expression

Figure 1 Example quantum program: GHZ state preparation.

We can join multiple qubits together by means of the *tensor product* ($\otimes$) from linear algebra. For convenience, we write $|i\rangle \otimes |j\rangle$ as $|ij\rangle$ for $i, j \in \{0, 1\}$; we may also write $|k\rangle$ where $k \in \mathbb{N}$ is the decimal interpretation of bits ij. We use $|\psi\rangle$ to refer to an arbitrary quantum state. Sometimes a multi-qubit state cannot be expressed as the tensor of individual qubits; such states are called *entangled*. One example is the state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, known as a *Bell pair*.

2.2 Quantum Programs

Quantum programs are composed of a series of *quantum operations*, each of which acts on a subset of qubits in the quantum state. In the standard presentation, quantum programs are expressed as *circuits*, as shown in Figure 1(a). In these circuits, each horizontal wire represents a qubit and boxes on these wires indicate quantum operations, or *gates*. The circuit in Figure 1(a) uses three qubits and applies three gates: the *Hadamard* (`H`) gate and two *controlled-not* (`CNOT`) gates. The semantics of a gate is a *unitary matrix* (a matrix that preserves the unitarity invariant of quantum states); applying a gate to a state is tantamount to multiplying the state vector by the gate's matrix. The matrix corresponding to the circuit in Figure 1(a) is shown in Figure 1(g), where I is the 2×2 identity matrix, $CNOT$ is the matrix corresponding to the `CNOT` gate, and H is the matrix corresponding to the `H` gate.

A special, non-unitary *measurement* operation is used to extract classical information from a quantum state (often, when a computation completes). Measurement collapses the state to one of the basis states with a probability related to the state's amplitudes. For example, measuring $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ will collapse the state to $|0\rangle$ with probability $\frac{1}{2}$ and likewise for $|1\rangle$, returning classical values 0 or 1, respectively. The semantics of a program involving measurement amounts to a probability distribution over quantum states; such a distribution is called a *mixed state*. In our example above, measurement produces a mixed state that is a uniform distribution over $|0\rangle$ and $|1\rangle$. By contrast, *pure states* like $|0\rangle$ and $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ can be produced without measurement. Section 3.3 discusses non-unitary semantics further.

3 SQIR: A Small Quantum Intermediate Representation

SQIR is a simple quantum language deeply embedded in the Coq proof assistant. This section presents SQIR's syntax and semantics. We defer a detailed discussion of SQIR's design rationale to the next section.

3.1 Unitary SQIR: Syntax

SQIR's *unitary fragment* is a sub-language of full SQIR for expressing programs consisting of unitary gates. (The full SQIR language extends unitary SQIR with measurement.) A program in the unitary fragment has type `ucom` (for "unitary command"), which we define in Coq as follows:

```
Inductive ucom (U: ℕ → Set) (d : ℕ) : Set :=
| useq  : ucom U d → ucom U d → ucom U d
| uapp1 : U 1 → ℕ → ucom U d
| uapp2 : U 2 → ℕ → ℕ → ucom U d
```

The `useq` constructor sequences two commands; we use notational shorthand `p1` ; `p2` for `useq p1 p2`. The two `uappn` constructors indicate the application of a quantum gate to n qubits, where n is 1 or 2. Qubits are identified as numbered indices into a *global qubit register* of size d, which stores the quantum state. Gates are drawn from parameter `U`, which is indexed by a gate's size. For writing and verifying programs, we use the following `base` set for `U`, inspired by IBM's OpenQASM [9]:[2]

```
Inductive base : ℕ → Set :=
  | U_R (θ ϕ λ : R) : base 1
  | U_CNOT          : base 2.
```

That is, we have a one-qubit gate `U_R` (which we write U_R when using math notation), which takes three real-valued arguments, and the standard two-qubit *controlled-not* gate, `U_CNOT` (written $CNOT$ in math notation), which negates the second qubit wherever the first qubit is $|1\rangle$, making it the quantum equivalent of a *xor* gate. The `U_R` gate can be used to express any single-qubit gate (see Section 3.2). Together, `U_R` and `U_CNOT` form a *universal* gate set, meaning that they can be composed to describe any unitary operation [3].

Example: SWAP

The following Coq function produces a unitary SQIR program that applies three controlled-not gates in a row, with the effect of exchanging two qubits in the register. We define `CNOT` as shorthand for `uapp2 U_CNOT`.

```
Definition SWAP d a b : ucom base d := CNOT a b; CNOT b a; CNOT a b.
```

[2] It is helpful for proofs to keep `U` small because the number of cases in the proof about a value of type `ucom U d` will depend on the number of gates in `U`. In our work on VOQC [17], we define optimizations over a larger gate set that includes common gates like Hadamard, but convert these gates to our `base` set for proof.

Example: GHZ

Figure 1(b) is the SQIR representation of the circuit in Figure 1(a), which prepares the three-qubit GHZ state [15]. We describe *families* of SQIR circuits by meta-programming in the Coq host language. The Coq function in Figure 1(c) produces a SQIR program that prepares the n-qubit GHZ state, producing the program in Figure 1(b) when given input 3. In Figures 1(b–c), `H` and `I` apply the `U_R` encodings of the Hadamard and identity gates.

3.2 Unitary SQIR: Semantics

Each k-qubit quantum gate corresponds to a $2^k \times 2^k$ unitary matrix. The matrices for our `base` set are:

$$[\![U_R(\theta,\phi,\lambda)]\!] = \begin{pmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2) \\ e^{i\phi}\sin(\theta/2) & e^{i(\phi+\lambda)}\cos(\theta/2) \end{pmatrix}, \qquad [\![CNOT]\!] = \begin{pmatrix} 1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0 \end{pmatrix}.$$

Conveniently, the U_R gate can encode any single-qubit gate [22, Chapter 4]. For instance, two commonly-used single-qubit gates are X ("not") and H ("Hadamard"). The former has the matrix $\left(\begin{smallmatrix}0&1\\1&0\end{smallmatrix}\right)$ and serves to flip a qubit's α and β amplitudes; it can be encoded as $U_R(\pi,0,\pi)$. The H gate has the matrix $\frac{1}{\sqrt{2}}\left(\begin{smallmatrix}1&1\\1&-1\end{smallmatrix}\right)$, and is often used to put a qubit into superposition (it takes $|0\rangle$ to $\frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)$); it can be encoded as $U_R(\pi/2,0,\pi)$. Multi-qubit gates are easily produced by combinations of $CNOT$ and U_R; we show the definition of the three-qubit "Toffoli" gate in Section 4.6. Keeping our gate set small simplifies the language and enables easy case analysis – and does not complicate proofs. We rarely unfold the definition of gates like X or the three-qubit Toffoli, instead providing automation to directly translate these gates to their intended denotations. Hence, X is translated directly to $\left(\begin{smallmatrix}0&1\\1&0\end{smallmatrix}\right)$. Users can thereby easily extend SQIR with new gates and denotations.

A unitary SQIR program operating on a size-d register corresponds to a $2^d \times 2^d$ unitary matrix. Function `uc_eval` denotes the matrix corresponding to program `c`.

```
Fixpoint uc_eval {d} (c : ucom base d) : Matrix (2^d) (2^d) := ...
```

We write $[\![\texttt{c}]\!]_d$ for `uc_eval d c`. The denotation of composition is simple matrix multiplication: $[\![\texttt{U1; U2}]\!]_d = [\![\texttt{U2}]\!]_d \times [\![\texttt{U1}]\!]_d$. The denotation of `uapp1` is the denotation of its argument gate, but padded with the identity matrix so it has size $2^d \times 2^d$. To be precise, we have:

$$[\![\texttt{uapp1 U q}]\!]_d = \begin{cases} I_{2^q} \otimes [\![U]\!] \otimes I_{2^{d-q-1}} & q < d \\ 0_{2^d} & \text{otherwise} \end{cases}$$

where I_n is the $n \times n$ identity matrix. In the case of our `base` gate set, $[\![U]\!]$ is the U_R matrix shown above. The denotation of any gate applied to an out-of-bounds qubit is the zero matrix, ensuring that a circuit corresponds to a zero matrix if and only if it is ill-formed. We likewise prove that every well-formed circuit corresponds to a unitary matrix.

As our only two-qubit gate in the `base` set is `U_CNOT`, we specialize our semantics for `uapp2` to this gate. To compute $[\![\texttt{CNOT q1 q2}]\!]_d$, we first decompose the $CNOT$ matrix into $\left(\begin{smallmatrix}1&0\\0&0\end{smallmatrix}\right) \otimes I_2 + \left(\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right) \otimes X$. We then pad the expression appropriately, obtaining the following when $q_1 < q_2 < d$:

$$I_{2^{q_1}} \otimes \left(\begin{smallmatrix}1&0\\0&0\end{smallmatrix}\right) \otimes I_{2^{q_2-q_1-1}} \otimes I_2 \otimes I_{2^{d-q_2-1}} \;+\; I_{2^{q_1}} \otimes \left(\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right) \otimes I_{2^{q_2-q_1-1}} \otimes X \otimes I_{2^{d-q_2-1}}.$$

When $q_2 < q_1 < d$, we obtain a symmetric expression, and when either qubit is out of bounds, we get the zero matrix. Additionally, since the two inputs to $CNOT$ cannot be the same, if $q_1 = q_2$ we also obtain the zero matrix.

Example: Verifying SWAP

We can prove in Coq that `SWAP` 2 0 1, which swaps the first and second qubits in a two-qubit register, behaves as expected on two unentangled qubits:

```
Lemma swap2: ∀ (ϕ ψ : Vector 2), WF_Matrix ϕ → WF_Matrix ψ →
  ⟦SWAP 2 0 1⟧₂ × (ϕ ⊗ ψ) = ψ ⊗ ϕ.
```

`WF_Matrix` says that ϕ and ψ are well-formed vectors [29, Section 2]. This proof can be completed by simple matrix multiplication. In the full development we prove the correctness of `SWAP d a b` for arbitrary dimension `d` and qubits `a` and `b`.

3.3 Full SQIR: Adding Measurement

The full SQIR language adds a branching measurement construct inspired by Selinger's QPL [30]. This construct permits measuring a qubit, taking one of two branches based on the measurement outcome. Full SQIR defines "commands" `com` as either a unitary sub-program, a no-op `skip`, branching measurement, or a sequence of these.

```
Inductive com (U: ℕ → Set) (d : ℕ) : Set :=
| uc   : ucom U d → com U d
| skip : com U d
| meas : ℕ → com U d → com U d → com U d
| seq  : com U d → com U d → com U d.
```

The command `meas q` P_1 P_2 measures qubit `q` and performs P_1 if the outcome is 1 and P_2 if it is 0. We define non-branching measurement and resetting to a zero state in terms of branching measurement:

```
Definition measure q := meas q skip skip.
Definition reset q := meas q (X q) skip.
```

As before, we use our `base` set of unitary gates for full SQIR.

Example: Flipping a Coin

It is simple to generate a random coin flip with a quantum computer: Use the Hadamard gate to put a qubit into equal superposition $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and then measure it.

```
Definition coin : com base 1 := H 0; measure 0.
```

Density Matrix Semantics

As discussed in Section 2.2, measurement induces a probabilistic transition, so the semantics of a program with measurement is a probability distribution over states, called a mixed state. As is standard [25, 34], we represent such a state using a *density matrix*. The density matrix of a pure state $|\psi\rangle$ is $|\psi\rangle\langle\psi|$ where $\langle\psi| = |\psi\rangle^\dagger$ is the conjugate transpose of $|\psi\rangle$. The density matrix of a mixed state is a sum over its constituent pure states. For example, the density matrix corresponding to the uniform distribution over $|0\rangle$ and $|1\rangle$ is $\frac{1}{2}|0\rangle\langle 0| + \frac{1}{2}|1\rangle\langle 1|$.

The semantics $\{\!|\texttt{P}|\!\}_d$ of a full SQIR program `P` is a function from density matrices to density matrices. Naturally, $\{\!|\texttt{skip}|\!\}_d\ \rho = \rho$ and $\{\!|\texttt{P1 ; P2}|\!\}_d = \{\!|\texttt{P2}|\!\}_d \circ \{\!|\texttt{P1}|\!\}_d$. For unitary subroutines, we have $\{\!|\texttt{uc U}|\!\}_d\ \rho\ = [\![\texttt{U}]\!]_d \rho [\![\texttt{U}]\!]_d{}^\dagger$: Applying a unitary matrix to a state vector is equivalent to applying it to both sides of its density matrix. Finally, using $|i\rangle_q\langle j|$ for $I_{2^q} \otimes |i\rangle\langle j| \otimes I_{2^{d-q-1}}$, the semantics for $\{\!|\texttt{meas q P1 P2}|\!\}_d\ \rho$ is

$$\{\!|P_1|\!\}_d(|1\rangle_q\langle 1|\,\rho\,|1\rangle_q\langle 1|) + \{\!|P_2|\!\}_d(|0\rangle_q\langle 0|\,\rho\,|0\rangle_q\langle 0|)$$

which corresponds to probabilistically applying `P1` to ρ with the specified qubit projected to $|1\rangle\langle 1|$ or applying `P2` to a similarly altered ρ.

Example: A Provably Random Coin

We can now prove that our `coin` circuit above produces the $|1\rangle\langle 1|$ or $|0\rangle\langle 0|$ density matrix (corresponding to the $|1\rangle$ or $|0\rangle$ pure state), each with probability $\frac{1}{2}$.

```
Lemma coin_dist : {|coin|}₁ |0⟩⟨0| = ½|0⟩⟨0| + ½|0⟩⟨0|.
```

The proof proceeds by simple matrix arithmetic. $\{\!|\texttt{H}|\!\}\ |0\rangle\langle 0|$ is $H\,|0\rangle\langle 0|\,H^\dagger = \frac{1}{2}\left(\begin{smallmatrix}1 & 1\\ 1 & 1\end{smallmatrix}\right)$. Calling this ρ_{12}, applying `measure` yields $|1\rangle\langle 1|\,\rho_{12}\,|1\rangle\langle 1| + |0\rangle\langle 0|\,\rho_{12}\,|0\rangle\langle 0|$, which can be further simplified using the fact $\langle 1|\,\rho_{12}\,|1\rangle = \langle 0|\,\rho_{12}\,|0\rangle = \left(\,\frac{1}{2}\,\right)$, yielding $\frac{1}{2}\,|1\rangle\langle 1| + \frac{1}{2}\,|0\rangle\langle 0|$ as desired.

Measurement plays a key role in many quantum algorithms; we discuss further examples and an alternative semantics in Appendix A of the extended version of this paper.

4 SQIR's Design

This section describes key elements in the design of SQIR and its infrastructure for verifying quantum programs. To place those decisions in context, we first introduce several related verification frameworks and contrast SQIR's design with theirs. In summary, SQIR benefits from the use of *concrete indices into a global register* (a common feature in the tools we looked at), support for *reasoning about unitary programs in isolation* (supported by one other tool), and the *flexibility to allow different semantics and approaches to proof* (best supported in SQIR).

4.1 Related Approaches

Several prior works have had the goal of formally verifying quantum programs. In 2010, Green [14] developed an Agda implementation of the Quantum IO Monad, and in 2015 Boender et al. [5] produced a small Coq quantum library for reasoning about quantum "programs" directly via their matrix semantics (e.g. see Figure 1(g)). These were both proofs of concept, and were only capable of verifying basic protocols. More recently, Bordg et al. [6] took a step further in verifying quantum programs expressed as matrix products (Figure 1(g)), providing a library for reasoning about quantum computation in Isabelle/HOL and verifying more interesting protocols like the n-qubit Deutsch-Jozsa algorithm (shown in SQIR in Appendix B of the extended version of this paper).

In this section, we compare SQIR's design against three other tools for verified quantum programming that have been used to verify interesting, parameterized quantum programs: $\mathcal{Q}$WIRE [27] (implemented in Coq [8]); quantum Hoare logic [19] (in Isabelle/HOL [23]); and $\mathcal{Q}$BRICKS [7] (in Why3 [11]). We do not include Bordg et al. [6], despite its recency, because it operates one level below the surface programming language, so many issues considered here do not apply. Bordg et al.'s library is similar to the quantum libraries developed for

$\mathcal{Q}$WIRE and the quantum Hoare logic. All matrix formalisms provided by Bordg et al. are also available in $\mathcal{Q}$WIRE's library, which we re-use and extend (by ~ 3000 lines [17, Section 2.2]) in SQIR.

QWIRE

The $\mathcal{Q}$WIRE language [25, 27] originated as an embedded circuit description language in the style of Quipper [13] but with a more powerful type system. Figure 1(d) shows the $\mathcal{Q}$WIRE equivalent of the SQIR program in Figure 1(b). $\mathcal{Q}$WIRE uses variables from the host language Coq to reference qubits, an instantiation of higher-order abstract syntax [26]. In Figure 1, the $\mathcal{Q}$WIRE program uses variables `x`, `y`, and `z`, while the SQIR program uses indices 0, 1, and 2 to refer to the first, second, and third qubits in the global register. $\mathcal{Q}$WIRE does not distinguish between unitary and non-unitary programs, and thus uses density matrices for its semantics. $\mathcal{Q}$WIRE has been used to verify simple randomness generation circuits and a few textbook examples [28].

QBRICKS

$\mathcal{Q}$BRICKS [7] is a quantum proof framework implemented in Why3 [11], developed concurrently with SQIR. $\mathcal{Q}$BRICKS provides a domain-specific language (DSL) for constructing quantum circuits using combinators for parallel and sequential composition (among others). Figure 1(e) presents the GHZ example written in $\mathcal{Q}$BRICKS' DSL. The semantics of $\mathcal{Q}$BRICKS are based on the *path-sums* formalism by Amy [1, 2], which can express the semantics of unitary programs in a form amenable to proof automation. $\mathcal{Q}$BRICKS extends path-sums to support parameterized circuits. $\mathcal{Q}$BRICKS has been used to verify a variety of quantum algorithms, including Grover's algorithm and Quantum Phase Estimation (QPE).

Quantum Hoare Logic

Quantum Hoare logic (QHL) [34] has been formalized in the Isabelle/HOL proof assistant [18]. QHL is built on top of the quantum while language (QWhile), which is the quantum analog of the classical while language, allowing looping and branching on measurement results. Figure 1(f) presents the GHZ example written in QHL. QWhile does not use a fixed gate set; gates are instead described directly by their unitary matrices. As such, the program in Figure 1(f) could instead be written as the application of a single gate that prepares the 3-qubit GHZ state. Given that measurement is a core part of the language, QWhile's semantics are given in terms of (partial) density matrices. A density matrix is *partial* when it may represent a sub-distribution – that is, a subset of the outcomes of measurement.

QHL has been used to verify Grover's algorithm [18]. An earlier effort by Liu et al. [20] to formalize QHL claimed to prove correctness of QPE, too. However, the approach used a combination of Isabelle/HOL and Python, calling out to Numpy to solve matrix (in)equalities; as such, we consider this only a partial verification effort. We cannot find a proof of QPE in the associated Github repository[3] and believe that this approach was abandoned in favor of Liu et al. [18].

[3] `https://github.com/ijcar2016/propitious-barnacle`

4.2 Concrete Indices into a Global Register

The first key element of SQIR's design is its use of concrete indices into a fixed-sized global register to refer to qubits. For example, in our `SWAP` program (end of Section 3.1), `a` and `b` are natural numbers indexing into a global register of size `d`. Expressing the semantics of a program that uses concrete indices is simple because concrete indices map directly to the appropriate rows and columns in the denoted matrix. Moreover, it is easy to check relationships between operations – `X a` and `X b` act on the same qubit if and only if `a` = `b`. Keeping the register size fixed means that the denoted matrix's size is known, too.

On the other hand, concrete indices hamper programmability. The `ghz` example in Figure 1(c) only produces circuits that occupy global qubits 0...`n`; we could imagine further generalizing it to add a lower bound `m` (so the circuit uses qubits `m` ... `n`), but it is not clear how it could be generalized to use non-contiguous wires. A natural solution, employed by $\mathcal{Q}$WIRE, is to use host-level variables to refer to *abstract* qubits that can be freely introduced and discarded, simplifying circuit construction and sub-program composition. Unfortunately, abstract qubits significantly complicate formal verification. To translate circuits to operations on density matrices, variables must be mapped to concrete matrix indices. Each time a qubit is discarded, indices undergo a de Bruijn-style shifting.

Similar to SQIR's use of concrete indices, $\mathcal{Q}$BRICKS-DSL's compositional structure makes it easy to map programs to their denotation: The "index" of a gate application can be computed by its nested position in the program. However, this syntax is even less convenient than SQIR's for programming: Although $\mathcal{Q}$BRICKS provides a utility function for defining `CNOT` gates between non-adjacent qubits, their underlying syntax does not support this, meaning that expressions like `CNOT` 7 2 are translated into large sequences of `CNOT` gates. QHL is presented as having variables (e.g. `q1` in Figure 1(f)), but these variables are fixed before a program is executed and persist throughout the program. In the Isabelle formalization, they are represented by natural numbers, making them comparable to SQIR concrete indices.

4.3 Extensible Language around a Unitary Core

Another key aspect of SQIR's design is its decomposition into a unitary sub-language and the non-unitary full language. While the full language (with measurement) is more powerful, its density matrix-based semantics adds unneeded complication to the proof of unitary programs. For example, given the program $U_1; U_2; U_3$, its unitary semantics is a matrix $U_3 \times U_2 \times U_1$ while its density matrix semantics is a function $\rho \mapsto U_3 \times U_2 \times U_1 \times \rho \times U_1^\dagger \times U_2^\dagger \times U_3^\dagger$. The latter is a larger term, with a type that is harder to work with. This added complexity, borne by $\mathcal{Q}$WIRE and QHL, lacks a compelling justification given that many algorithms can be viewed as unitary programs with measurement occurring implicitly at their conclusion (see Section 4.7).

On the other hand, $\mathcal{Q}$BRICKS' semantics is based on (higher-order) path-sums, which cannot describe mixed states, and thus cannot give a semantics to measurement. SQIR's design allows for a "best of both worlds," utilizing a unitary semantics when possible, but supporting non-unitary semantics when needed. Furthermore, as we show in Section 4.6, abstractions like path-sums can be easily defined on top of SQIR's unitary semantics.

4.4 Semantics of Ill-typed Programs

We say that a SQIR program is well-typed if every gate is applied to indices within range of the global register and indices used in each multi-qubit gate are distinct. This second condition enforces quantum mechanics' *no-cloning theorem*, which disallows copying an arbitrary quantum state, as would be required to evaluate an expression like `CNOT q q`. For example, `SWAP d a b` is well-typed if $a < d$, $b < d$, and $a \neq b$.

$\mathcal{Q}$WIRE addresses this issue through its linear type system, which also guarantees that qubits are never reused. However, well-typedness is a (non-trivial) extrinsic proposition in $\mathcal{Q}$WIRE, meaning that many proofs require an assumption that the input program is well-typed and must manipulate this typing judgment within the proof. $\mathcal{Q}$BRICKS avoids the issue of well-typedness through its language design: It is not possible to construct an ill-typed circuit using sequential and parallel composition. The Isabelle implementation of QHL uses a well-typedness predicate to enforce some program restrictions (e.g. the gate in a unitary application is indeed a unitary matrix), but the issue of gate argument validity is enforced by Isabelle's type system: Gate arguments are represented as a set (disallowing duplicates) where all elements are valid variables.

In SQIR, ill-typed programs are denoted by the zero matrix. This often means that we do not need to explicitly assume or prove that a program is well-typed in order to state a property about its semantics, thereby removing clutter from theorems and proofs. For example, we can prove symmetry of `SWAP`, i.e. `SWAP d a b` $\equiv$ `SWAP d b a`, without any well-typedness constraint because either both sides of the equation are well-typed or both are ill-typed. However, we cannot always avoid well-typedness preconditions. Say we want to prove transitivity of `SWAP`, i.e. `SWAP d a c` $\equiv$ `SWAP d a b ; SWAP d b c`. In this case the left-hand side may be well-typed while the right-hand side is ill-typed. To verify this equivalence, we (minimally) need the precondition $\texttt{b} < \texttt{d} \wedge \texttt{b} \neq \texttt{a} \wedge \texttt{b} \neq \texttt{c}$. We capture these in our `uc_well_typed` predicate, which resembles the `WF_Matrix` predicate (used in the `SWAP` example in Section 3.2) that guarantees that a matrix's non-zero entries are all within its bounds [17, Section 3.3]. Both conditions are easily checked via automation.

4.5 Automation for Matrix Expressions

The SQIR development provides a variety of automation techniques for dealing with matrix expressions. Most of this automation is focused on simplifying matrix terms to be easier to work with. The best example of this is our `gridify` tactic [17, Section 4.5], which rewrites terms into *grid normal form* where matrix addition is on the outside, followed by tensor product, with matrix multiplication on the inside, i.e., $((..\times..)\otimes(..\times..))+((..\times..)\otimes(..\times..))$. Most of the circuit equivalences available in SQIR (e.g. $\forall$ `a`, `b`, `c`. `CNOT a c ; CNOT b c` $\equiv$ `CNOT b c ; CNOT a c`) are proved using `gridify`. This style of automation is available in other verification tools too; `gridify` is similar to Liu et al.'s Isabelle tactic for matrix normalization [18, Section 5.1]. $\mathcal{Q}$BRICKS avoids the issue by using path-sums; they provide a matrix semantics for comparison's sake, but do not discuss automation for it.

Some of our automation is aimed at alleviating difficulties caused by our use of *phantom types* [29] to store the dimensions of a matrix, the rationale of which is explained in our prior work [17, Section 3.3]. In our development, matrices have the type `Matrix m n`, where `m` is the number of rows and `n` is the number of columns. One challenge with this definition is that the dimensions stored in the type may be "out of sync" with the structure of the expression itself. For example, due to simplification, rewriting, or declaration, the expression $|0\rangle \otimes |0\rangle$ may be annotated with the type `Vector` 4, although rewrite rules expect it to be of the form `Vector` $(2 * 2)$. We provide a tactic `restore_dims` that analyzes the structure of a term and rewrites its type to the desired form, allowing for more effective automated simplification.

4.6 Vector State Abstractions

To verify that the `SWAP` program has the intended semantics, we can unfold its definition (`CNOT a b`; `CNOT b a`; `CNOT a b`) and compute the associated matrix expression. However, while this proof is made simpler by automation like `gridify`, it is still fairly complicated considering that `SWAP` has a simple classical (non-quantum) purpose. In fact, this operation is much more naturally analyzed using its action on basis states. A *(computational) basis state* is any state of the form $|i_1 \dots i_d\rangle$ for $i_1, \dots, i_d \in \{0, 1\}$ (so $|00\rangle$ and $|11\rangle$ are basis states, while $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ is not). The set of all d-qubit basis states form a basis for the underlying d-dimensional vector space, meaning that any $2^d \times 2^d$ unitary operation can be uniquely described by its action on those basis states.

Using basis states, the reasoning for our `SWAP` example proceeds as follows, where we use $|\dots x \dots y \dots\rangle$ as informal notation to describe the state where the qubit at index a is in state x and the qubit at index b is in state y.

1. Begin with the state $|\dots x \dots y \dots\rangle$.
2. $CNOT\ a\ b$ produces $|\dots x \dots (x \oplus y) \dots\rangle$.
3. $CNOT\ b\ a$ produces $|\dots (x \oplus (x \oplus y)) \dots (x \oplus y) \dots\rangle = |\dots y \dots (x \oplus y) \dots\rangle$.
4. $CNOT\ a\ b$ produces $|\dots y \dots (y \oplus (x \oplus y)) \dots\rangle = |\dots y \dots x \dots\rangle$.

In our development, we describe basis states using `f_to_vec d f` where `d` : $\mathbb{N}$ and `f` : $\mathbb{N} \to \mathbb{B}$. This describes a d-qubit quantum state where qubit i is in the basis state $f(i)$, and `false` corresponds to 0 and `true` to 1. We also sometimes describe basis states using `basis_vector d i` where $i < 2^d$ is the index of the only 1 in the vector. We provide methods to translate between the two representations (simply converting between binary and decimal encodings). For the remainder of the paper, we will write $|\mathtt{f}\rangle$ for `f_to_vec n f` and $|\mathtt{i}\rangle$ for `basis_vector n i`, omitting the `n` parameter when it is clear from the context.

We prove a variety of facts about the actions of gates on basis states. For example, the following succinctly describe the behavior of the $CNOT$ and $Rz(\theta)$ gates, where $Rz(\theta) = U_R(0, 0, \theta)$:

```
Lemma f_to_vec_CNOT : ∀ (d i j : ℕ) (f : ℕ → 𝔹),
  i < d → j < d → i ≠ j →
  let f' := update f j (f j ⊕ f i) in
  ⟦CNOT i j⟧_d × |f⟩ = |f'⟩.

Lemma f_to_vec_Rz: ∀ (d j : ℕ) (θ : R) (f : ℕ → 𝔹),
  j < d →
  ⟦Rz θ j⟧_d × |f⟩ = e^{iθ(f j)} * |f⟩.
```

Above, `update f i v` updates the value of `f` at index `i` to be `v` (i.e. for the resulting function f', $f'(i) = v$ and $f'(j) = f(j)$ for all $j \neq i$). So $CNOT\ i\ j$ has the effect of updating the j^{th} entry of the input state to the exclusive-or of its i^{th} and j^{th} entries. $Rz\ \theta\ j$ updates the *phase* associated with the input state.

There are several advantages to applying these rewrite rules instead of unfolding the definitions of $[\![$`CNOT i j`$]\!]_d$ and $[\![$`Rz` θ `j`$]\!]_d$. For example, these rewrite rules assume well-typedness and do not depend on the ordering of qubit arguments, avoiding the case analysis needed in `gridify` [17, Section 4.5]. In addition, the rule for $CNOT$ above is simpler to work with than the general unitary semantics ($CNOT \mapsto \_ \otimes \left(\begin{smallmatrix}1&0\\0&0\end{smallmatrix}\right) \otimes \_ \otimes I_2 \otimes \_ \ + \ \_ \otimes \left(\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right) \otimes \_ \otimes \sigma_x \otimes \_$).

As a concrete example of where vector-based reasoning was critical, consider the three-qubit Toffoli gate, which implements a *controlled-controlled-not*, and can be thought of as the quantum equivalent of an *and* gate. It is frequently used in algorithms, but (like all n-qubit gates with $n > 2$) rarely supported in hardware, meaning that it must be decomposed into more basic gates before execution. In practice, we found `gridify` too inefficient to verify the standard decomposition of the gate [22, Chapter 4], shown below.

```
Definition TOFF {d} a b c : ucom base d :=
  H c ; CNOT b c ; T† c ; CNOT a c ; T c ; CNOT b c ; T† c ;
  CNOT a c ; CNOT a b ; T† b ; CNOT a b ; T a ; T b ; T c ; H c.
```

However, like SWAP, the semantics of the Toffoli gate is naturally expressed through its action on basis states:

```
Lemma f_to_vec_TOFF : ∀ (d a b c : ℕ) (f : ℕ → 𝔹),
  a < d → b < d → c < d →
  a ≠ b → a ≠ c → b ≠ c →
  let f' := update f c (f c ⊕ (f a && f b)) in
  ⟦TOFF a b c⟧_d × |f⟩ = |f'⟩.
```

The proof of `f_to_vec_TOFF` is almost entirely automated using a tactic that rewrites using the `f_to_vec` lemmas shown above, since `T` and $\mathtt{T}^\dagger$ are `Rz (PI / 4)` and `Rz (−PI / 4)`, respectively.

The `f_to_vec` abstraction is simple and easy to use, but not universally applicable: Not all quantum algorithms produce basis states, or even sums over a small number of basis states, and reasoning about 2^d terms of the form $|i_1 \dots i_d\rangle$ is no easier than reasoning directly about matrices. To support more general types of quantum states we define indexed sums and tensor (Kronecker) products of vectors.

```
Fixpoint vsum {d} n (f: ℕ → Vector d) : Vector d := ...
Fixpoint vkron n (f: ℕ → Vector 2) : Vector 2^n := ...
```

As an example of a state that uses these constructs, the action of n parallel Hadamard gates on the state $|f\rangle$ can be written as

$$\texttt{vkron n (fun i} \Rightarrow \tfrac{1}{\sqrt{2}}(|0\rangle + (-1)^{f(i)}\,|1\rangle)) \quad \text{or} \quad \tfrac{1}{\sqrt{2^n}} * (\texttt{vsum } 2^n \texttt{ (fun i} \Rightarrow (-1)^{\texttt{to\_int}(f)\bullet i} * |i\rangle)),$$

both commonly-used facts in quantum algorithms. For the remainder of the paper, we will write $\sum_{i=0}^{n-1} f(i)$ for `vsum n (fun i ⇒ f i)` and $\bigotimes_{i=0}^{n-1} f(i)$ for `vkron n (fun i ⇒ f i)`.

Relation with Path-sums

Our `vsum` and `vkron` definitions share similarities with the *path-sums* [1, 2] semantics used by 𝒬Bricks [7]. In the path-sums formalism, every unitary transformation is represented as a function of the form

$$|x\rangle \to \frac{1}{\sqrt{2^m}} \sum_{y=0}^{2^m-1} e^{2\pi i P(x,y)/2^m}\,|f(x,y)\rangle$$

where $m \in \mathbb{N}$, P is an arithmetic function over x and y, and f is of the form $|f_1(x,y)\rangle \otimes \cdots \otimes |f_m(x,y)\rangle$ where each f_i is a Boolean function over x and y. For instance, the Hadamard gate H has the form $|x\rangle \to \frac{1}{\sqrt{2}} \sum_{y=0}^{1} e^{2\pi i xy/2}\,|y\rangle$. Path-sums provide a compact way to describe the behavior of unitary matrices and are closed under matrix and tensor products, making them well-suited for automation. They can be naturally described in terms of our `vkron` and `vsum` vector-state abstractions:

```
Definition path_sum (m : ℕ) P f x :=
  vsum 2^m (fun y ⇒ e^(2πiP(x,y)/2^m) * (vkron m (fun i ⇒ f i x y))).
```

As above, `P` is an arithmetic function over `x` and `y` and `f i` is a Boolean function over `x` and `y` for any `i`.

4.7 Measurement Predicates

The proofs in Section 5 do not use the non-unitary semantics directly, but instead describe the probability of different measurement outcomes using predicates `probability_of_outcome` and `prob_partial_meas`.

```
(* Probability of measuring φ given input ψ. *)
Definition probability_of_outcome {n} (φ ψ : Vector n) : R :=
  let c := (φ† × ψ) 0 0 in |c|².

(* Probability of measuring φ on the first n qubits given (n+m) qubit input ψ. *)
Definition prob_partial_meas {n m} (φ : Vector 2ⁿ) (ψ : Vector 2ⁿ⁺ᵐ) :=
  ‖ (φ† ⊗ I₂ᵐ) × ψ ‖².
```

Above, $\|v\|$ is the 2-norm of vector v and $|c|$ is the complex norm of c. In formal terms, the "probability of measuring φ" is the probability of outcome φ when measuring a state in the basis $\{\varphi \times \varphi^\dagger, \mathtt{I}_{2^n} - \varphi \times \varphi^\dagger\}$.

The *principle of deferred measurement* [22, Chapter 4] says that measurement can always be deferred until the end of a quantum computation without changing the result. However, we included measurement in Section 3.3 because it is an important feature of quantum programming languages that is used in a variety of constructs like repeat-until-success loops [24] and error-correcting codes [12]. $\mathcal{Q}$BRICKS also uses measurement predicates, but unlike SQIR does not support a general measurement construct.

5 Proofs of Quantum Algorithms

In this section we discuss the formal verification of two classic quantum algorithms: Grover's algorithm [22, Chapter 6] and quantum phase estimation [22, Chapter 5]. We present additional, simpler examples in Appendices A and B of the extended version of this paper. All proofs and specifications follow the corresponding textbook arguments.

5.1 Grover's Algorithm

Overview

Given a circuit implementing Boolean oracle $f : \{0,1\}^n \to \{0,1\}$, the goal of Grover's algorithm is to find an input x satisfying $f(x) = 1$. Suppose that $n \geq 2$. In the classical (worst-)case where $f(x) = 1$ has a unique solution, finding this solution requires $O(2^n)$ queries to the oracle. However, the quantum algorithm finds the solution with high probability using only $O(\sqrt{2^n})$ queries.

The algorithm alternates between applying the oracle and a "diffusion operator." Individually, these operations each perform a reflection in the two-dimensional space spanned by the input vector (a uniform superposition) and a uniform superposition over the solutions to f. Together, they perform a rotation in the same space. By choosing an appropriate number of iterations i, the algorithm will rotate the input state to be suitably close to the solution vector. The SQIR definition of Grover's algorithm is shown in Figure 2.

The SQIR version of Grover's algorithm is 15 lines, excluding utility definitions like `control` and `npar`. The specification and proof are around 770 lines. The proof took approximately one person-week.

Proof Details

The statement of correctness says that after i iterations, the probability of measuring a solution is $\sin^2((2i+1)\theta)$ where $\theta = \arcsin(\sqrt{k/2^n})$ and k is the number of satisfying solutions to f. Note that this implies that the optimal number of iterations is $\frac{\pi}{4}\sqrt{\frac{2^n}{k}}$.

```
(* Controlled-X with target (n-1) and controls 0, 1, ..., n-2. *)
Fixpoint generalized_Toffoli' n0 : ucom base n :=
  match n0 with
  | 0 | S 0 ⇒ X (n - 1)
  | S n0' ⇒ control (n - n0) (generalized_Toffoli' n0')
  end.
Definition generalized_Toffoli := generalized_Toffoli' n.

(* Diffusion operator. *)
Definition diff : ucom base n :=
  npar n H; npar n X ;
  H (n - 1) ; generalized_Toffoli ; H (n - 1) ;
  npar n X; npar n H.

(* Main program (iterates applying U_f and diff). *)
Definition body := U_f ; cast diff (S n).
Definition grover i := X n ; npar (S n) H ; niter i body.
```

Figure 2 Grover's algorithm in SQIR. `control` performs a unitary program conditioned on an input qubit, `npar` performs copies of a unitary program in parallel, `cast` is a no-op that changes the dimension in a `ucom`'s type, and `niter` iterates a unitary program.

We begin the proof by showing that the uniform superposition can be rewritten as a sum of "good" states (ψg) that satisfy f and "bad" states (ψb) that do not satisfy f.

Definition ψ := $\frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} |k\rangle$.

Definition θ := asin $(\sqrt{k/2^n})$.
Lemma decompose_ψ : ψ = (sin θ) ψg + (cos θ) ψb.

We then prove that `U_f` and `diff` perform the expected reflections (e.g. $[\![\texttt{diff}]\!]_n = -2\,|\psi\rangle\langle\psi| + I_{2^n}$) and the following key lemma, which shows the output state after i iterations of `body`.

Lemma loop_body_action_on_unif_superpos : ∀ i,
  $[\![\text{body}]\!]^i_{n+1}$ (ψ ⊗ |-⟩) =
    $(-1)^i$ (sin ((2 * i + 1) * θ) ψg + cos ((2 * i + 1) * θ) ψb) ⊗ |-⟩.

This property is straightforward to prove by induction on `i`, and implies the desired result, which specifies the probability of measuring any solution to f.

Lemma grover_correct : ∀ i,
  Rsum 2^n (fun z ⇒ if f z
                    then prob_partial_meas |z⟩ ($[\![\text{grover i}]\!]_{n+1}$ × $|0\rangle^{n+1}$)
                    else 0) =
  (sin ((2 * i + 1) * θ))^2.

That is, the sum over the probability of all possible outcomes z such that $f(z)$ is true is $\sin^2((2i+1)\theta)$. Above, `Rsum` is a sum over real numbers.

5.2 Quantum Phase Estimation

Overview

Given a unitary matrix U and eigenvector $|\psi\rangle$ such that $U|\psi\rangle = e^{2\pi i\theta}|\psi\rangle$, the goal of quantum phase estimation (QPE) is to find a k-bit representation of θ. In the case where θ can be exactly represented using k bits (i.e. $\theta = z/2^k$ for some $z \in \mathbb{Z}$), QPE recovers θ

exactly. Otherwise, the algorithm finds a good k-bit approximation with high probability. QPE is often used as a subroutine in quantum algorithms, most famously Shor's factoring algorithm [31].

The SQIR program for QPE is shown in Figure 3. For comparison, the standard circuit diagrams for QPE and the quantum Fourier transform (QFT), which is used as a subroutine in QPE, are shown in Figure 4. The SQIR version of QPE is around 40 lines and the specification and proof in the simple case ($\theta = z/2^k$) is around 800 lines. The fully general case ($\theta \neq z/2^k$) adds about 250 lines. The proof of the simple case was completed in about two person-weeks. When working out the proof of the general case, we found that we needed some non-trivial bounds on trigonometric functions (for $x \in \mathbb{R}$, $|\sin(x)| \leq |x|$ and if $|x| \leq \frac{1}{2}$ then $|2 * x| \leq |\sin(\pi x)|$). Laurent Théry kindly provided proofs of these facts using the Coq Interval package [21].

Proof Details

The correctness property for QPE in the case where θ can be described exactly using k bits ($\theta = z/2^k$) says that the QPE program will exactly recover z. It can be stated in SQIR's development as follows.

```
Lemma QPE_correct_simplified: ∀ k n (u : ucom base n) z (ψ : Vector 2^n),
  n > 0 → k > 1 → uc_well_typed u → WF_Matrix ψ →
  let θ := z / 2^k in
  ⟦u⟧_n × ψ = e^{2πiθ} * ψ →
  ⟦QPE k n u⟧_{k+n} × (|0⟩^k ⊗ ψ) = |z⟩ ⊗ ψ.
```

The first four conditions ensure well-formedness of the inputs. The fifth condition enforces that input ψ is an eigenvector of c. The conclusion says that running the `QPE` program computes the value z, as desired.

In the general case where θ cannot be exactly described using k bits, we instead prove that `QPE` recovers the best k-bit approximation with high probability (in particular, with probability $\geq 4/\pi^2$).

```
Lemma QPE_semantics_full : ∀ k n (u : ucom base n) z (ψ : Vector 2^n) (δ : R),
  n > 0 → k > 1 → uc_well_typed u → Pure_State_Vector ψ →
  -1 / 2^{k+1} ≤ δ < 1 / 2^{k+1} → δ ≠ 0 →
  let θ := z / 2^k + δ in
  ⟦u⟧_n × ψ = e^{2πiθ} * ψ →
  prob_partial_meas |z⟩ (⟦QPE k n u⟧_{k+n} × (|0⟩^k ⊗ ψ)) ≥ 4 / π^2.
```

`Pure_State_Vector` is a restricted form of `WF_Matrix` that requires a vector to have norm 1.

As an example of the reasoning that goes into proving these properties, consider the QFT subroutine of QPE. The correctness property for `controlled_rotations` says that evaluating the program on input $|x\rangle$ will produce the state $e^{2\pi i(x_0 \ \cdot \ x_1x_2...x_{n-1})/2^n} |x\rangle$ where x_0 is the highest-order bit of x represented as a binary string and $x_1x_2...x_{n-1}$ are the lower-order $n-1$ bits.

```
Lemma controlled_rotations_correct : ∀ n x,
  n > 1 → ⟦controlled_rotations n⟧_n × |x⟩ = e^{2πi(x_0 · x_1x_2...x_{n-1})/2^n}|x⟩.
```

We can prove this property via induction on n. In the base case ($n = 2$) we have that x is a 2-bit string x_0x_1. In this case, the output of the program is $e^{2\pi i(x_0 \cdot x_1)/2^2} |x_0x_1\rangle$, as desired. In the inductive step, we assume that:

$$[\![\texttt{controlled\_rotations n}]\!]_n \times |x_1x_2...x_{n-1}\rangle = e^{2\pi i(x_0 \ \cdot \ x_1x_2...x_{n-1})/2^n} |x_1x_2...x_{n-1}\rangle.$$

```
(* Controlled rotation cascade on n qubits. *)
Fixpoint controlled_rotations n : ucom base n :=
  match n with
  | 0 | 1 ⇒ SKIP
  | S n'  ⇒ controlled_rotations n' ; control n' (Rz (2π / 2^n) 0)
  end.

(* Quantum Fourier transform on n qubits. *)
Fixpoint QFT n : ucom base n :=
  match n with
  | 0    ⇒ SKIP
  | S n' ⇒ H 0 ; controlled_rotations n ; map_qubits (fun q ⇒ q + 1) (QFT n')
  end.

(* The output of QFT needs to be reversed before further processing. *)
Definition reverse_qubits n := ...
Definition QFT_w_reverse n := QFT n ; reverse_qubits n.

(* Controlled powers of u. *)
Fixpoint controlled_powers' {n} (u : ucom base n) k kmax : ucom base (kmax+n) :=
  match k with
  | 0    ⇒ SKIP
  | S k' ⇒ controlled_powers' u k' kmax ; niter 2^k' (control (kmax - k' - 1) u)
  end.
Definition controlled_powers {n} (u : ucom base n) k := controlled_powers' u k k.

(* QPE circuit for program u.
   k = number of bits in resulting estimate
   n = number of qubits in input state *)
Definition QPE k n (u : ucom base n) : ucom base (k + n) :=
  npar k H ;
  controlled_powers (map_qubits (fun q ⇒ k + q) u) k;
  invert (QFT_w_reverse k).
```

Figure 3 SQIR definition of QPE. Some type annotations and calls to `cast` have been removed for clarity. `control`, `map_qubits`, `niter`, `npar`, and `invert` are Coq functions that transform SQIR programs; we have proved that they have the expected behavior (e.g. $\forall\, u.\ [\![\text{invert } u]\!]_n = [\![u]\!]_n^\dagger$).

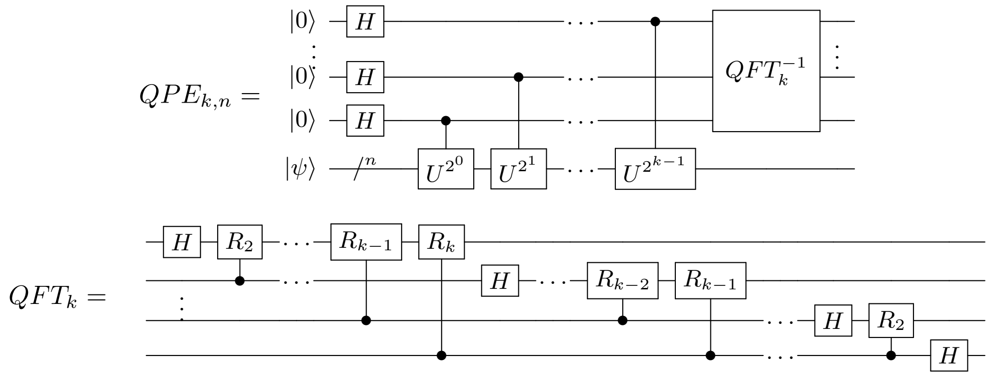


Figure 4 Circuit for quantum phase estimation (QPE) with k bits of precision and an n-qubit input state (top) and quantum Fourier transform (QFT) on k qubits (bottom). $|\psi\rangle$ and U are inputs to QPE. R_m is a z-axis rotation by $2\pi/2^m$.

$$\begin{aligned}
&[\![\texttt{controlled\_rotations (n+1)}]\!]_{n+1} \times |x\rangle \\
&\quad = [\![\texttt{control } x_n \texttt{ (Rz } (2\pi/2^{n+1}) \texttt{ 0}]\!]_{n+1} \times [\![\texttt{controlled\_rotations n}]\!]_{n+1} \times |x\rangle \\
&\quad = [\![\texttt{control } x_n \texttt{ (Rz } (2\pi/2^{n+1}) \texttt{ 0}]\!]_{n+1} \times e^{2\pi i (x_0 \,\cdot\, x_1 x_2 \ldots x_{n-1})/2^n} |x_1 x_2 \ldots x_{n-1} x_n\rangle \\
&\quad = e^{2\pi i (x_0 \,\cdot\, x_n)/2^{n+1}} e^{2\pi i (x_0 \,\cdot\, x_1 x_2 \ldots x_{n-1})/2^n} |x_1 x_2 \ldots x_{n-1} x_n\rangle \\
&\quad = e^{2\pi i (x_0 \,\cdot\, x_1 x_2 \ldots x_n)/2^{n+1}} |x_1 x_2 \ldots x_{n-1} x_n\rangle
\end{aligned}$$

Figure 5 Reasoning used in the proof of `controlled_rotations`. The first step unfolds the definition of `controlled_rotations`; the second step applies the inductive hypothesis; the third step evaluates the semantics of `control`; and the fourth step combines the exponential terms.

We then perform the simplifications shown in Figure 5, which complete the proof.

Our correctness property for `QFT n` (shown below) can similarly be proved by induction on n, and relies on the lemma `controlled_rotations_correct`.

```
Lemma QFT_semantics : ∀ n x, n>0 → ⟦QFT n⟧_n × |x⟩ = 1/√2^n ⨂_{j=0}^{n-1}(|0⟩ + e^{2πix/2^{n-j}} |1⟩).
```

6 Open Problems and Future Work

We previously presented SQIR as the intermediate representation in a verified circuit optimizer [17]. In this paper, we presented SQIR as a source language for quantum programming and discussed how our design choices (e.g. concrete indices, unitary core, vector state abstractions) ease proofs about SQIR programs. But there is still work to be done.

So far, work on formally verified quantum computation has been limited to textbook quantum algorithms like QPE and Grover's. Although these algorithms are a useful stress-test for tools, they do not accurately reflect the types of quantum programs that are expected to run on near-term machines. Near-term algorithms are usually *approximate*. They do not implement the desired operation exactly, but rather perform an operation "close" to what was intended. Our `probability_of_outcome` and `prob_partial_meas` predicates can be used to express distance between vector states, but we currently do not have support for reasoning about distance between general quantum operations.

Another issue is that near-term algorithms often need to account for hardware errors. Thus, verifying these algorithms may require considering their behavior in the presence of errors. So far, most of our work in SQIR has revolved around the unitary semantics and vector-based state abstractions because we find these simpler to work with. However, it is more natural to describe states subject to error using density matrices, since noisy states are mixtures of pure states [22, Chapter 8].

On another front, there is important work to be done on describing quantum algorithms and correctness properties at a higher level of abstraction. The proofs and definitions in this paper follow the standard textbook presentation, but are still lower-level than similar proofs about classical programs. Rather than working from the circuit model, used in $\mathcal{Q}$WIRE, SQIR, $\mathcal{Q}$BRICKS, and (to some extent) QWhile, it would be interesting to verify programs written in higher-level languages like Silq [4] or Q# [33].

We hope that SQIR's extensible design and flexible semantics, developed while verifying circuit optimizations and textbook quantum programs, will serve as a solid foundation for the proposed verification efforts above and those to come.

References

1 Matt Amy. *Formal Methods in Quantum Circuit Design.* PhD thesis, University of Waterloo, 2019.

2 Matthew Amy. Towards large-scale functional verification of universal quantum circuits. In *Proceedings of the 15th International Conference on Quantum Physics and Logic, QPL 2018*, June 2018.

3 Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin, and Harald Weinfurter. Elementary gates for quantum computation. *Phys. Rev. A*, 52:3457–3467, November 1995. doi:10.1103/PhysRevA.52.3457.

4 Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, 2020.

5 Jaap Boender, Florian Kammüller, and Rajagopal Nagarajan. Formalization of quantum protocols using coq. In Chris Heunen, Peter Selinger, and Jamie Vicary, editors, *Proceedings of the 12th International Workshop on Quantum Physics and Logic, Oxford, U.K., July 15-17, 2015*, volume 195 of *Electronic Proceedings in Theoretical Computer Science*, pages 71–83. Open Publishing Association, 2015. doi:10.4204/EPTCS.195.6.

6 Anthony Bordg, Hanna Lachnitt, and Yijun He. Certified quantum computation in Isabelle/HOL. *Journal of Automated Reasoning*, 2020. doi:10.1007/s10817-020-09584-7.

7 Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoit Valiron. Toward certified quantum programming. *arXiv e-prints*, 2020. arXiv:2003.05841.

8 The Coq Development Team. The coq proof assistant, version 8.10.0, 2019. doi:10.5281/zenodo.3476303.

9 Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open Quantum Assembly Language. *arXiv e-prints*, July 2017. arXiv:1707.03429.

10 David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 439(1907):553–558, 1992.

11 Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In *Proceedings of the 22nd European Symposium on Programming*, Lecture Notes in Computer Science, 2013.

12 Daniel Gottesman. An introduction to quantum error correction and fault-tolerant quantum computation. In *Quantum information science and its contributions to mathematics, Proceedings of Symposia in Applied Mathematics*, volume 68, pages 13–58, 2010.

13 Alexander Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: A scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2013, pages 333–342, 2013.

14 Alexander S Green. *Towards a formally verified functional quantum programming language.* PhD thesis, University of Nottingham, 2010.

15 Daniel M. Greenberger, Michael A. Horne, and Anton Zeilinger. *Going Beyond Bell's Theorem*, pages 69–72. Springer Netherlands, Dordrecht, 1989. doi:10.1007/978-94-017-0849-4_10.

16 Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM Symposium on Theory of Computing*, pages 212–219, 1996.

17 Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. A verified optimizer for quantum circuits. *Proceedings of the ACM on Programming Languages*, 5(37), 2021.

18 Junyi Liu, Bohua Zhan, Shuling Wang, Shenggang Ying, Tao Liu, Yangjia Li, Mingsheng Ying, and Naijun Zhan. Formal verification of quantum algorithms using quantum hoare logic. In *Computer Aided Verification - 31st International Conference, CAV 2019, New*

York City, NY, USA, July 15-18, 2019, Proceedings, Part II, pages 187–207, 2019. doi:10.1007/978-3-030-25543-5_12.

19 Junyi Liu, Bohua Zhan, Shuling Wang, Shenggang Ying, Tao Liu, Yangjia Li, Mingsheng Ying, and Naijun Zhan. Quantum hoare logic. *Archive of Formal Proofs*, March 2019. , Formal proof development. URL: http://isa-afp.org/entries/QHLProver.html.

20 Tao Liu, Yangjia Li, Shuling Wang, Mingsheng Ying, and Naijun Zhan. A theorem prover for quantum hoare logic and its applications. *arXiv preprint arXiv:1601.03835*, 2016.

21 Guillaume Melquiond. Interval package for coq, 2020. URL: https://gitlab.inria.fr/coqinterval/interval.

22 Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

23 Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

24 Adam Paetznick and Krysta M Svore. Repeat-until-success: non-deterministic decomposition of single-qubit unitaries. *Quantum Information & Computation*, 14(15-16):1277–1301, 2014.

25 Jennifer Paykin, Robert Rand, and Steve Zdancewic. QWIRE: A core language for quantum circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 846–858, New York, NY, USA, 2017. ACM. doi:10.1145/3009837.3009894.

26 Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 199–208, New York, NY, USA, 1988. ACM. doi:10.1145/53990.54010.

27 Robert Rand. *Formally Verified Quantum Programming*. PhD thesis, University of Pennsylvania, 2018.

28 Robert Rand, Jennifer Paykin, and Steve Zdancewic. QWIRE practice: Formal verification of quantum circuits in Coq. In *Proceedings 14th International Conference on Quantum Physics and Logic, QPL 2017, Nijmegen, The Netherlands, 3-7 July 2017.*, pages 119–132, 2017. doi:10.4204/EPTCS.266.8.

29 Robert Rand, Jennifer Paykin, and Steve Zdancewic. Phantom types for quantum programs. The Fourth International Workshop on Coq for Programming Languages, January 2018.

30 Peter Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, August 2004.

31 P. W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, FOCS '94, 1994.

32 DR Simon. On the power of quantum computation. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 116–123, 1994.

33 Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#: Enabling scalable quantum computing and development with a high-level dsl. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, page 7. ACM, 2018.

34 Mingsheng Ying. Floyd–hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(6):19, 2011.

Formalization of Basic Combinatorics on Words

Štěpán Holub
Department of Algebra, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic

Štěpán Starosta
Dept. of Applied Math., Faculty of Information Technology, Czech Technical University in Prague, Czech Republic

Abstract

Combinatorics on Words is a rather young domain encompassing the study of words and formal languages. An archetypal example of a task in Combinatorics on Words is to solve the equation $x \cdot y = y \cdot x$, i.e., to describe words that commute.

This contribution contains formalization of three important classical results in Isabelle/HOL. Namely i) the Periodicity Lemma (a.k.a. the theorem of Fine and Wilf), including a construction of a word proving its optimality; ii) the solution of the equation $x^a \cdot y^b = z^c$ with $2 \leq a, b, c$, known as the Lyndon-Schützenberger Equation; and iii) the Graph Lemma, which yields a generic upper bound on the rank of a solution of a system of equations.

The formalization of those results is based on an evolving toolkit of several hundred auxiliary results which provide for smooth reasoning within more complex tasks.

2012 ACM Subject Classification Mathematics of computing → Combinatorics on words

Keywords and phrases combinatorics on words, formalization, Isabelle/HOL

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.22

Supplementary Material *Software (Code Repository)*: `https://gitlab.com/formalcow/combinatorics-on-words-formalized`
archived at `swh:1:dir:78c9955742137e63eb137885a27acfd231a576f5`

Funding The authors acknowledge support by the Czech Science Foundation grant GAČR 20-20621S.

Acknowledgements We would like to thank Manuel Eberl for useful suggestions concerning formalization. We are also grateful to anonymous referees whose criticism helped to improve the presentation significantly.

1 Introduction

Combinatorics on Words usually dates its beginning (cf. Berstel and Perrin [5]) back to the works of Axel Thue on repetitions in infinite words published more than hundred years ago [34, 35]. Nevertheless, the first (collective) monograph on the subject was published only in 1983 [26]. In this paper, we are interested in the part of the field dealing with finite (rather than infinite) words, which in particular includes solving word equations (without constants). Solving general word equations is a difficult algorithmic task. Once believed to be undecidable, the first algorithm was described by Makanin in 1977 [28] (see [7] for a self-contained exposition by Diekert). Currently, the approach of *recompression* introduced by Jeż [22] is the most efficient one, with nondeterministic linear space complexity (see Jeż [23]). While the problem is NP hard, it remains a challenging open question whether it is NP complete.

We believe that combinatorics of (finite) words is an area where computer assisted formalization may be very helpful. Proofs of even fairly simple results tend to be tedious and repetitive, featuring complicated analysis of cases, which makes them hard (both for referees and readers) to verify. Moreover, despite the short history of the field, basic auxiliary results are sometimes forgotten and rediscovered, or simply repeatedly proven in many papers. Some easily stated problems, like the solution of equations in three unknowns by Nowotka and Saarela [29, 30], or the characterization of binary equality languages by the first author [19], are nontrivial classification tasks, for which computer formalization can be decisive. Prominent examples are classification of finite groups formalized by Gonthier et al. in Coq [9], four-colour problem formalized by Gontihier also in Coq [11] or Kepler's conjecture formalized by Hales et al. in HOL Light and Isabelle [12]. Our long term ambition is to create a library of formalized results with three objectives: 1) verified basic facts (the "folklore") that can become a standard starting point for further formalization; 2) verified classical results, making sure that occasional gaps in the published proofs are not fatal, and sometimes providing polished, more straightforward proofs; 3) allowing to push boundaries of the current research in areas where a sheer complexity of the topic may be the most important barrier for further advances. Automation of repeated steps can make a crucial difference here. (In particular, we have in mind the above mentioned classification tasks.)

In this paper, we present advances in the first two of those objectives. Namely, we formalize three important classical results, which together reveal the main features of the general project of formalization of word equations. We want our formalization to reflect as clearly as possible the main ideas that would be given in (a good) paper proof. This requires an auxiliary background theory that collects humanly trivial facts about words that are nevertheless not covered by the main Isabelle/HOL library. Our auxiliary theory contains several hundred claims which we deem of fundamental nature in order to formalize some advanced results in Combinatorics on Words (see more in Section 2.4.1).

The first classical result presented in this paper is the Periodicity Lemma, also known as the theorem of Fine and Wilf [10], which regulates the possibility of a word having more than one period. It states that if a word of length at least $p + q - \gcd(p, q)$ has periods p and q, then it has also a period $\gcd(p, q)$. We present here a particularly simple proof at which we arrived through the formalization process. We take the opportunity of this simple example to illustrate some common features of our project. We have also formalized an explicit verified construction of a word witnessing that the bound given in the Periodicity Lemma is sharp. For example, the word `0102010` of length seven has periods 4 and 6 but not the period $\gcd(4, 6) = 2$, while any word of length at least eight having periods four and six has also a period two.

The second theorem deals with the equation $x^a y^b = z^c$ with $2 \leq a, b, c$. We formalize a proof that this equation admits only solutions where all unknown words x, y, and z are powers of a common word. Such solutions are called *periodic.* This classical result was first proven by Lyndon and Schützenberger [27] in a more general setting of free groups. Historically, it was the first challenging equation with three unknowns whose solutions were completely characterized. The presented proofs of the Periodicity Lemma and the solution of the Lyndon and Schützenberger equation (LSE) are mainly combinatorial.

The need to deal with equations like LSE in an *ad hoc* manner is tightly related to the fact that word equations are rather immune against the so called *defect effect.* To understand what this means, consider systems of linear equations. Each new independent linear equation decreases the degree of freedom of a solution of the corresponding system, so that n independent equations over n unknowns admit only one solution. In contrast, there is no known upper bound on the size of an independent system of word equations over $n \geq 4$ unknowns, and only a rough bound for $n = 3$ (see e.g. Saarela [31] for a survey).

The best general form of the defect effect for word equations is provided by the Graph Lemma, which is the third important result presented and formalized in this paper. We shall discuss the Graph Lemma in detail in Section 2.3. Here, let us illustrate the main idea by an example. Consider the following system of two equations over three unknowns:

$$\begin{aligned} xyz &= yzx, \\ xzy &= zyx. \end{aligned}$$

We construct an undirected graph whose vertices are the unknowns x, y, z. The edges, one for each equation of the system, connect first letters of left and right hand side of the equation. In our example, the edges are (x, y) and (x, z). By the Graph Lemma, such a system has periodic solutions only, since the resulting graph is connected. In other terms, since the graph has *one* connected component, all three words in any solution are powers of *one* common word. Consider, on the other hand, the system

$$\begin{aligned} xyz &= zyx, \\ xyyz &= zyyx. \end{aligned}$$

The graph of this system has the unique edge (x, z), hence the Graph Lemma does not tell us whether the system has a non-periodic solution or not. In fact, this system has an obvious non-periodic solution $x \mapsto a$, $y \mapsto b$, $z \mapsto a$.

Our approach to the proof of the Graph Lemma exploits the algebraic concept of the *free hull* of a solution, and of its rank, that is, of the cardinality of its basis. This also means that auxiliary claims needed in the proof of the Graph Lemma are of a more algebraic flavor, compared to the proof of the Periodicity Lemma and the solution of the LSE. These claims are covered by the second background auxiliary theory used in this paper, described in more detail in Section 2.4.2.

We start by introducing the notation and terminology followed by an overview of related algebraic structures and related work. In Section 2, we present the three main results and conclude by the details on the structure and background theories of the formalization.

1.1 Notation and terminology

Words are finite sequences of elements from a given set Σ, where Σ is called an *alphabet*, and its elements are called *letters*. Accordingly, we represent words by the datatype of lists in our formalization, and the alphabet is typically represented in Isabelle by a type variable $'$a. The set of all words over Σ is denoted by Σ^*, including the empty word, denoted by ε, which is represented as Nil or [] in Isabelle/HOL.

Words are endowed by the operation of *concatenation*, which corresponds to `append` for lists. Words with the operation of concatenation form a *free monoid*. The infix notation for the `append`-operation is @. For words, the concatenation is denoted by the multiplication sign $\cdot$ (which, as usual, is often omitted). We therefore allow, in our formalization, to write $\cdot$ instead of @. That is, $x \cdot y$ is equivalent to $x@y$. We write $u \leq_p v$ if u is a prefix of v, that is, if $v = u \cdot z$ for some z.

Seeing concatenation as a monoid multiplication naturally yields the concept of a power. We use the usual notation x^n of the n-th power of x in the mathematical text, and by $x^{@}n$ in the formalization. The set of all powers of a word t is usually denoted as t^* using the Kleene star familiar from regular expressions, where it is commonly used even for sets as, for example, in $\{u, v\}^*$. However, this allows a certain confusion. If G is a set of words over Σ, then G^* should denote all words over Σ generated by G. On the other hand, Σ^* denotes all words over the alphabet Σ, and the difference between the alphabet Σ and the set of words G has to be kept in mind. Strictly speaking, Σ^* is not generated by the alphabet

Σ, but rather by the set of singletons, that is, words of length one. While the difference between letters and singletons is typically ignored in the literature without any significant harm, the difference between a letter a, and the list $[a]$ must obviously be respected in the formalization. We therefore prefer to use the notation $\langle G \rangle$ for the submonoid of Σ^* generated by a set $G \subset \Sigma^*$. We also call it the *hull* of G. We nevertheless allow the expression $x \in t^*$ which is an abbreviation for $x \in \langle \{t\} \rangle$. The term `decompose` G u, abbreviated as $\mathrm{Dec}\, G\, u$, represents some decomposition of the word u into elements of G. It returns a list of words, i.e., of type 'a list list.

```
fun decompose :: 'a list set ⇒ 'a list ⇒ 'a list list (Dec - - ) where
  decompose G u = (SOME us. us ≠ ε ∧ us ∈ lists G ∧ u = concat us)
```

Hilbert's choice operator SOME is used here. The output of the function makes no good sense if the second argument is not in $\langle G \rangle$. Note, however, that even for elements of $\langle G \rangle$ the list is an unspecified choice among all possible decompositions. For example, if $G = \{a, ab, ba\}$ and $u = aba$, then $\mathrm{Dec}\, G\, u$ is either $[a, ba]$ or $[ba, a]$. This in particular implies that we cannot prove $\mathrm{Dec}\, G\, (u \cdot v) = \mathrm{Dec}\, G\, u \cdot \mathrm{Dec}\, G\, v$.

We deal with finite words only. An apparent exception is the infinite repetition $u^\omega = u \cdot u \cdot u \cdots$. However, this infinite word will be used exclusively in expressions of the form $w \leq_p u^\omega$, which is just a handy way of writing $\exists n. w \leq_p u^n$.

The length of a word w, that is the usual list `length`, is denoted by $|w|$. A word w of length n can be spelled as the list $[w_0, w_1, \ldots, w_{n-1}]$, where w_i represents the $(i+1)$-th letter of w. The first letter of a nonempty word w is also denoted hd w. The prefix of w of length $k \leq |w|$ is denoted $\mathrm{pref}_k w$ (`take` k w in Isabelle).

The word w has *a period* p if $1 \leq p$, and if $w_i = w_{i+p}$ for each $0 \leq i < |w| - p$. We allow (trivial) periods $p \geq |w|$.

One of our main interests is in *word equations*. Formally, a word equation is a pair of words $(L, R) \in X^* \times X^*$ over an alphabet X of *unknowns*. Nevertheless, the equation like $([x, y, z], [z, y, x])$ is usually written as $xyz = zyx$, a convention we already used above. A solution (in an alphabet Σ) of the equation (L, R) is a monoid morphism $f : X^* \to \Sigma^*$ (often called a *substitution*) such that $f(L) = f(R)$. (The defined concept should be more precisely described as *word equations without constants*. We do not deal with equations with constants in this paper.) The reader may further refer to Harju et al. [14].

1.2 Related algebraic structures and related work

Combinatorics of finite words focused on word equations has two basic aspects: the combinatorial and the algebraic. The combinatorial aspect is in an obvious way connected to words as lists, the algebraic aspect becomes important when considering a set of words as a generating set of a monoid. The algebraic aspect is exhibited and further discussed in Section 2.3 dedicated to the Graph Lemma. It is a basic decision of the formalization how to represent words in order to capture these two aspects. The first author in [21] conducted an inquiry into the possibility to deal with free monoids axiomatically. In particular, free monoids are fully characterized by the *equidivisibility property*:

```
lemma eqd: x · y = u · v ⟹ |x| ≤ |u| ⟹ ∃ t. x · t = u ∧ t · v = y
```

together with the provision that the length of possible decompositions of any element is bounded. Experience from this research confirms that the axiomatic approach has no advantages. On the contrary, the elements of the free monoid will eventually be represented as lists of generators in any case (so in the Lean prover, for example, `free-monoid` over alphabet α is directly defined as a synonym for `list` α). Our formalization is therefore based on the datatype of lists. This fundamental datatype is well developed in Isabelle/HOL (as well as in all other provers), and we heavily build on the theory List.thy from the Main library, and the theory Sublist.thy from the HOL-Library.

Nevertheless, from the point of view of word equations, those theories contain only the solution of the easiest nontrivial word equation, namely $x \cdot y = y \cdot x$, showing that commuting words x and y are always powers of the same (shorter) word:

```
lemma comm-append-are-replicate:
  xs @ ys = ys @ xs
  ⟹ ∃m n zs. concat (replicate m zs) = xs ∧ concat (replicate n zs) = ys
```

(We remark that this is the formulation in the 2021 release without redundant assumptions removed following our suggestion.) This result is called the *Commutation Lemma.* Since equations are our main interest, we improve readability using a slightly modified notation. Our version reads:

```
theorem comm-root:  x · y = y · x   ⟷  (∃ t. x ∈ t* ∧ y ∈ t*)
```

Here t^* denotes the set $\{t^n \mid 0 \leq n\}$.

A similar remark concerning applicability for word equations applies to potentially related area of combinatorics of free groups, or even more generally, to combinatorial theory of (free) (semi)groups. The Isabelle/HOL theory Free-Groups by Breitner [6] contains fundamental properties of free groups including recently the Ping Pong lemma, which naturally exhibits some combinatorial features related to our work. Nevertheless, there is no direct overlap.

To our knowledge, the situation in other provers is not different. The most related to Combinatorics on Words is the Coq package Coq-Combi by Hivert [18] which uses specific parts of Combinatorics on Words results to prove some other results such as the Littlewood–Richardson rule. Another Coq package which is related is Coq-free-groups, formalizing elements of the free group theory (which is not as much developed as the above mentioned Isabelle/HOL free group theory by Breitner). Another related pieces of formalization can be found in the Lean Mathematical Library: it contains a basic formalization of free groups and free monoids, with no specific tools for submonoids of free groups (besides general submonoids).

Isabelle's *Archive of Formal Proofs* [1] contains a large group of theories on Automata and formal languages. The Coq package Coq-automata is situated within the same topic. However, there is almost no overlap with word equations and questions we are interested in. For example, the theory of regular expressions (or, more generally, Kleene algebras) deals with structures on sets of languages, not with individual languages, which moreover typically are not themselves monoids. We can illustrate this by one of our recent formalizations [20]. It is a basic property of regular languages to be closed under intersection. However, to classify possible intersections $\{x, y\}^* \cap \{u, v\}^*$ of two monoids generated by pairs of non-commuting words is a nontrivial task, which has little to do with finite automata or with a general theory of regular languages.

It should be stressed that monoids as such are too general a structure, and do not provide any significant theoretical support for reasoning about lists. The main defining property of monoids, associativity, is captured by lists trivially. The single exception are properties of powers. We therefore interpret lists as an instance of the class `monoid-mult`:

```
interpretation monoid-mult ε append
```

This immediately yields a series of claims like

```
lemma power-add-list: x@n·x@m = x@(n+m)
```

where $x^{@}n$ is our notation for the interpreted `power`.

2 Presented results

2.1 The Periodicity Lemma

Periodicity is one of the most important and most studied properties of words. In our formalization, we use the following definition:

```
definition periodN :: 'a list ⇒ nat ⇒ bool
  where periodN w n =  w ≤p (take n w)^ω
```

A related definition is the definition of the *period root*:

```
definition period-root :: 'a list ⇒ 'a list ⇒ bool (- ≤p -^ω)
  where [simp]: period-root u r = (u ≤p r · u ∧ r ≠ ε)
```

with the notation $u \leq_p r^\omega$. This notation is justified by the observation that the following claims are equivalent:

- w has a period p (in the sense given in Section 1.1);
- w is a prefix of $u \cdot w$, where u is a word of length p (the period root);
- w is a prefix of u^ω.

A word can have more than one period. This possibility is regulated by the following famous result.

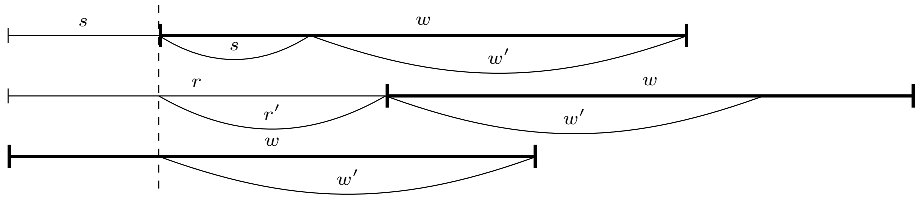
▶ **Lemma 1** (Periodicity Lemma [10]). *If a word w of length at least $p + q - \gcd(p, q)$ has periods p and q, then it also has a period $\gcd(p, q)$.*

The proof is a combination of two elementary facts. The first one is the above mentioned characterization of the period by the period root. The second one is the Commutation Lemma. We first prove the following claim, which can be seen as a modification of the Euclidean algorithm.

▶ **Lemma 2.** *Let $w \leq_p r \cdot w$ and $w \leq_p s \cdot w$. If $|r| + |s| - \gcd(|s|, |r|) \leq |w|$, then $r \cdot s = s \cdot r$.*

Proof. The assumptions imply that both s and r are prefixes of w. By symmetry, we can suppose $|s| \leq |r|$ which yields $s \leq_p r$. Let r' and w' be such that $r = s \cdot r'$ and $w = s \cdot w'$.

Then s, r' and w' satisfy the assumptions of the claim, see the following figure.



In particular, we have

$$|r'| + |s| - \gcd(|s|, |r'|) = |r| - |s| + |s| - \gcd(|s|, |r| - |s|) =$$
$$|r| + |s| - \gcd(|s|, |r|) - |s| \leq |w| - |s| = |w'|.$$

If $s = \varepsilon$, the claim holds. If s is nonempty, then we have that s and r' commute by induction on $|s| + |r|$. Hence also s and r commute. ◀

The proof of the Periodicity lemma is now easily concluded using the Commutation Lemma (see Section 1.2):

Proof of the Periodicity lemma. Assume $p \leq q$, and let t be the common root of $s = \text{pref}_p w$ and $r = \text{pref}_q w$. Then $|t|$ divides $\gcd(p, q)$. Since w is a prefix of s^ω, it is also a prefix of t^ω, hence it has a period $\gcd(p, q)$. ◀

We want to point out, based on this very simple example, several observations. First, we note the interplay between intuition brought about by the picture in the above proof, and the formal manipulation. In order to make the induction step, namely to see that both $w' \leq_p r' \cdot w$ and $w' \leq_p s \cdot w'$, one can either consult the picture, or use a formal verification which consists in the following considerations:

1. cancellation of s from $w \leq_p s \cdot w$ after substitution of both occurrences of w with $s \cdot w'$ yields $w' \leq_p s \cdot w'$;
2. cancellation of s from $w \leq_p s \cdot w$ after substitution of just the first occurrence of w with $s \cdot w'$ yields $w' \leq_p w$;
3. cancellation of s from $w \leq_p r \cdot w$ yields $w' \leq_p r' \cdot w$;
4. the latter and $w' \leq_p w$ yields $w' \leq r' \cdot w'$.

Actually, the last step still requires a simple length argument.

Although a similar point could be probably made about mathematical proofs in general, in Combinatorics on words, thanks to the elementary character of lists, the gap between the insight and the formal proof is very typical. Calibrating the right mixture of the insight and the detail, which is naturally very reader-specific, is an almost impossible task. One of the main advantages of the formalization becomes apparent here: it allows to focus on ideas while being sure that no unexpected gaps were missed.

Another lesson from this example is that it was in the context of this formalization that we realized how important and useful the equivalent characterization of periods are. More precisely, the formalization makes clear that the characterization by the period root is "the right one". In fact, we believe that the proof presented here is the shortest one available in the literature. The Periodicity lemma has many different proofs, several of them presented already in the original paper by Fine and Wilf [10]. Proofs based on the numeric definition of period by indexes (that is, by $w_i = w_{i+p}$) can be rather involved (see, for example, the basic reference monograph [26]). Our proof is close to the version in Berstel an Karhumäki [3] but without the need to deal separately with the case when the periods are not coprime.

The superiority of the periodic root definition of a period can be captured as its suitability for equational reasoning. We add another example of this phenomenon. Consider the following claim:

▶ **Lemma 3.** *If $x \cdot y = z$ and the words x and z commute, then also y and z commute.*

This is a trivial claim which would be justified (if needed) as follows:

Proof. Commuting words are powers of the same word. Canceling x from $x \cdot y = z$ therefore yields that also y is the power of the same word. ◀

This appeal to the Commutation Lemma is an almost instinctive move for a researcher in Combinatorics on Words. However, this argument does not seem to be sufficiently trivial for an automated tool (like `try0` in Isabelle). Nevertheless, the proof is the simple **`by`** `force` anyway, since Isabelle employs a different approach, which is humanly less transparent but is based on a simple manipulation of equalities.

Proof. Substitute $x \cdot y$ for z in $x \cdot z = z \cdot x$ to obtain $x \cdot x \cdot y = x \cdot y \cdot x$. Cancel x and multiply both sides by y from right to obtain $x \cdot y \cdot y = y \cdot x \cdot y$, which is the desired equality after substituting z back for $x \cdot y$. ◀

Finally, a particular challenge for the formalization of the Periodicity Lemma is the humanly obvious argument from symmetry (cf. Harrison [17]), which allows to assume that s is not longer than r. This move is sometimes dealt with in formalization by defining s_1 and r_1 as the shorter and the longer of the two words respectively, and then carrying out the proof using s_1 and r_1. This approach is nevertheless quite tedious, in particular in proofs by induction. We use a little trick to deal with this problem: the induction is made not simply on $|s| + |r|$ but rather on $|s| + |s| + |r|$. Then, considering the cases $|r| < |s|$ and $|s| \leq |r|$, the former case is covered by the induction hypothesis exactly by symmetry of s and r as in the informal proof.

The bound in the Periodicity Lemma is optimal in the following sense:

▶ **Lemma 4.** *Let p and q be positive integers such that $p \nmid q$ and $q \nmid p$. Then there is a word of length $p + q - \gcd(p, q) - 1$ that has periods p and q, and not a period $\gcd(p, q)$.*

The word from the lemma is called an FW-word(p, q) (for Fine and Wilf). With the additional requirement that it contains maximum number of distinct letters, it is unique up to renaming of letters (this property is not proved in our formalization). Such a word FW-word(p, q), which is equal to FW-word(q, p), with the maximum number of distinct letters can be constructed as follows. Use natural numbers as the alphabet, and let $[n]$ denote the word $\mathtt{0} \cdot \mathtt{1} \cdots (\mathtt{n} - \mathtt{1})$. Assume $p < q$ and let $d = \gcd(p, q)$. If $p = kd$ and $q = (k + 1)d$, $1 <$ k, then the word

$$\text{FW-word}(p, q) = [d]^{k-1} \cdot [d - 1] \cdot d \cdot [d]^{k-1} \cdot [d - 1]$$

satisfies the required conditions. Otherwise FW-word(p, q) is defined inductively as the prefix of $(\text{FW-word}(p, q - p))^\omega$ of the required length. The correctness of the construction can be proved as follows:

Proof. If $q = p + d$, then the word FW-word(p, q) defined above has the required properties as can be directly verified. If $q = p + kd$ with $1 < k$, then kd does not divide p and by induction we obtain a word v of length $q - d - 1 = (q - p) + p - d - 1 > \max(p, q - p)$, which has periods p and $q-p$ and does not have a period d. The word v is then a prefix of $(\mathrm{pref}_p v)^\omega$ and of $(\mathrm{pref}_{q-p} v)^\omega$. It is therefore also a prefix of words $\mathrm{pref}_p v \cdot v$ and $\mathrm{pref}_{q-p} v \cdot v$. Consider the prefix w of $(\mathrm{pref}_p v)^\omega$ of length $p+q-d-1 > q$. The word w has a period p since it is a prefix of $(\mathrm{pref}_p v)^\omega$, and it does not have the period d since v is a prefix of w. It remains to show that w has a period q, that is, that w is a prefix of $\mathrm{pref}_q w \cdot w$. First, note that $w = \mathrm{pref}_p v \cdot v$, hence $\mathrm{pref}_q w = \mathrm{pref}_p v \cdot \mathrm{pref}_{q-p} v$. Since v is a prefix of $\mathrm{pref}_{q-p} v \cdot v$, we have that w is a prefix of $\mathrm{pref}_p v \cdot \mathrm{pref}_{q-p} v \cdot \mathrm{pref}_p v = \mathrm{pref}_q w \cdot \mathrm{pref}_p v$, which is a prefix of $\mathrm{pref}_q w \cdot w$. ◀

We have implemented the above construction, and formalized the proof of its correctness:

```
theorem fw-word: assumes ¬ p dvd q and  ¬ q dvd p
  shows |FW-word p q| = p + q − gcd p q − 1 and
        periodN (FW-word p q) p and
        periodN (FW-word p q) q and
        ¬ periodN (FW-word p q) (gcd p q)
```

The formalized proof is relatively long (over 200 lines). This reflects the number of facts that have to be verified, including the shifty claim about the “direct verification” of the base case which spans more than half of the proof.

We thereby provide a formally verified calculation of an FW-word. Here are some sample values:

```
value FW-word 3 7
  [0, 0, 1, 0, 0, 1, 0, 0]
```

```
value FW-word 4 6
  [0, 1, 0, 2, 0, 1, 0]
```

```
value FW-word 12 18
  [0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 6, 0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4]
```

2.2 The theorem of Lyndon and Schützenberger

The very first folklore result in the basic course of Combinatorics of Words is the Commutation Lemma mentioned above, solving the equation $x \cdot y = y \cdot x$. The Commutation Lemma is easy to prove directly, but it can be also noted that the word $w = uv = vu$ has periods $|u|$ and $|v|$, and the claim follows from the Periodicity Lemma.

Moved from two to three unknowns, solving equations becomes a challenging task. Although a classification of monoids generated by three words is available (see a survey by Harju and Nowotka [15]), it is a complex one. Recall that the question about the maximal number of independent equations in three unknowns remains open as mentioned in the Introduction. From this point of view, the LSE, i.e. the equation $x^a \cdot y^b = z^c$ with $2 \leq a, b, c$ solved by Lyndon and Schützenberger in 1962, is important both historically and conceptually. As already mentioned, this equation with three unknowns was originally solved in a more general case of a free group, but it has been subsequently further investigated in free monoids, and several alternative proofs have been suggested for example by Dömösi and Horváth [8] or Harju and Nowotka [16]. It would be interesting to formalize the original proof in free groups, however this task goes beyond our present focus. We expect that the proof in free groups could not be simplified as the word variant we present below. Note that the equation can be seen as a natural follow up of the Periodicity Lemma since it deals with a special configuration of three distinct periods.

```
theorem Lyndon-Schutzenberger:
  assumes x@a·y@b = z@c and  2 ≤ a  and 2 ≤ b and 2 ≤ c
  shows  x·y = y·x and x·z = z·x and y·z = z·y
```

We present here a concise formalization of the theorem of Lyndon and Schützenberger in free monoids. We first give a full paper proof that we formalized. It is similar to the one given in [26, Section 9.2], however, the core case $c = 3$ is significantly simplified.

Proof. By symmetry, assume $|x^a| \geq |y^b|$.

The word x^a has periods $|x|$ and $|z|$. If $|x^a| \geq |z| + |x|$, then the Periodicity Lemma implies that x and z have a period dividing $|x|$ and $|z|$, which easily yields that they commute. Similarly if $|y^b| \geq |z| + |y|$.

Therefore, suppose that x^{n-1} is a proper prefix of z and y^{m-1} a proper suffix of z. Then $|x^a| < 2|z|$ and $|y^b| < 2|z|$, hence $c < 4$.

Let $c = 3$. If $a \geq 3$, then $|x^2| < |z|$ implies $|x^3| < \frac{3}{2}|z|$, contradicting the assumption $|x^a| \geq |y^b|$. Therefore $a = 2$ and $|x| \geq |y|$. There are words u, v, w such that $x = uw = wv$, $z = xu = wvu$ and $y^b = vuwvu$. From $uw = wv$ we deduce that uwv has a period $|u|$. Moreover, uwv is a factor of y^b which implies that it has a period $|y|$. Since $|y| + |u| \leq |uwv|$, the Periodicity Lemma implies that $d = \gcd(|u|, |y|)$ is a period of uwv. It is easy to see that d divides also $|v|$ and $|w|$, which implies that words u, v and w commute. Therefore also x, y and z commute.

The case $c = 2$ remains. We have $z = x^{a-1}u = wy^b$, where $uw = x$. Then $wz = (wu)^a = w^2y^b$, where wu is shorter than z. By induction on $|z|$, we obtain that w, y and wu commute. Therefore also x, y and z commute. ◀

In the formalization, we first prove that x and y commute:

```
lemma per-lemma-case:
  assumes |z| + |x| ≤ |x@a|
  shows x·y=y·x
```

The other two commutation claims, humanly obvious consequences of the first one, are proved relatively easily using auxiliary lemmas about roots formalized in our background theory.

Two of the three cases in the proof are proven as separate lemmas. Namely, the case solved by the Periodicity Lemma:

```
lemma per-lemma-case:
  assumes |z| + |x| ≤ |x@a| and x ≠ ε
  shows x·y=y·x
```

and the core case $c = 3$:

```
lemma core-case:
  assumes
    c = 3 and
    b∗|y| ≤ a∗|x| and x ≠ ε and y ≠ ε and
    lenx: a∗|x| < |z| + |x| and
    leny: b∗|y| < |z| + |y|
  shows x·y = y·x
```

It would seem natural to solve even the remaining case $c = 2$ separately, and then simply put the three cases together. However, this is not possible, since the induction, abruptly announced at the end of the paper proof, actually governs the whole proof since it covers the first two cases as well. (This is one of the typical backtracking moments of the development.) We conclude this section noting that also in this case we use a similar trick to deal with the symmetry as in the proof of the Periodicity lemma. Namely, the the induction is on $|z| + b|y|$. If $|x^a| < |y^b|$, then we switch to the symmetric case which yields the result immediately by induction.

2.3 The Graph Lemma

In order to present the third classical result, the Graph Lemma, we first need to explain its algebraic background which is covered by our second auxiliary formalized theory. It is immediate that (unlike in the free group case) submonoids of the free monoid are not always free. Consider, for example, the monoid $M = \langle\{aa, aab, baa\}\rangle$ generated by words aa, aab and baa. While $\{aa, aab, baa\}$ is its *basis*, denoted $\mathfrak{B}M$,that is, the minimal generating subset (which is unique for submonoids of the free monoid), the monoid M is not free since $aab \cdot aa = aa \cdot baa$ are two distinct decompositions of the word $aabaa$ into elements of the basis. In other words, $x \mapsto aa$, $y \mapsto baa$, $z \mapsto aab$ is a solution of the equation $x \cdot y = z \cdot x$. On the other hand, each set G of words has a *free hull* $\langle G\rangle_F$, the unique smallest free monoid containing G. This can be seen using another equivalent characterization of free monoids, namely the *stability condition*:

$$p, pw, wq, q \in M \Longrightarrow w \in M. \tag{1}$$

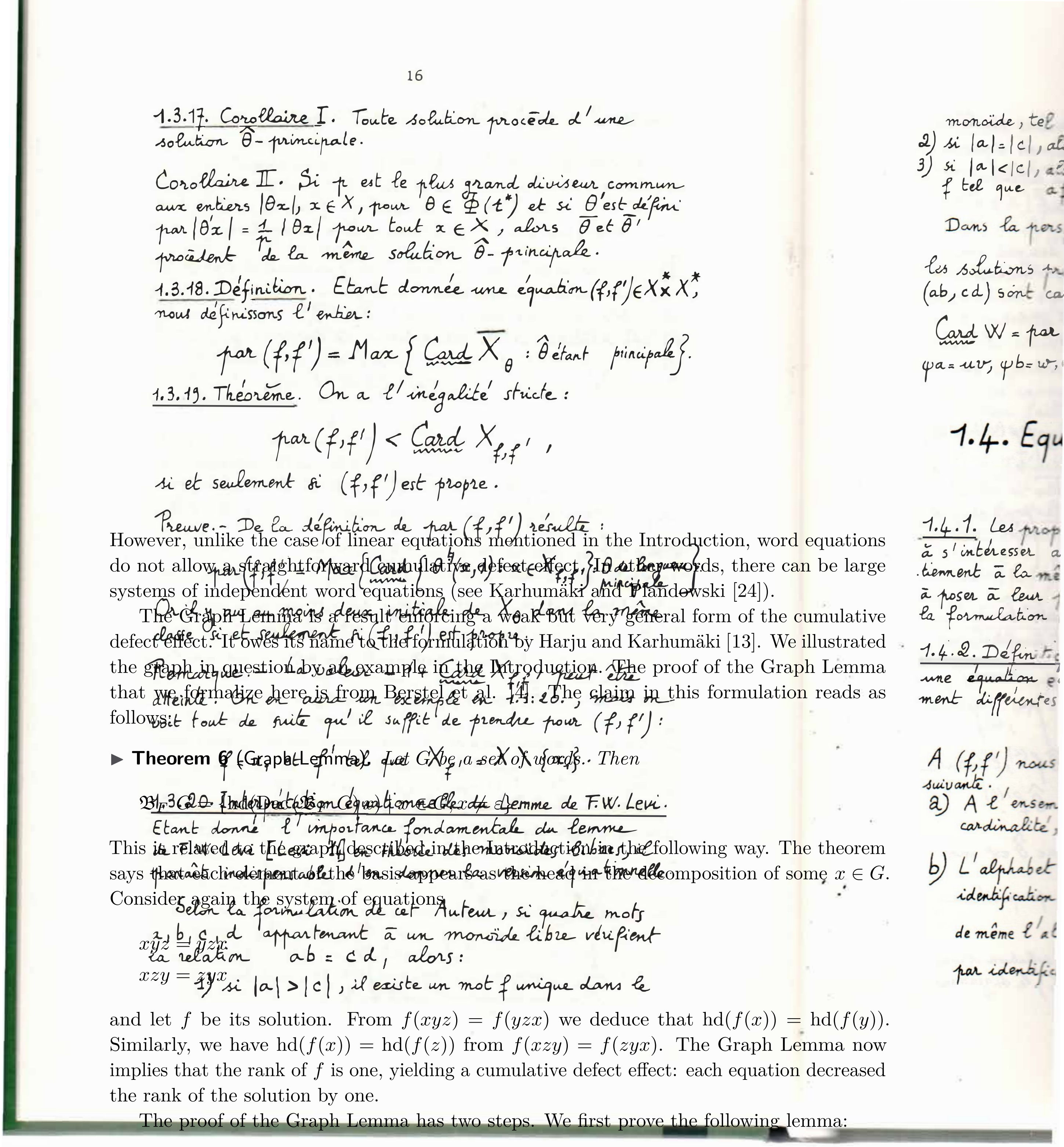
We remark that the equality $p \cdot wq = pw \cdot q$ provides a link to the equidivisibility property, another equivalent characterization of freeness mentioned in Section 1.2. Since the stability condition is obviously closed under intersection, we obtain

▶ **Lemma 5.**

$$\langle G\rangle_F = \bigcap\{M \mid G \subseteq M, M \textit{ is free}\}\,.$$

For example, the free hull of $G = \{aa, aab, baa\}$ is $\langle\{aa, b\}\rangle$. The basis of $\langle G\rangle_F$ is also called the *free basis* of G, and is denoted $\mathfrak{B}_F\ G$. The key (and defining) property of *free* monoids is uniqueness of the decomposition into elements of the basis. That is, Dec $(\mathfrak{B}_F\ G)$ is a well defined decomposition function for any G. In our example, we have Dec $(\mathfrak{B}_F\ G)\, aabaa = [aa, b, aa]$. If some set G is equal to its free basis, that is, if it is the minimal generating set of a free monoid, then G is called a *code*.

If $f : X^* \to \Sigma^*$ is a morphism (a solution of a word equation), then its *rank* is the cardinality of the free basis of the set of images $\{f(x) \mid x \in X\}$. The fact that any solution of a nontrivial equation has rank less than the number of unknowns is sometimes called “a



16

1.3.17. Corollaire I. Toute solution procède d'une solution $\hat{\theta}$-principale.

Corollaire II. Si p est le plus grand diviseur commun aux entiers $|\theta x|$, $x \in X$, pour $\theta \in \Phi(t^*)$ et si θ' est défini par $|\theta' x| = \frac{1}{p}\,|\theta x|$ pour tout $x \in X$, alors $\bar{\theta}$ et $\bar{\theta}'$ procèdent de la même solution $\hat{\theta}$-principale.

1.3.18. Définition. Etant donnée une équation $(f,f') \in X^* \times X^*$, nous définissons l'entier :

$$\mathrm{par}(f,f') = \mathrm{Max}\left\{ \mathrm{Card}\,\overline{X}_\theta : \hat{\theta} \text{ étant principale} \right\}.$$

1.3.19. Théorème. On a l'inégalité stricte :

$$\mathrm{par}(f,f') < \mathrm{Card}\,X_{f,f'},$$

si et seulement si (f,f') est propre.

Preuve.- De la définition de par(f,f') résulte :

$$\mathrm{par}(f,f') = \mathrm{Max}\left\{ \mathrm{Card}\,[illegible] \;\middle|\; \hat{\theta} \text{ principale} \right\}$$

Or il y a au moins deux initiale de X dans la même classe si et seulement si (f,f') est propre.

Remarque.- La valeur $-1 + \mathrm{Card}\,X_{f,f'}$ peut être atteinte : On en aura un exemple en 1.3.20, mais on voit tout de suite qu'il suffit de prendre pour (f,f') :

[illegible]

1.3.20. Interprétation équationnelle du Lemme de F.W. Levi.

Etant donné l'importance fondamentale du lemme [illegible] Levi [illegible] en théorie des monoïdes libres, [illegible] interprétation équationnelle.

Selon la formulation de cet Auteur, si quatre mots a, b, c, d appartenant à un monoïde libre vérifient la relation $ab = cd$, alors :

1) si $|a| > |c|$, il existe un mot f unique dans le

monoïde, tel

2) si $|a| = |c|$, a

3) si $|a| < |c|$, a

f tel que a

Dans la pers

les solutions

(ab, cd) sont ca

Card W = par

$\varphi a = uv$, $\varphi b = w$,

1.4. Equ

1.4.1. Les prop

à s'intéresser a

tiennent à la mê

à poser à leur

la formulation

1.4.2. Défini

une équation e

ment différentes

$A\,(f,f')$ nous

suivante.

a) A l'ensem

cardinalité,

b) L'alphabet

identification

de même l'al

par identific

However, unlike the case of linear equations mentioned in the Introduction, word equations do not allow a straightforward cumulative defect effect. In other words, there can be large systems of independent word equations (see Karhumäki and Plandowski [24]).

The Graph Lemma is a result enforcing a weak but very general form of the cumulative defect effect. It owes its name to the formulation by Harju and Karhumäki [13]. We illustrated the graph in question by an example in the Introduction. The proof of the Graph Lemma that we formalize here is from Berstel et al. [4]. The claim in this formulation reads as follows:

▶ **Theorem 6** (Graph Lemma). *Let G be a set of words. Then*

This is related to the graph described in the Introduction in the following way. The theorem says that each element of the basis appears as the head in the decomposition of some $x \in G$. Consider again the system of equations

$xyz = yzx$

$xzy = zyx$

and let f be its solution. From $f(xyz) = f(yzx)$ we deduce that $\mathrm{hd}(f(x)) = \mathrm{hd}(f(y))$. Similarly, we have $\mathrm{hd}(f(x)) = \mathrm{hd}(f(z))$ from $f(xzy) = f(zyx)$. The Graph Lemma now implies that the rank of f is one, yielding a cumulative defect effect: each equation decreased the rank of the solution by one.

The proof of the Graph Lemma has two steps. We first prove the following lemma:

▶ **Lemma 7.** *Let C be a code and let $b \in C$. Then also*

$$C' = \{zb^k \mid z \in C, z \neq b\}$$

is a code, and it generates the submonoid $S = \{x \in \langle C\rangle \mid \mathrm{hd}\, z \neq b\}$ of $\langle C\rangle$.

This lemma is considered to be humanly obvious. In [4] (see p. 171), this is not even formulated as a separate lemma, and the claim is justified by a simple appeal to intuition: any word not starting with b has a unique decomposition into elements of C'. On the other hand, the formalization of this claim is challenging. Indeed, the lemma actually contains (at least) the following claims:

- C' is a basis;
- C' generates S;

- C' is a code,

each of which requires nontrivial formalization effort.

Having proved Lemma 7, we can prove the Graph Lemma by contradiction. If $b \in \mathfrak{B}_F\ G$ is not a head of any decomposition, then G is contained in $\langle C' \rangle$ where

$$C' = \{zb^k \mid k \geq 0,\ z \in \mathfrak{B}_F\ X,\ z \neq b\}$$

is a code. Since $\langle C' \rangle$ does not contain b, we have $\langle C' \rangle \subsetneq \langle G \rangle_F$, a contradiction with Lemma 5.

2.4 Overview of the structure of the published formalization

The formalization is published in the Gitlab repository [36] as a part of an evolving Combinatorics on Words formalization project. The content described in this article is covered by the following five theories:

- **Basics/CoWBasic.thy**: defines basic concepts, and contains more than five hundred auxiliary lemmas (not all of them needed for the three main presented results);
- **Basics/Submonoids.thy**: defines submonoids, and contains the algebraic backbone: submonoids, fundamental properties of bases, codes and free hulls;

and three more advanced and more specific theories:

- **Basics/Periodicity_Lemma.thy**: contains the periodicity lemma, along with the proof of its optimality;
- **Basics/Lyndon_Schutzenberger.thy**: covers the Lyndon-Schützenberger theorem;
- **Graph_Lemma/Graph_Lemma.thy**: contains the Graph Lemma and its application to binary codes.

We describe the two background theories, CoWBasic and Submonoids, in more detail in the next two sections.

2.4.1 CoWBasic background theory

As already mentioned, the theory CoWBasic serves as a basis for a formalization of a Combinatorics on Words results such as the three results presented in this article. Its purpose is to cover elementary concepts (the "folklore" mentioned in Introduction) using a common notation and theorem formulation, and thus make them ready to be used by a Combinatorics on Words researcher.

CoWBasic is builds heavily on the Main's theory List and on the theory HOL-Library.Sublist. Besides the definition of the fundamental datatype list, the first mentioned theory contains many Combinatorics on Words relevant concepts such as the functions `take`, `drop`, `rotate`, `concat`, and `length`, accompanied by many relevant lemmas. The theory HOL-Library.Sublist extends the range of available tools by defining `prefix`, `longest-common-prefix`, `suffix`, and (contiguous) `sublist`, again furnished with many relevant claims. As summarized in Section 1.1, the theory first establishes some elementary prevalent notation in Combinatorics on Words. It extends the coverage of supporting claims related existing concepts ranging from observation level lemmas such as

lemma pref-drop: u $\leq$p v $\Longrightarrow$ drop p u $\leq$p drop p v

to slightly more elaborate (in terms of a formal proof) claims such as

lemma rotate-back: **obtains** m **where** rotate m (rotate n u) = u.

Most of the claims themselves can be considered quite simple, i.e., a human reader, not necessarily an expert in Combinatorics on Words, would consider them "obvious" or maybe requiring a simple argument or a picture (cf. the discussion in Section 2.1). Naturally, many of these lemmas are implicitly used in paper proofs hidden under claims such as "It easily follows". The selection of these auxiliary claims is based first on our consideration, second on the actual need in the formalization of more advanced results. As the development is an iterative process, many definitions and lemmas are results of several optimizations based on our usage experience.

In the same spirit, the theory CoWBasic introduces new concepts and supporting claims. While some of these were mentioned along with the main presented results in Section 2, we list here some most prominent other examples. We define the left quotient of a word as follows:

> **definition** left-quotient:: $'$a list $\Rightarrow$ $'$a list $\Rightarrow$ $'$a list   ((-$^{-1>}$)(-))
>   **where** left-quotient-def[simp]: left-quotient u v = (THE z. u $\cdot$ z = v).

A word is primitive if it is not a power of some other word:

> **definition** primitive :: $'$a list $\Rightarrow$ bool
>   **where** primitive u = ($\forall$ r k. r$^{@}$k = u $\longrightarrow$ k = 1)

Given a non-empty word w which is not primitive, it is natural to look for the shortest u such that $w = u^k$. Such a word is primitive, and it is the primitive root of w:

> **definition** primitive-root :: $'$a list $\Rightarrow$ $'$a list $\Rightarrow$ bool (- $\in_p$ - $*$)
>   **where** primitive-root x r = (x $\neq$ ε $\wedge$ x $\in$ r* $\wedge$ primitive r)

2.4.2 Submonoids background theory

Whereas the first auxiliary theory overlaps with existing tools, Submonoids theory develops its own tools, building on CoWBasic. Its main purpose is to cover algebraic properties of submonoids of a free monoids, a background needed for the Graph Lemma and already introduced in Section 2.3.

The first two notions were already introduced in Section 1.1, the first is the *hull*:

> **inductive-set** hull :: $'$a list set $\Rightarrow$ $'$a list set ($\langle$-$\rangle$)
>   **for** G **where**
>   emp-in: ε $\in$ $\langle$G$\rangle$
> |prod-cl: w1 $\in$ G $\Longrightarrow$ w2 $\in$ $\langle$G$\rangle$ $\Longrightarrow$ w1 $\cdot$ w2 $\in$ $\langle$G$\rangle$

and the second is *a* decomposition of a word into some sequence of words, i.e., the function `decompose` (abbreviated as Dec).

The remaining notions introduced in Section 2.3 follow. It is a noteworthy fact that their definitions are slightly different from the "paper" version above. This difference is motivated purely by a more suitable use in the formalization, based on authors' experience with primordial versions of the formalization using exactly the "paper" versions. Basis relies on the notion of a *simple element*:

```
function simple-element :: 'a list ⇒ 'a list set  ⇒ bool ( - ∈B - ) where
  simple-element b G = (b ∈ G ∧ (∀ us. us  ≠ ε ∧ us ∈ lists G ∧ concat us = b ⟶ |us|
= 1))
```

Basis is then the set of all simple elements:

```
fun basis :: 'a list set  ⇒ 'a list set (𝔅 - ) where
  basisdef:  basis G = {x. x ∈B G}
```

The definition stated above is shown as a pair of theorems – the basis is the minimal generating set:

```
theorem ⟨𝔅 G⟩ = ⟨G⟩
theorem ⟨S⟩ = ⟨G⟩ ⟹ 𝔅 G ⊆ S
```

The concept of a *code*, implemented as a locale, is formalized as

```
locale code =
  fixes 𝒞
  assumes 𝒞-is-code: xs ∈ lists 𝒞 ⟹ ys ∈ lists 𝒞 ⟹ concat xs = concat ys ⟹ xs = ys
```

and finally the inductive definition of the *free hull* reads

```
inductive-set free-hull :: 'a list set ⇒ 'a list set (⟨-⟩_F)
  for G where
    ε ∈ ⟨G⟩_F
  | free-gen-in: w ∈ G ⟹ w ∈ ⟨G⟩_F
  | w1 ∈ ⟨G⟩_F ⟹ w2 ∈ ⟨G⟩_F ⟹ w1 · w2 ∈ ⟨G⟩_F
  | p ∈ ⟨G⟩_F ⟹ q ∈ ⟨G⟩_F ⟹ p · w ∈ ⟨G⟩_F ⟹ w · q ∈ ⟨G⟩_F ⟹ w ∈ ⟨G⟩_F
```

The freeness is ensured by the last condition which is the stability condition (1). The fact that the free hull is the smallest free monoid containing the generating set is again proven as a theorem:

```
theorem free-hull-inter: ⟨G⟩_F = ⋂ {M. G ⊆ M ∧ M = ⟨M⟩_F}
```

Finally, free basis is exactly as introduced above, namely $\mathfrak{B}_F\ G = \mathfrak{B}\,\langle G\rangle_F$:

```
definition free-basis ::  'a list set ⇒ 'a list set (𝔅_F -)
  where  free-basis G ≡ 𝔅 ⟨G⟩_F
```

3 Conclusion

The aim of this paper is to introduce an ongoing formalization of Combinatorics on Words. The next step after the Lyndon-Schützenberger theorem is its natural extension obtained independently by J.-P. Spehner [33], and by E. Barbin-Le Rest, M. Le Rest [2] which claims that x^iy is the only non-trivial way (up to symmetry and conjugation) how two non-commuting words can form a non-primitive word (like z^c). The history of this result is another good motivation for our formalization project. The result, while very natural and important, has been almost forgotten (it was cited only six times before 2015). A weaker form of this result was even rediscovered in 1994 [32], and started to be referenced. One reason for this is that already this relatively simple result is very technical and difficult to read. Moreover, the paper contains several minor inaccuracies which makes the reading even more labored. This is by no means an exceptional situation in Combinatorics on Words, which testifies for a strong need of formally verified proofs in the field.

References

1 Archive of Formal Proofs. https://www.isa-afp.org/topics.html.

2 Evelyne Barbin-Le Rest and Michel Le Rest. Sur la combinatoire des codes à deux mots. *Theor. Comput. Sci.*, 41:61–80, 1985. doi:10.1016/0304-3975(85)90060-X.

3 J Berstel and J Kkarhumäki. Combinatorics on Words – a tutorial. In *Current Trends in Theoretical Computer Science*, pages 415–475. World Scientific, April 2004. doi:10.1142/9789812562494_0059.

4 J Berstel, D Perrin, J.F Perrot, and A Restivo. Sur le théorème du défaut. *Journal of Algebra*, 60(1):169–180, 1979. doi:10.1016/0021-8693(79)90113-3.

5 Jean Berstel and Dominique Perrin. The origins of combinatorics on words. *European Journal of Combinatorics*, 28(3):996–1022, 2007. doi:10.1016/j.ejc.2005.07.019.

6 Joachim Breitner. Free groups. *Archive of Formal Proofs*, 2010. , Formal proof development. URL: https://isa-afp.org/entries/Free-Groups.html.

7 Volker Diekert. Makanin's algorithm. In *Algebraic Combinatorics on Words*, Encyclopedia of Mathematics and its Applications, pages 387—442. Cambridge University Press, 2002. doi:10.1017/CBO9781107326019.013.

8 Pál Dömösi and Géza Horváth. Alternative proof of the Lyndon–Schützenberger theorem. *Theoretical Computer Science*, 366(3):194–198, 2006. Automata and Formal Languages. doi:10.1016/j.tcs.2006.08.023.

9 Georges Gonthier et al. A machine-checked proof of the odd order theorem. In *ITP*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2013.

10 N. J. Fine and H. S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–109, January 1965. doi:10.1090/S0002-9939-1965-0174934-9.

11 Georges Gonthier. Formal proof—the four-color theorem. *Notices Amer. Math. Soc.*, 55(11):1382–1393, 2008.

12 Thomas Hales et al. A formal proof of the Kepler conjecture. *Forum of Mathematics, Pi*, 5:e2, 2017. doi:10.1017/fmp.2017.1.

13 T. Harju and J. Karhumäki. On the defect theorem and simplifiability. *Semigroup Forum*, 33:199–217, 1986.

14 Tero Harju, Juhani Karhumäki, and Wojciech Plandowski. Independent systems of equations. In *Algebraic Combinatorics on Words*, Encyclopedia of Mathematics and its Applications, pages 443—471. Cambridge University Press, 2002. doi:10.1017/CBO9781107326019.014.

15 Tero Harju and Dirk Nowotka. On the independence of equations in three variables. *Theoretical Computer Science*, 307(1):139–172, 2003. WORDS. doi:10.1016/S0304-3975(03)00098-7.

16 Tero Harju and Dirk Nowotka. The equation $x^i = y^j z^k$ in a free semigroup. *Semigroup Forum*, 68(3):488–490, 2004. doi:10.1007/s00233-003-0028-6.

17 John Harrison. Without loss of generality. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 43–59, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

18 Florent Hivert et al. Coq-Combi. https://github.com/hivert/Coq-Combi, 2021.

19 Štěpán Holub. Commutation and beyond. In Srečko Brlek, Francesco Dolce, Christophe Reutenauer, and Élise Vandomme, editors, *Combinatorics on Words*, pages 1–5, Cham, 2017. Springer International Publishing.

20 Štěpán Holub and Štěpán Starosta. Binary intersection formalized. *Theor. Comput. Sci.*, to appear.

21 Štěpán Holub and Robert Veroff. Formalizing a fragment of combinatorics on words. In Jarkko Kari, Florin Manea, and Ion Petre, editors, *Unveiling Dynamics and Complexity*, pages 24–31, Cham, 2017. Springer International Publishing. `doi:10.1007/978-3-319-58741-7_3`.

22 Artur Jez. Recompression: A simple and powerful technique for word equations. *J. ACM*, 63(1):4:1–4:51, 2016. `doi:10.1145/2743014`.

23 Artur Jeż. Word equations in nondeterministic linear space. In *44th International Colloquium on Automata, Languages, and Programming*, volume 80 of *LIPIcs. Leibniz Int. Proc. Inform.*, pages Art. No. 95, 13. Schloss Dagstuhl. Leibniz-Zent. Inform., Wadern, 2017.

24 Juhani Karhumäki and Wojciech Plandowski. On the size of independent systems of equations in semigroups. *Theoretical Computer Science*, 168(1):105–119, 1996. `doi:10.1016/S0304-3975(96)00064-3`.

25 A. Lentin. *Equations dans les monoides libres.* De Gruyter Mouton, 1972. `doi:10.1515/9783111544526`.

26 M. Lothaire. *Combinatorics on words.* Cambridge Mathematical Library. Cambridge University Press, Cambridge, 1997. `doi:10.1017/CBO9780511566097`.

27 R. C. Lyndon and M. P. Schützenberger. The equation $a^m = b^n c^p$ in a free group. *Michigan Math. J.*, 9(4):289–298, December 1962. `doi:10.1307/mmj/1028998766`.

28 Gennadiy Semenovich Makanin. The problem of solvability of equations in a free semigroup. *Matematicheskii Sbornik*, 145(2):147–236, 1977.

29 Dirk Nowotka and Aleksi Saarela. One-variable word equations and three-variable constant-free word equations. *Int. J. Found. Comput. Sci.*, 29(5):935–950, 2018. `doi:10.1142/S0129054118420121`.

30 Dirk Nowotka and Aleksi Saarela. An optimal bound on the solution sets of one-variable word equations and its consequences. In *45th International Colloquium on Automata, Languages, and Programming*, volume 107 of *LIPIcs. Leibniz Int. Proc. Inform.*, pages Art. No. 136, 13. Schloss Dagstuhl. Leibniz-Zent. Inform., Wadern, 2018.

31 Aleksi Saarela. Independent systems of word equations: From Ehrenfeucht to eighteen. In Robert Mercaş and Daniel Reidenbach, editors, *Combinatorics on Words*, pages 60–67, Cham, 2019. Springer International Publishing.

32 H.J. Shyr and S.S. Yu. Non-primitive words in the language p^+q^+. *Soochow Journal of Mathematics*, 20, January 1994.

33 J.-P. Spehner. *Quelques problèmes d'extension, de conjugaison et de presentation des sous-monoïdes d'un monoïde libre.* PhD thesis, Université Paris VII, Paris, 1976.

34 Axel Thue. Über unendliche Zeichenreichen. *Skrifter: Matematisk-Naturvidenskapelig Klasse*, 1906.

35 Axel Thue. Uber die gegenseitige lage gleicher teile gewisser zeichenreihen. *Kra. Vidensk. Selsk. Skrifer, I. Mat. Nat. Kl.*, pages 1–67, 1912.

36 Štěpán Holub, Štěpán Starosta, et al. Combinatorics on words formalized (release v1.3). `https://gitlab.com/formalcow/combinatorics-on-words-formalized`, 2021.

Synthetic Undecidability and Incompleteness of First-Order Axiom Systems in Coq

Dominik Kirst
Universität des Saarlandes, Saarland Informatics Campus, Saarbrücken, Germany

Marc Hermes
Universität des Saarlandes, Department of Mathematics, Saarbrücken, Germany

Abstract

We mechanise the undecidability of various first-order axiom systems in Coq, employing the synthetic approach to computability underlying the growing Coq Library of Undecidability Proofs. Concretely, we cover both semantic and deductive entailment in fragments of Peano arithmetic (PA) and Zermelo-Fraenkel set theory (ZF), with their undecidability established by many-one reductions from solvability of Diophantine equations, i.e. Hilbert's tenth problem (H10), and the Post correspondence problem (PCP), respectively. In the synthetic setting based on the computability of all functions definable in a constructive foundation, such as Coq's type theory, it suffices to define these reductions as meta-level functions with no need for further encoding in a formalised model of computation.

The concrete cases of PA and ZF are prepared by a general synthetic theory of undecidable axiomatisations, focusing on well-known connections to consistency and incompleteness. Specifically, our reductions rely on the existence of standard models, necessitating additional assumptions in the case of full ZF, and all axiomatic extensions still justified by such standard models are shown incomplete. As a by-product of the undecidability of ZF formulated using only membership and no equality symbol, we obtain the undecidability of first-order logic with a single binary relation.

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, incompleteness, Peano arithmetic, ZF set theory, constructive type theory, Coq

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.23

Supplementary Material *Sofware*: `https://www.ps.uni-saarland.de/extras/axiomatisations/`

Acknowledgements The authors thank Andrej Dudenhefner, Yannick Forster, Lennard Gäher, Julian Rosemann, Gert Smolka, and the anonymous reviewers for helpful comments and suggestions.

1 Introduction

Being among the mainstream formalisms to underpin mathematics, first-order logic has been subject to investigation from many different perspectives since its concretisation in the late 19th century. One of them is concerned with algorithmic properties, prominently pushed by Hilbert and Ackermann with the formulation of the *Entscheidungsproblem* [16], namely the search for a decision procedure determining the formulas φ that are valid in all interpretations, usually written $\vDash \varphi$. With their groundbreaking work in the 1930s, Turing [41] and Church [6] established that such a general decision procedure cannot exist. However, this outcome can change if one considers validity of φ restricted to interpretations satisfying a given collection $\mathcal{A}$ of axioms, written $\mathcal{A} \vDash \varphi$. Already in 1929, Presburger presented a decision procedure for an axiomatisation of linear arithmetic [28] and Tarski contributed further instances with his work on Boolean algebras, real-closed ordered fields, and Euclidean geometry in the 1940s [8].

On the other hand, as soon as an axiomatisation $\mathcal{A}$ is strong enough to express computation, the undecidability proof for the Entscheidungsproblem can be replayed within $\mathcal{A}$, turning its entailed theory undecidable. Used as standard foundations for large branches of mathematics exactly due to their expressiveness, Peano arithmetic (PA) and Zermelo-Fraenkel set theory (ZF) are prime examples of such axiomatisations. In this paper, we use the Coq proof assistant [38] to mechanise the undecidability of PA and ZF based on the synthetic approach to computability results available in Coq's constructive type theory.

As is common in constructive foundations, all functions definable in Coq's type theory are effectively computable. So for instance any Boolean function on natural numbers $f : \mathbb{N} \to \mathbb{B}$ coinciding with a predicate $P \subseteq \mathbb{N}$ may be understood as a *decider* for P, even without explicitly relating f to some encoding as a Turing machine, μ-recursive function, or untyped λ-term. In this fashion, many positive notions of computability theory can be rendered *synthetically*, disposing of the need for an intermediate formal model of computation [3, 11]. Moreover, negative notions like *undecidability* are mostly established by transport along *reductions*, i.e. computable functions encoding instances of one problem in terms of another problem. Synthetically, the requirement that reductions are computable is again satisfied by construction. In fact, all problems included in the growing Coq Library of Undecidability Proofs [14] are shown undecidable in the sense that their decidability would entail the decidability of Turing machine halting by synthetic reduction from the latter.

Therefore, revisiting the undecidability of first-order axiom systems using a proof assistant like Coq is worthwhile for several reasons. First, using the synthetic approach to undecidability makes a mechanisation of these fundamental results of metamathematics pleasently feasible [11, 17]. Our mechanisations follow the informal (and instructive) practice to just define and verify reduction functions while leaving their computability implicit, with the key difference that in our constructive setting this relaxation is formally justified.

Secondly, it is well-known that undecidable axiomatisations $\mathcal{A}$ are negation-incomplete, i.e. admit φ with neither $\mathcal{A} \vDash \varphi$ nor $\mathcal{A} \vDash \neg\varphi$. By characterising $\mathcal{A} \vDash \varphi$ with an enumerable deduction system $\mathcal{A} \vdash \varphi$, this is a consequence of Post's theorem [27] stating that bi-enumerable predicates are decidable. Indeed, assuming negation-completeness, also the complement $\mathcal{A} \nvDash \varphi$ would be enumerable via $\mathcal{A} \vdash \neg\varphi$. Based on a synthetic proof of Post's theorem [3, 11], all axiomatisations shown synthetically undecidable in the present paper are incomplete in the sense that their completeness would imply the decidability of Turing machine halting. These algorithmic observations complement the otherwise notoriously hard to mechanise incompleteness proofs based on Gödel sentences [24, 25].

Lastly, undecidability of a first-order axiomatisation $\mathcal{A}$ like PA or ZF can only be established in a stronger system, since a reduction from a non-trivial problem yields the consistency of $\mathcal{A}$. Coq exhibits standard models for PA and ZF (the latter relying on mild assumptions [18]), enabling proofs of their undecidability. In fact, we sharpen the results for weak fragments Q′ and Z′ even strictly below Robinson arithmetic Q and Zermelo set theory Z, respectively, with the latter now also admitting a fully constructive standard model.

In summary, the contributions of this paper can be listed as follows:

- We extend the Coq Library of Undecidability Proofs with verified reductions to Q′, Q, PA, Z′, Z, and ZF(-regularity), regarding both Tarski semantics and natural deduction.
- We verify a translation of set theory over a convenient signature with function symbols for set operations to smaller signatures just containing one or two binary relation symbols.
- By composition, we obtain the undecidability of the Entscheidungsproblem for a single binary relation, improving on a previous mechanisation with additional symbols [11].
- By isolating a generic theorem, we obtain synthetic undecidability and incompleteness for all axiomatisations extending the fragments Q′ and Z′ w.r.t. standard models.

After a preliminary discussion of constructive type theory, synthetic undecidability, and first-order logic in Section 2, we proceed with the general results relating undecidabilitity, incompleteness, and consistency of first-order axiom systems in Section 3. This is followed by the case studies concerning arithmetical axiomatisations (Section 4) as well as set theory with Skolem functions (Section 5) and without (Section 6). We conclude in Section 7.

2 Preliminaries

In order to make this paper self-contained and accessible, we briefly outline the synthetic approach to undecidability proofs and the representation of first-order logic in constructive type theory used in previous papers.

2.1 Constructive Type Theory

We work in the framework of a constructive type theory such as the one implemented in Coq, providing a predicative hierarchy of *type universes* above a single impredicative universe $\mathbb{P}$ of *propositions*. On type level, we have the unit type $\mathbb{1}$ with a single element $* : \mathbb{1}$, the void type $\mathbb{0}$, function spaces $X \to Y$, products $X \times Y$, sums $X + Y$, dependent products $\forall(x : X).\, F\, x$, and dependent sums $\Sigma(x : X).\, F\, x$. On propositional level, these types are denoted by the usual logical notation ($\top$, $\bot$, $\to$, $\land$, $\lor$, $\forall$, and $\exists$). So-called *large elimination* from $\mathbb{P}$ into computational types is restricted, in particular case distinction on proofs of $\lor$ and $\exists$ to form computational values is disallowed. On the other hand, this restriction is permeable enough to allow large elimination of the equality predicate $= :\ \forall X.\, X \to X \to \mathbb{P}$ specified by the constructor $\forall(x : X).\, x = x$, as well as function definitions by well-founded recursion.

We employ the basic inductive types of *Booleans* ($\mathbb{B} := \mathsf{tt} \mid \mathsf{ff}$), *Peano natural numbers* ($n : \mathbb{N} := 0 \mid n + 1$), the *option type* ($\mathbb{O}(X) := \ulcorner x \urcorner \mid \emptyset$), and *lists* ($l : \mathbb{L}(X) := [\,] \mid x :: l$). We write $|l|$ for the length of a list, $l +\!\!+\, l'$ for the concatenation of l and l', $x \in l$ for membership, and just $f\,[x_1; \dots; x_n] := [f\,x_1; \dots; f\,x_n]$ for the map function. We denote by X^n the type of *vectors* $\vec{v}$ of length $n : \mathbb{N}$ over X and reuse the definitions and notations introduced for lists.

2.2 Synthetic Undecidability

The base of the synthetic approach to computability theory [30, 3] is the fact that all functions definable in a constructive foundation are computable. This fact applies to many variants of constructive type theory and we let the assumed variant sketched in the previous section be one of those. Of course, we are confident that in particular the polymorphic calculus of cumulative inductive constructions (pCuIC) [36] currently implemented in Coq satisfies this condition although there is no formal proof yet.

Now beginning with positive notions, we can introduce decidability and enumerability of decision problems synthetically, i.e. without reference to a formal model of computation:

Definition 1. *Let $P : X \to \mathbb{P}$ be a predicate over a type X.*

- *P is* decidable *if there exists $f : X \to \mathbb{B}$ s.t. $P\,x$ iff $f\,x = \mathsf{tt}$,*
- *P is* enumerable *if there exists $f : \mathbb{N} \to \mathbb{O}(X)$ s.t. $P\,x$ iff $f\,n = \ulcorner x \urcorner$ for some $n : \mathbb{N}$.*

Note that it is commonly accepted practice to mechanise decidability results in this synthetic sense (e.g. [4, 22, 31]). In the present paper, however, we mostly consider negative results in the form of undecidability of decision problems regarding first-order axiomatisations. Such negative results cannot be established in form of the actual negation of positive results, since constructive type theory is consistent with strong classical axioms turning every problem (synthetically) decidable (as witnessed by fully classical set-theoretic models, cf. [42]).

The approximation chosen in the Coq Library of Undecidability Proofs [14] is to call P (synthetically) undecidable if the decidability of P would imply the decidability of a seed problem known to be undecidable, specifically the halting problem for Turing machines. Therefore the negative notion can be turned into a positive notion, namely the existence of a computable reduction function, that again admits a synthetic rendering:

Definition 2. *Given predicates $P : X \to \mathbb{P}$ and $Q : Y \to \mathbb{P}$, we call a function $f : X \to Y$ a* (many-one) reduction *if $P\,x$ iff $Q\,(f\,x)$ for all x. We write $P \preceq Q$ if such a function exists.*

Then interpreting reductions from the halting problem for Turing machines as undecidability results is backed by the following fact:

Fact 3. *If $P \preceq Q$ and Q is decidable, then so is P.*

Such reductions have already been verified for Hilbert's tenth problem (H_{10}) [20] and the Post correspondence problem (PCP) [10] that we employ in the present paper, so by transitivity it is enough to verify continuing reductions to the axiom systems considered.

2.3 Syntax, Semantics, and Deduction Systems of First-Order Logic

We now review the representation of first-order syntax, semantics, and natural deduction systems developed in previous papers [11, 12, 17]. Beginning with the syntax, we describe terms $t : \mathbb{T}$ and formulas $\varphi : \mathbb{F}$ as inductive types over a fixed 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 $|P|$:

$$t ::= \mathsf{x}_n \mid f\,\vec{t} \quad (n : \mathbb{N}, \vec{t} : \mathbb{T}^{|f|}) \qquad \varphi ::= P\,\vec{t} \mid \bot \mid \varphi \to \psi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \forall\varphi \mid \exists\varphi \quad (\vec{t} : \mathbb{T}^{|P|})$$

Negation $\neg\varphi$ and equivalence $\varphi \leftrightarrow \psi$ are then obtained by the standard abbreviations.

In the chosen de Bruijn representation [7], a bound variable is encoded as the number of quantifiers shadowing its binder, e.g. $\forall x.\,\exists y.\,P\,x\,u \to P\,y\,v$ may be represented by $\forall\,\exists\,P\,\mathsf{x}_1\,\mathsf{x}_4 \to P\,\mathsf{x}_0\,\mathsf{x}_5$. For the sake of legibility, we write concrete formulas with named binders where instructive and defer de Bruijn representations to the Coq development. A formula with all occurring variables bound by some quantifier is called *closed*.

Next, we define the usual Tarski semantics providing an interpretation of formulas:

Definition 4. *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 $\hat{\rho} : \mathbb{T} \to D$ *and* formula satisfiability $\rho \vDash \varphi$ *by*

$$\hat{\rho}\,\mathsf{x}_n := \rho\,n \qquad \hat{\rho}\,(f\,\vec{t}) := f^{\mathcal{M}}\,(\hat{\rho}\,\vec{t}) \qquad \rho \vDash P\,\vec{t} \;:=\; P^{\mathcal{M}}(\hat{\rho}\,\vec{t})$$

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, we write $\mathcal{M} \vDash \varphi$. Moreover, given a *theory* $\mathcal{T} : \mathbb{F} \to \mathbb{P}$, we write $\mathcal{M} \vDash \mathcal{T}$ if $\mathcal{M} \vDash \psi$ for all ψ with $\mathcal{T}\,\psi$ and $\mathcal{T} \vDash \varphi$ if $\mathcal{M} \vDash \mathcal{T}$ implies $\mathcal{M} \vDash \varphi$ for all $\mathcal{M}$. The same notations apply to *(finite) contexts* $\Gamma : \mathbb{L}(\mathbb{F})$.

Finally, we represent deduction systems as inductive predicates of type $\mathbb{L}(\mathbb{F}) \to \mathbb{F} \to \mathbb{P}$. In this paper, we consider intuitionistic and classical natural deduction $\Gamma \vdash_i \varphi$ and $\Gamma \vdash_c \varphi$, respectively, and write $\Gamma \vdash \varphi$ if a statement applies to both variants. The rules characterising the two systems are standard and listed in Appendix A, here we only highlight the quantifier rules depending on the de Bruijn encoding of bound variables

$$\frac{\Gamma[\uparrow] \vdash \varphi}{\Gamma \vdash \forall\varphi}\ \text{AI} \qquad \frac{\Gamma \vdash \forall\varphi}{\Gamma \vdash \varphi[t]}\ \text{AE} \qquad \frac{\Gamma \vdash \varphi[t]}{\Gamma \vdash \exists\varphi}\ \text{EI} \qquad \frac{\Gamma \vdash \exists\varphi \quad \Gamma[\uparrow], \varphi \vdash \psi[\uparrow]}{\Gamma \vdash \psi}\ \text{EE}$$

where $\varphi[\sigma]$ denotes the *capture-avoiding instantiation* of a formula φ with a *parallel substitution* $\sigma : \mathbb{N} \to \mathbb{T}$, where the substitution $\uparrow$ maps n to x_{n+1}, where the substitution $(t;\sigma)$ maps 0 to t and $n+1$ to $\sigma\,n$, and where $\varphi[t]$ is short for $\varphi[t; (\lambda n.\,\mathsf{x}_n)]$. Extending the deduction systems to theories $\mathcal{T} : \mathbb{F} \to \mathbb{P}$, we write $\mathcal{T} \vdash \varphi$ if there is $\Gamma \subseteq \mathcal{T}$ with $\Gamma \vdash \varphi$.

Constructively, only soundness of the intuitionistic system ($\mathcal{T} \vdash_i \varphi$ implies $\mathcal{T} \vDash \varphi$) is provable without imposing a restriction on the admitted models (as done in [12]). However, it is easy to verify the usual weakening ($\Gamma \vdash \varphi$ implies $\Delta \vdash \varphi$ for $\Gamma \subseteq \Delta$) and substitution ($\Gamma \vdash \varphi$ implies $\Gamma[\sigma] \vdash \varphi[\sigma]$) properties of both variants by induction on the given derivations. The latter gives rise to named reformulations of (AI) and (EE) helpful in concrete derivations

$$\frac{\Gamma \vdash \varphi[\mathsf{x}_n]}{\Gamma \vdash \forall\varphi}\ \mathsf{x}_n \notin \Gamma, \varphi \qquad\qquad \frac{\Gamma \vdash \exists\varphi \quad \Gamma, \varphi[\mathsf{x}_n] \vdash \psi}{\Gamma \vdash \psi}\ \mathsf{x}_n \notin \Gamma, \varphi, \psi$$

where $\mathsf{x}_n \notin \Gamma$ denotes that x_n is *fresh*, i.e. does no occur unbound in any formula of Γ.

The concrete signatures used in this paper all contain a reserved binary relation symbol $\equiv$ for equality. Instead of making equality primitive in the syntax, semantics, and deduction systems, we implicitly restrict $\mathcal{M} \vDash \varphi$ to extensional models $\mathcal{M}$ interpreting $\equiv$ as actual equality $=$ and understand $\mathcal{T} \vdash \varphi$ as derivability from $\mathcal{T}$ augmented with the standard axioms characterising $\equiv$ as an equivalence relation congruent for the symbols in Σ.

3 Undecidable and Incomplete First-Order Axiom Systems

In this section, we record some general algorithmic facts concerning first-order axiomatisations and outline the common scheme underlying the undecidability proofs presented in the subsequent two sections. We fix an enumerable and discrete signature Σ for the remainder of this section and begin by introducing the central notion of axiom systems formally.

Definition 5. *We call a theory $\mathcal{A} : \mathbb{F} \to \mathbb{P}$ an* axiomatisation *if $\mathcal{A}$ is enumerable.*

Any given axiomatisation induces two related decision problems, namely semantic entailment $\mathcal{A}^{\vDash} := \lambda\varphi.\, \mathcal{A} \vDash \varphi$ and deductive entailment $\mathcal{A}^{\vdash} := \lambda\varphi.\, \mathcal{A} \vdash \varphi$. Since in our constructive setting we can show the classical deduction system $\vdash_c$ neither sound nor complete (cf. [12]), we mostly consider a combined notion of classical semantics and intuitionistic deduction:

Definition 6. *We say that a predicate $P : X \to \mathbb{P}$* reduces to $\mathcal{A}$, *written $P \preceq \mathcal{A}$, if there is a function $f : X \to \mathbb{F}$ witnessing both $P \preceq \mathcal{A}^{\vDash}$ and $P \preceq \mathcal{A}^{\vdash_i}$.*

Assuming the law of excluded middle $\mathsf{LEM} := \forall p : \mathbb{P}.\, p \lor \neg p$ would be sufficient to obtain $P \preceq \mathcal{A}^{\vdash_c}$ from $P \preceq \mathcal{A}^{\vDash}$, since then $\mathcal{A} \vdash_c \varphi$ and $\mathcal{A} \vDash \varphi$ coincide. In fact, already the soundness direction is enough for our case studies on PA and ZF, since for them it is still feasible to verify $\mathcal{A} \vdash f\,x$ given $P\,x$ by hand without appealing to completeness.

We now formulate two facts stating the well-known connections of undecidability with consistency and incompleteness for our synthetic setting. The first observation is that verifying a reduction from a non-trivial problem is at least as hard as a consistency proof.

Fact 7. *If $P \preceq \mathcal{A}^{\vdash}$ and there is x with $\neg P\,x$, then $\mathcal{A} \nvdash \bot$.*

Proof. If $f : X \to \mathbb{F}$ witnesses $P \preceq \mathcal{A}^{\vdash}$, then by $\neg P\,x$ we obtain $\mathcal{A} \nvdash f\,x$. This prohibits a derivation $\mathcal{A} \vdash \bot$ by the explosion rule (E). ◀

The second observation is a synthetic version of incompleteness for all axiomatisations strong enough to express an undecidable problem. We follow the common practice to focus on incompleteness of the classical deduction system, see Section 7.1 for a discussion.

Definition 8. *We call $\mathcal{A}$* (negation-)complete *if for all closed φ either $\mathcal{A} \vdash_c \varphi$ or $\mathcal{A} \vdash_c \neg\varphi$.*

Fact 9. *If $\mathcal{A}$ is complete with $\mathcal{A} \nvdash_c \bot$, then $\lambda\varphi.\, \mathcal{A} \vdash_c \varphi$ is decidable for closed φ. Consequently, if f witnesses $P \preceq \mathcal{A}^{\vdash_c}$ such that all $f\,x$ are closed, then P is decidable.*

Proof. By a synthetic version of Post's theorem ([11, Lemma 2.15]) it suffices to show that $\mathcal{A}^{\vdash_c}$ is bi-enumerable, i.e. both $\lambda\varphi.\, \mathcal{A} \vdash_c \varphi$ and $\lambda\varphi.\, \mathcal{A} \nvdash_c \varphi$ are enumerable, and logically decidable, i.e. $\mathcal{A} \vdash_c \varphi$ or $\mathcal{A} \nvdash_c \varphi$ for all φ. This follows by enumerability of $\vdash_c$ and since by consistency and completeness $\mathcal{A} \nvdash_c \varphi$ iff $\mathcal{A} \vdash_c \neg\varphi$. The consequence is by Fact 3. ◀

Note that this fact is an approximation of the usual incompleteness theorem in two ways. First, similar to the synthetic rendering of undecidability, axiomatisations $\mathcal{A}$ subject to a reduction $P \preceq \mathcal{A}^{\vdash_c}$ for P known to be undecidable are only shown incomplete in the sense that their completeness would imply decidability of P. Deriving an actual contradiction would rely on computability axioms (e.g. Church's thesis [19, 9] or an undecidability assumption [11]) or extraction to a concrete model (e.g. a weak call-by-value λ-calculus [13]). Secondly, the fact does not produce a witness of an independent formula the way a more informative proof based on Gödel sentences does. Also note that inconsistent axiomatisations are trivially decidable, so the requirement $\mathcal{A} \nvdash_c \bot$ is inessential (especially given Fact 7).

Next, we outline the general pattern underlying the reductions verified in this paper:

1. We choose an undecidable seed problem $P : X \to \mathbb{P}$ easy to encode in the domain of the target axiomatisations. This will be H_{10} for PA and PCP for ZF.
2. We define the translation function $X \to \mathbb{F}$ mapping instances $x : X$ to formulas φ_x in a way compact enough to be stated without developing much of the internal theory of $\mathcal{A}$.
3. We isolate a finite fragment $A \subseteq \mathcal{A}$ of axioms that suffices to implement the main argument. This yields a reusable factorisation and is easier to mechanise.
4. We verify the semantic part locally by showing for every $\mathcal{M}$ with $\mathcal{M} \vDash A$ that $P\,x$ iff $\mathcal{M} \vDash \varphi_x$. For the backwards direction, we in fact need to restrict $\mathcal{M}$ to satisfy a suitable property of standardness allowing us to reconstruct an actual solution of P.
5. We construct standard models for A and $\mathcal{A}$, possibly relying on additional assumptions.
6. We verify the deductive part by establishing that $P\,x$ implies $A \vdash \varphi_x$, closely following the semantic proof from before. The backwards direction follows from soundness.
7. We conclude the undecidability of A, $\mathcal{A}$, and any $\mathcal{B} \supseteq A$ by virtue of the following:

Theorem 10. *Let a problem $P : X \to \mathbb{P}$, an axiomatisation $\mathcal{A}$, a notion of standardness on models $\mathcal{M} \vDash \mathcal{A}$, and a function $\varphi\_ : X \to \mathbb{F}$ be given with the following properties:*

(i) *$P\,x$ implies $\mathcal{A} \vDash \varphi_x$.*
(ii) *Every standard model $\mathcal{M} \vDash \mathcal{A}$ with $\mathcal{M} \vDash \varphi_x$ yields $P\,x$.*
(iii) *$P\,x$ implies $\mathcal{A} \vdash \varphi_x$.*

Then $P \preceq \mathcal{B}$ for all $\mathcal{B} \supseteq \mathcal{A}$ admitting a standard model. Assuming LEM, *then also $P \preceq \mathcal{B}^{\vdash_c}$.*

Proof. We begin with $P \preceq \mathcal{B}^{\vDash}$. That $P\,x$ implies $\mathcal{B} \vDash \varphi_x$ is direct by (i) since every model of $\mathcal{B}$ is a model of $\mathcal{A}$. Conversely, if $\mathcal{B} \vDash \varphi_x$ then in particular the assumed standard model $\mathcal{M} \vDash \mathcal{B}$ satisfies φ_x. Thus we obtain $P\,x$ by (ii).

Turning to $P \preceq \mathcal{B}^{\vdash_i}$, the first direction is again trivial, this time by (iii) and weakening. For the converse, we assume that $\mathcal{B} \vdash_i \varphi_x$ and hence $\mathcal{B} \vDash \varphi_x$ by soundness. Thus we conclude $P\,x$ with the previous argument relying on (ii).

Finally assuming LEM, we obtain $P \preceq \mathcal{B}^{\vdash_c}$ since then already $\mathcal{B} \vdash_c \varphi_x$ implies $\mathcal{B} \vDash \varphi_x$. ◀

Of course (i) follows from (iii) via soundness, so the initial semantic verification could be eliminated from Theorem 10 and the informal strategy outlined before. However, we deem it more instructive to first present a self-contained semantic verification without the overhead introduced by working in a syntactic deduction system, mostly apparent in the Coq mechanisation. Also note that the necessity of a standard model will be no burden in the treatment of PA but in the case of ZF this will require a careful analysis of preconditions.

We end this section with the unsurprising but still important fact that we can reduce the decision problem for finite axiomatisations A to the classical Entscheidungsproblem of first-order logic concerning validity and provability in the empty context [16].

Fact 11. *For* $A : \mathbb{L}(\mathbb{F})$ *we have* $A^{\vDash} \preceq (\lambda\varphi.\ \vDash \varphi)$ *and* $A^{\vdash} \preceq (\lambda\varphi.\ \vdash \varphi)$.

Proof. It is straightforward to verify that the function $\lambda\varphi.\ \bigwedge A \to \varphi$ prefixing φ with the conjunction of all formulas in A establishes both reductions. ◀

So the reductions to finite fragments of PA and ZF presented in the next sections in particular complement the direct reductions to the Entscheidungsproblem given in [11].

4 Peano Arithmetic

We begin with a rather simple case study to illustrate our general approach to undecidability and incompleteness. For the theory of Peano arithmetic (PA) we use a signature containing symbols for the constant zero, the successor function, addition, multiplication and equality:

$$(O,\ S\_\,,\ \_\oplus\_\,,\ \_\otimes\_\,;\ \_\equiv\_)$$

The core of PA consists of axioms characterising addition and multiplication:

$\oplus$-base: $\forall x.\, O \oplus x \equiv x$ $\qquad$ $\oplus$-recursion: $\forall xy.\,(Sx) \oplus y \equiv S(x \oplus y)$

$\otimes$-base: $\forall x.\, O \otimes x \equiv O$ $\qquad$ $\otimes$-recursion: $\forall xy.\,(Sx) \otimes y \equiv y \oplus x \otimes y$

The list Q′ consisting of these four axioms is strong enough to be undecidable. Undecidability (and incompleteness) then transport in particular to the (infinite) axiomatisation PA adding

Disjointness: $\forall x.\, Sx \equiv O \to \bot$ $\qquad$ Injectivity: $\forall xy.\, Sx \equiv Sy \to x \equiv y$

and the axiom scheme of induction, which we define as a type-theoretic function on formulas:

$$\lambda\varphi.\,\varphi[O] \to (\forall x.\,\varphi[x] \to \varphi[Sx]) \to \forall x.\,\varphi[x]$$

Another typical reference point in the context of incompleteness is Robinson arithmetic Q, obtained by replacing the induction scheme by the single axiom $\forall x.\, x \equiv O \vee \exists y.\, x \equiv Sy$.

Hilbert's 10th problem (H_{10}) is concerned with the solvability of Diophantine equations and comes as a natural seed problem for showing the undecidability of PA, since the equations are a syntactic fragment of PA formulas. To be more precise, H_{10} consists of deciding whether a Diophantine equation $p = q$ has a solution in the natural numbers $\mathbb{N}$, where p, q are polynomials constructed by parameters, variables, addition, and multiplication:

$$p, q ::= \mathsf{a}_n \mid \mathsf{var}\ k \mid \mathsf{add}\ p\ q \mid \mathsf{mult}\ p\ q \qquad (n, k : \mathbb{N})$$

The evaluation $[\![p]\!]_\alpha$ of a polynomial p for a variable assignment $\alpha : \mathbb{N} \to \mathbb{N}$ is defined by

$$[\![\mathsf{a}_n]\!]_\alpha := n \qquad [\![\mathsf{var}\ k]\!]_\alpha := \alpha\, k \qquad [\![\mathsf{add}\ p\ q]\!]_\alpha := [\![p]\!]_\alpha + [\![q]\!]_\alpha \qquad [\![\mathsf{mult}\ p\ q]\!]_\alpha := [\![p]\!]_\alpha \times [\![q]\!]_\alpha$$

and a Diophantine equation $p = q$ then has a solution, if there is α such that $[\![p]\!]_\alpha = [\![q]\!]_\alpha$. Given their syntactic similarity, it is easy to encode H_{10} into PA, beginning with numerals:

Definition 12. *We define* $\nu : \mathbb{N} \to \mathbb{T}$ *recursively by* $\nu(0) := O$ *and* $\nu(n+1) := S(\nu(n))$.

We now translate polynomials into PA terms by defining $p^* : \mathbb{T}$ recursively:

$$\mathsf{a}_n{}^* := \nu(n) \qquad (\mathsf{var}\ k)^* := \mathsf{x}_k \qquad (\mathsf{add}\ p\ q)^* := p^* \oplus q^* \qquad (\mathsf{mult}\ p\ q)^* := p^* \otimes q^*$$

A Diophantine equation with greatest free variable N can now be encoded as the formula $\varphi_{p,q} := \exists^N p^* \equiv q^*$ where we use N leading existential quantifiers to internalise the solvability condition. The formula $\varphi_{p,q}$ thus asserts the existence of a solution for $p = q$ which gives us a natural encoding from Diophantine equations into PA.

We prepare the verification of the three requirements (Facts 19, 21, and 24) necessary to apply Theorem 10 with the following lemma about closed existential formulas:

Lemma 13. *If $\exists^N\varphi$ is closed, then*
(i) *$\mathcal{M} \vDash \exists^N\varphi$ iff there is $\rho : \mathbb{N} \to \mathcal{M}$ such that $\rho \vDash \varphi$,*
(ii) *$\Gamma \vdash \exists^N\varphi$ if there is $\sigma : \mathbb{N} \to \mathbb{T}$ such that $\Gamma \vdash \varphi[\sigma]$.*

Proof. We only provide some intuition for (i). For the implication from left to right, the assumption $\mathcal{M} \vDash \exists^N\varphi$ gives us $x_1, \dots, x_N : \mathcal{M}$ such that $\forall \rho.\, x_1; \dots; x_N; \rho \vDash \varphi$, so in particular we have $\rho' \vDash \varphi$ for $\rho' := x_1; \dots; x_N; (\lambda x.\, O^{\mathcal{M}})$, showing the claim. For the other implication, we get ρ with $\rho \vDash \varphi$. By setting $\rho' := \lambda x.\, \rho(x+N)$ we have $\rho = \rho(0); \dots; \rho(N); \rho'$ and hence there are $x_1, \dots, x_N : \mathcal{M}$ such that $x_1; \dots; x_N; \rho' \vDash \varphi$. Since φ has at most N free variables, ρ' can be exchanged with any other $\tau : \mathbb{N} \to \mathcal{M}$. ◀

By Lemma 13, showing $\varphi_{p,q}$ is equivalent to finding a satisfying environment $\rho : \mathbb{N} \to \mathcal{M}$ for $p^* \equiv q^*$ in a model $\mathcal{M}$ or deductively showing that a substitution $\sigma : \mathbb{N} \to \mathbb{T}$ solves it. This enables us to transport a solution for $p = q$ to both the model and the deduction system.

We now verify the semantic part of the reduction for the axiomatic fragment Q'. To this end, we fix a model $\mathcal{M} \vDash \mathsf{Q}'$ for the next definitions and lemmas.

Definition 14. *We define $\mu : \mathbb{N} \to \mathcal{M}$ by $\mu(0) := O^{\mathcal{M}}$ and $\mu(n+1) := S^{\mathcal{M}}(\mu(n))$.*

The axioms in Q' are sufficient to prove that μ is a homomorphism.

Lemma 15. *For $n, m : \mathbb{N}$, $\mu(n+m) = \mu(n) \oplus^{\mathcal{M}} \mu(m)$ and $\mu(n+m) = \mu(n) \otimes^{\mathcal{M}} \mu(m)$.*

Proof. The proof for addition is done by induction on $n : \mathbb{N}$ and using the axioms for addition in Q'. The proof for multiplication is done in the same fashion, using the axioms for multiplication and the previous result for addition. ◀

Lemma 16. *For any $\rho : \mathbb{N} \to \mathcal{M}$ and $n : \mathbb{N}$ we have $\hat{\rho}\,(\nu(n)) = \mu(n)$.*

Given an assignment $\alpha : \mathbb{N} \to \mathbb{N}$, we can transport the evaluation of a polynomial $[\![p]\!]_\alpha$ to any Q' model by applying μ. The homomorphism property of μ now makes it easy to verify that we get the same result by evaluating the encoded version p^* with the composition $\mu \circ \alpha$.

Lemma 17. *For any polynomial p and $\alpha : \mathbb{N} \to \mathbb{N}$ we have $\widehat{(\mu \circ \alpha)}(p^*) = \mu([\![p]\!]_\alpha)$.*

Proof. By induction on p, using Lemmas 16 and 17. ◀

Corollary 18. *If $p = q$ has a solution α, then in any Q' model $(\mu \circ \alpha) \vDash p^* \equiv q^*$.*

Proof. We have $\mu([\![p]\!]_\alpha) = \mu([\![q]\!]_\alpha) \overset{L.17}{\Longrightarrow} \widehat{(\mu \circ \alpha)}(p^*) = \widehat{(\mu \circ \alpha)}(q^*) \implies (\mu \circ \alpha) \vDash p^* \equiv q^*$. ◀

Fact 19. *If $p = q$ has a solution, then $\mathsf{Q}' \vDash \varphi_{p,q}$.*

Proof. Let α be the solution of $p = q$, then $(\mu \circ \alpha) \vDash p^* \equiv q^*$ holds by Corollary 18 and since $\exists^N p^* \equiv q^*$ is closed by construction, the goal follows by Lemma 13. ◀

Turning to the converse direction, the natural choice for a standard model is the type $\mathbb{N}$.

Lemma 20. *$\mathbb{N}$ is a model of Q', Q, and PA.*

It is straightforward to extract a solution of $p = q$ if $\mathbb{N} \vDash \varphi_{p,q}$ using the previous lemmas.

Fact 21. *If $\mathbb{N} \vDash \varphi_{p,q}$ then $p = q$ has a solution.*

Proof. By assumption we have $\mathbb{N} \vDash \varphi_{p,q}$ which by Lemma 13 gives us $\alpha : \mathbb{N} \to \mathbb{N}$ with

$$\alpha \vDash p^* \equiv q^* \implies \widehat{(\mu \circ \alpha)}(p^*) = \widehat{(\mu \circ \alpha)}(q^*) \overset{L.17}{\implies} \mu([\![p]\!]_\alpha) = \mu([\![q]\!]_\alpha).$$

Since over $\mathbb{N}$ the function μ is simply the identity, we conclude $[\![p]\!]_\alpha = [\![q]\!]_\alpha$. ◀

The deductive part of the reduction can be shown analogously to Fact 19, encoding the proofs of all intermediate results as ND derivations. We just list the relevant statements and refer to the Coq code for more detail.

Lemma 22. *For $n, m : \mathbb{N}$, $\mathsf{Q}' \vdash \nu(n + m) \equiv \nu(n) \oplus \nu(m)$ and $\mathsf{Q}' \vdash \nu(n \times m) \equiv \nu(n) \otimes \nu(m)$.*

Lemma 23. *If $p = q$ has a solution α, then we can deduce $\mathsf{Q}' \vdash (p^* \equiv q^*)[\nu \circ \alpha]$.*

Fact 24. *If $p = q$ has a solution then $\mathsf{Q}' \vdash \varphi_{p,q}$.*

Now we have all facts in place to verify the reductions with Theorem 10.

Theorem 25. *$\mathsf{H}_{10} \preceq \mathsf{Q}'$, $\mathsf{H}_{10} \preceq Q$, and $\mathsf{H}_{10} \preceq \mathsf{PA}$.*

Proof. Since $\mathbb{N}$ is a standard model for Q', Q, and PA, the claims follow by Theorem 10 since we have shown the three necessary conditions in Facts 19, 21, and 24. ◀

As a consequence of these reductions, we can conclude incompleteness as follows:

Theorem 26. *Assuming LEM, completeness of any extension $\mathcal{A} \supseteq \mathsf{Q}'$ satisfied by the standard model $\mathbb{N}$ would imply the decidability of the halting problem of Turing machines.*

Proof. By Theorems 10 and 25, Fact 9, and the reductions verified in [20]. ◀

We close this section with a remark on separating models of Q', Q, and PA. For any $n : \mathbb{N}$, the quotient $\mathbb{Z}/n\mathbb{Z}$ is a model of Q'. So in particular Q' admits the trivial model and can hence be completed with $\forall xy.\, x \equiv y$, separating it from both Q and PA since they only admit infinite models and are essentially incomplete. A well-known model separating Q and PA is obtained by extending $\mathbb{N}$ to $\mathbb{N}^\infty$ with a maximal number ∞.

5 ZF Set Theory with Skolem Functions

Turning to set theory, we first work in a rich signature providing function symbols for the axiomatic operations of ZF. Concretely, for the rest of this section we fix the signature

$$\Sigma := (\emptyset,\ \{\_,\_\},\ \bigcup\_,\ \mathcal{P}(\_),\ \omega\ ;\ \_ \equiv \_,\ \_ \in \_)$$

with function symbols denoting the empty set, pairing, union, power set, the set of natural numbers, next to the usual relation symbols for equality and membership. Using such Skolem functions for axiomatic and other definable operations is common practice in set-theoretic literature and eases the definition and verification of the undecidability reduction in our case.

That the undecidability result can be transported to minimal signatures just containing equality and membership, or even just the latter, is subject of the next section.

We do not list all axioms in detail but refer the reader to Appendix B, the Coq code, and standard literature (eg. [35]). The only point worth mentioning again is the representation of axiom schemes as functions $\mathbb{F} \to \mathbb{F}$, for instance by the separation scheme expressed as

$$\lambda\varphi.\, \forall x.\, \exists y.\, \forall z.\, z \in y \;\leftrightarrow\; z \in x \wedge \varphi[x].$$

We then distinguish the following axiomatisations:

- Z' is the list containing extensionality and the specifications of the five function symbols.
- Z is the (infinite) theory obtained by adding all instances of the separation scheme.
- ZF is the theory obtained by further adding all instances of the replacement scheme.

Note that in ZF we do not include the axiom of regularity since this would force the theory classical and would require to extend Coq's type theory even further to obtain a model [23]. Alternatively, one could add the more constructive axiom for ϵ-induction, but instead we opt for staying more general and just leave the well-foundedness of sets unspecified.

Following the general outline for the undecidability proofs in this paper, we first focus on verifying a reduction to the base theory Z' and then extend to the stronger axiomatisations by use of Theorem 10. As a seed problem for this reduction, we could naturally pick just any decision problem since set theory is a general purpose foundation expressive enough for most standard mathematics. However, the concrete choice has an impact on the mechanisation overhead, where formalising Turing machine halting directly is tricky enough in Coq's type theory itself, and even a simple problem like H_{10} used in the previous section would presuppose a modest development of number theory and recursion in the axiomatic framework. We therefore base our reduction to Z' on the Post correspondence problem (PCP) which has a simple inductive characterisation expressing a matching problem given a finite stack S of pairs (s, t) of Boolean strings:

$$\frac{(s,t) \in S}{S \rhd (s,t)} \qquad\qquad \frac{S \rhd (u,v) \quad (s,t) \in S}{S \rhd (su, tv)} \qquad\qquad \frac{S \rhd (s,s)}{\mathsf{PCP}\, S}$$

Informally, S is used to derive pairs (s, t), written $S \rhd (s, t)$ by repeatedly appending the pairs from the stack componentwise in any order or multitude. The instance S admits a solution, written $\mathsf{PCP}\, S$, if a matching pair (s, s) can be derived by this procedure.

Encoding data like numbers and Booleans in set-theoretic terms is standard, using the usual derived notations for binary union $x \cup y$, singletons $\{x\}$, and ordered pairs (x, y):

- Numbers: $\overline{0} := \emptyset$ and $\overline{n+1} := \overline{n} \cup \{\overline{n}\}$
- Booleans: $\overline{\mathsf{tt}} := \{\emptyset\}$ and $\overline{\mathsf{ff}} := \emptyset$
- Strings: $\overline{b_1, \dots, b_n} := (\overline{b_1}, (\dots(\overline{b_n}, \emptyset)\dots))$
- Stacks: $\overline{S} := \{(\overline{s_1}, \overline{t_1}), \dots, (\overline{s_m}, \overline{t_m})\}$

Starting with an informal idea, the solvability condition of PCP can be directly expressed in set theory by just asserting the existence of a set encoding a match for S:

$$\exists x.\, (x,x) \in \bigcup_{k \in \omega} \overline{S}^k \quad \text{where} \quad \overline{S}^0 = \overline{S} \quad \text{and} \quad \overline{S}^{k+1} = S \boxtimes \overline{S}^k = \bigcup_{s/t \in S} \{(\overline{s}x, \overline{t}y) \mid (x,y) \in \overline{S}^k\}$$

Unfortunately, formalizing this idea is not straightforward, since the iteration operation $\overline{S}^k$ is described by recursion on set-theoretic numbers $k \in \omega$ missing a native recursion principle akin to the one for type-theoretic numbers $n : \mathbb{N}$. Such a recursion principle can of course be derived but in our case it is simpler to inline the main construction.

The main construction used in the recursion theorem for ω is a sequence of finite approximations f accumulating the first k steps of the recursive equations. Since in our case we do not need to form the limit of this sequence requiring the approximations to agree, it suffices to ensure that at least the first k steps are contained without cutting off, namely

$$f \gg k := (\emptyset, \overline{S}) \in f \land \forall (l, B) \in f.\, l \in k \to (l \cup \{l\}, S \boxtimes B) \in f$$

where we reuse the operation $S \boxtimes B$ appending the encoded elements of the list S component-wise to the elements of the set B as specified above. Note that this operation is not really definable as a function $\mathbb{L}(\mathbb{B}) \to \mathbb{T} \to \mathbb{T}$ and needs to be circumvented by quantifying over candidate sets satisfying the specification. However, for the sake of a more accessible explanation, we leave this subtlety to the Coq code and continue using $S \boxtimes B$ as a function.

Now solvability of S can be expressed formally as the existence of a functional approximation f of length k containing a match (x, x):

$$\varphi_S := \exists k, f, B, x.\, k \in \omega \land (\forall (l, B), (l, B') \in f.\, B = B') \land f \gg k \land (k, B) \in f \land (x, x) \in B$$

We proceed with the formal verification of the reduction function $\lambda S.\, \varphi_S$ by proving the three facts necessary to apply Theorem 10. Again beginning with the semantic part for clarity, we fix a model $\mathcal{M} \vDash \mathsf{Z}'$ for the next lemmas in preparation of the facts connecting $\mathsf{PCP}\, S$ with $\mathcal{M} \vDash \varphi_S$. We skip the development of basic set theory in $\mathcal{M}$ reviewable in the Coq code and only state lemmas concerned with encodings and the reduction function:

Lemma 27. *Let $n, m : \mathbb{N}$ and $s, t : \mathbb{L}(\mathbb{B})$ be given, then the following hold:*

(i) $\mathcal{M} \vDash \overline{n} \in \omega$
(ii) $\mathcal{M} \vDash \overline{n} \notin \overline{n}$
(iii) $\mathcal{M} \vDash \overline{n} \equiv \overline{m}$ *implies* $n = m$
(iv) $\mathcal{M} \vDash \overline{s} \equiv \overline{t}$ *implies* $s = t$

Proof.
(i) By induction on n, employing the infinity axiom characterising ω.
(ii) Again by induction on n, using the fact that numerals $\overline{n}$ are transitive sets.
(iii) By trichotomy we have $n < m$, $m < n$, or $n = m$ as desired. If w.l.o.g. it were $n < m$, then $\mathcal{M} \vDash \overline{n} \in \overline{m}$ would follow by structural induction on the derivation of $n < m$. But then the assumption $\mathcal{M} \vDash \overline{n} \equiv \overline{m}$ would yield $\mathcal{M} \vDash \overline{n} \in \overline{n}$ in conflict with (ii).
(iv) By induction on the given strings, employing injectivity of the encoding of Booleans. ◀

In order to match the structure of iterated derivations encoded in φ_S, we reformulate $S \rhd (s, t)$ by referring to the composed derivations S^n of length n, now definable by recursion on $n : \mathbb{N}$ via $S^0 := S$ and $S^{n+1} := S \boxtimes S^n$ reusing the operation $\boxtimes$ for lists as expected.

Lemma 28. *$S \rhd (s, t)$ iff there is $n : \mathbb{N}$ with $(s, t) \in S^n$.*

Then the iterations S^n can be encoded as set-level functions $f_S^n := \{(\emptyset, \overline{S}), \dots, (\overline{n}, \overline{S^n})\}$ that are indeed recognised by the model $\mathcal{M}$ as correct approximations:

Lemma 29. *For every $n : \mathbb{N}$ we have $\mathcal{M} \vDash f_S^n \gg \overline{n}$.*

Proof. In this proof we work inside of $\mathcal{M}$ to simplify intermediate statements. For the first conjunct, we need to show that $(\emptyset, \overline{S}) \in f_S^n$ which is straightforward since $(\emptyset, \overline{S}) \in f_S^0$ and $f_S^m \subseteq f_S^n$ whenever $m \le n$. Regarding the second conjunct, we assume $(k, B) \in f_S^n$ with $k \in \overline{n}$ and need to show $(k \cup \{k\}, S \boxtimes B) \in f_S^n$. From $(k, B) \in f_S^n$ we obtain that there is m with $k = \overline{m}$ and $B = \overline{S^m}$. Then from $\overline{m} \in \overline{n}$ and hence $m < n$ we deduce that also $(\overline{m+1}, \overline{S^{m+1}}) \in f_S^n$. The claim follows since $\overline{m+1} = k \cup \{k\}$ and

$$\overline{S^{m+1}} = \overline{S \boxtimes S^n} = S \boxtimes \overline{S^n} = S \boxtimes B$$

using that the $\boxtimes$ operation on lists respecitively sets interacts well with string encodings. ◀

With these lemmas in place, we can now conclude the first part of the semantic verification.

Fact 30. *If* $\mathsf{PCP}\, S$ *then* $\mathsf{Z}' \vDash \varphi_S$.

Proof. Assuming $\mathsf{PCP}\, S$, there are $s : \mathbb{L}(\mathbb{B})$ and $n : \mathbb{N}$ with $(s,s) \in S^n$ using Lemma 28. Now to prove $\mathsf{Z}' \vDash \varphi_S$ we assume $\mathcal{M} \vDash \mathsf{Z}'$ and need to show $\mathsf{Z}' \vDash \varphi_S$. Instantiating the leading existential quantifiers of φ_S with $\overline{n}$, f_S^n, $\overline{S^n}$, and $\overline{s}$ leaves the following facts to verify:

- $\mathcal{M} \vDash \overline{n} \in \omega$, immediate by (i) of Lemma 27.
- Functionality of f_S^n, straightforward by construction of f_S^n.
- $\mathcal{M} \vDash f_S^n \gg \overline{n}$, immediate by Lemma 29.
- $\mathcal{M} \vDash (\overline{n}, \overline{S^n}) \in f_S^n$, again by construction of f_S^n.
- $\mathcal{M} \vDash (\overline{s}, \overline{s}) \in \overline{S^n}$, by the assumption $(s,s) \in S^n$. ◀

For the converse direction, we again need to restrict to models $\mathcal{M}$ only containing standard natural numbers, i.e. satisfying that any $k \in \omega$ is the numeral $k = \overline{n}$ for some $n : \mathbb{N}$. Then the internally recognised solutions correspond to actual external solutions of PCP.

Lemma 31. *If in a standard model* $\mathcal{M}$ *there is a functional approximation* $f \gg k$ *for* $k \in \omega$ *with* $(k, B) \in f$, *then for all* $p \in B$ *there are* $s, t : \mathbb{L}(\mathbb{B})$ *with* $p = (\overline{s}, \overline{t})$ *and* $S \rhd (s,t)$.

Proof. Since $\mathcal{M}$ is standard, there is $n : \mathbb{N}$ with $k = \overline{n}$, so we have $f \gg \overline{n}$ and $(\overline{n}, B) \in f$. In any model with $f \gg \overline{n}$ we can show that $(\overline{k}, \overline{S^k}) \in f$ by induction on k, so in particular $(\overline{n}, \overline{S^n}) \in f$ in $\mathcal{M}$. But then by functionality of f it must be $B = \overline{S^n}$, so for any $p \in B$ we actually have $p \in \overline{S^n}$ for which it is easy to extract $s, t : \mathbb{L}(\mathbb{B})$ with $p = (\overline{s}, \overline{t})$ and $(s,t) \in S^n$. We then conclude $S \rhd (s,t)$ with Lemma 28. ◀

Fact 32. *Every standard model* $\mathcal{M} \vDash \mathsf{Z}'$ *with* $\mathcal{M} \vDash \varphi_S$ *yields* $\mathsf{PCP}\, S$.

Proof. A standard model of Z' with $\mathcal{M} \vDash \varphi_S$ yields a functional approximation $f \gg k$ for $k \in \omega$ with some $(k, B) \in f$ and $(x, x) \in B$. Then by Lemma 31 there are $s, t : \mathbb{L}(\mathbb{B})$ with $(x,x) = (\overline{s}, \overline{t})$ and $S \rhd (s,t)$. By the injectivity of ordered pairs and string encodings ((iv) of Lemma 27) we obtain $s = t$ and thus $S \rhd (s,s)$. ◀

Finally, we just record the fact that the semantic argument in Fact 32 can be repeated deductively with an analogous intermediate structure.

Fact 33. *If* $\mathsf{PCP}\, S$ *then* $\mathsf{Z}' \vdash \varphi_S$.

With the three facts verifying φ_S in place, we conclude reductions as follows:

Theorem 34. *We have the following reductions.*

- $\mathsf{PCP} \preceq \mathsf{Z}'$, *provided a standard model of* Z' *exists.*
- $\mathsf{PCP} \preceq \mathsf{Z}$, *provided a standard model of* Z *exists.*
- $\mathsf{PCP} \preceq \mathsf{ZF}$, *provided a standard model of* ZF *exists.*

Proof. By Facts 30, 32, and 33 as well as Theorem 10. ◀

In a previous paper [18] based on Aczel's sets-as-trees interpretation [1, 42, 2], we analyse assumptions necessary to obtain models of higher-order set theories in Coq's type theory. The two relevant axioms concerning the type $\mathcal{T}$ of well-founded trees can be formulated as the extensionality of classes, i.e. unary predicates, on trees (CE), and the existence of a description operator for isomorphism classes $[t]_\approx$ of trees (TD):

$$\mathsf{CE} := \forall (P, P' : \mathcal{T} \to \mathbb{P}).\, (\forall t.\, P\,t \leftrightarrow P'\,t) \to P = P'$$
$$\mathsf{TD} := \exists (\delta : (\mathcal{T} \to \mathbb{P}) \to \mathcal{T}).\, \forall P.\, (\exists t.\, P = [t]_\approx) \to P\,(\delta\, P)$$

Then Theorem 34 can be reformulated as follows.

Corollary 35. CE *implies both* $\mathsf{PCP} \preceq \mathsf{Z}'$ *and* $\mathsf{PCP} \preceq \mathsf{Z}$*, and* $\mathsf{CE} \wedge \mathsf{TD}$ *implies* $\mathsf{PCP} \preceq \mathsf{ZF}$*.*

Proof. By Fact 5.4 and Theorem 5.9 of [18] CE and $\mathsf{CE} \wedge \mathsf{TD}$ yield models of higher-order Z and ZF set theory, respectively. It is easy to show that they are standard models and satisfy the first-order axiomatisations Z and ZF. ◀

Note that assuming CE to obtain a model of higher-order Z is unnecessary if we allow the interpretation of equality by any equivalence relation congruent for membership, backed by the fully constructive model given in Theorem 4.6 of [18]. This variant is included in the Coq development but we focus on the simpler case of extensional models in this text.

As a consequence of these reductions, we can conclude the incompleteness of ZF.

Theorem 36. *Assuming* LEM*, completeness of any extension* $\mathcal{A} \supseteq \mathsf{Z}'$ *satisfied by a standard model would imply the decidability of the halting problem of Turing machines.*

Proof. By Corollary 35, Theorem 10, Fact 9, and the reductions verified in [10]. ◀

6 ZF Set Theory without Skolem Functions

We now work in the signature $\tilde{\Sigma} := (\_ \equiv \_, \_ \in \_)$ only containing equality and membership. To express set theory in this syntax, we reformulate the axioms specifying the Skolem symbols used in the previous signature Σ to just assert the existence of respective sets, for instance:

$$\begin{array}{rcrcl} \emptyset & : & \forall x.\, x \notin \emptyset & \rightsquigarrow & \exists u.\, \forall x.\, x \notin u \\ \mathcal{P}(x) & : & \forall xy.\, y \in \mathcal{P}(x) \leftrightarrow y \subseteq x & \rightsquigarrow & \forall x.\, \exists u.\, \forall y.\, y \in u \leftrightarrow y \subseteq x \end{array}$$

In this way we obtain axiomatisations $\tilde{\mathsf{Z}}'$, $\tilde{\mathsf{Z}}$, and $\widetilde{\mathsf{ZF}}$ as the respective counterparts of Z', Z, and ZF. In this section, we show that these symbol-free axiomatisations admit the same reduction from PCP.

Instead of reformulating the reduction given in the previous section to the smaller signature, which would require us to replace the natural encoding of numbers and strings as terms by a more obscure construction, we define a general translation $\tilde{\varphi} : \mathbb{F}_{\tilde{\Sigma}}$ of formulas $\varphi : \mathbb{F}_{\Sigma}$. We then show that $\tilde{\mathsf{Z}}' \vDash \tilde{\varphi}$ implies $\mathsf{Z}' \vDash \varphi$ (Fact 40) and that $\mathsf{Z}' \vdash \varphi$ implies $\tilde{\mathsf{Z}}' \vdash \tilde{\varphi}$ (Fact 43), which is enough to deduce the undecidability of $\tilde{\mathsf{Z}}'$, $\tilde{\mathsf{Z}}$, and $\widetilde{\mathsf{ZF}}$ (Theorem 44).

The informal idea of the translation function is to replace terms $t : \mathbb{T}_{\Sigma}$ by formulas $\varphi_t : \mathbb{F}_{\tilde{\Sigma}}$ characterising the index x_0 to behave like t, for instance:

$$\mathsf{x}_n \rightsquigarrow \mathsf{x}_0 \equiv \mathsf{x}_{n+1} \qquad \emptyset \rightsquigarrow \forall \mathsf{x}_0 \notin \mathsf{x}_1 \qquad \mathcal{P}(t) \rightsquigarrow \exists\, \varphi_t[\mathsf{x}_0; \uparrow^2] \wedge \forall \mathsf{x}_0 \in \mathsf{x}_2 \leftrightarrow \mathsf{x}_0 \subseteq \mathsf{x}_1$$

The formula expressing $\mathcal{P}(t)$ first asserts that there is a set satisfying φ_t (where the substitution $\uparrow^n$ shifts all indices by n) and then characterises x_0 (appearing as x_2 given the two quantifiers) as its power set. Similarly, formulas are translated by descending recursively to the atoms, which are replaced by formulas asserting the existence of characterised sets being in the expected relation, for instance:

$$t \in t' \rightsquigarrow \exists\, \varphi_t[\mathsf{x}_0; \uparrow^2] \wedge \exists\, \varphi_{t'}[\mathsf{x}_0; \uparrow^3] \wedge \mathsf{x}_1 \in \mathsf{x}_0$$

We now verify that the translation $\tilde{\varphi}$ satisfies the two desired facts, starting with the easier semantic implication. To this end, we denote by $\tilde{\mathcal{M}}$ the $\tilde{\Sigma}$-model obtained from a Σ-model $\mathcal{M}$ by forgetting the interpretation of the function symbols not present in $\tilde{\Sigma}$. Then for a model $\mathcal{M} \vDash \mathsf{Z}'$, satisfiability is preserved for translated formulas, given that the term characterisations are uniquely satisfied over the axioms of Z':

Lemma 37. *Given $\mathcal{M} \vDash \mathsf{Z}'$, $t : \mathbb{T}$, $\rho : \mathbb{N} \to \mathcal{M}$, and $x : \mathcal{M}$ we have $x = \hat{\rho}\, t$ iff $(x; \rho) \vDash_{\tilde{\mathcal{M}}} \varphi_t$.*

Proof. By induction on t with x generalised. We only consider the cases x_n and $\emptyset$:

- We need to show $x = \hat{\rho}\,\mathsf{x}_n$ iff $(x; \rho) \vDash_{\tilde{\mathcal{M}}} \mathsf{x}_0 \equiv \mathsf{x}_{n+1}$ which is immediate by definition.
- First assuming $x = \emptyset$, we need to show that $\forall y.\, y \notin x$, which is immediate since $\mathcal{M}$ satisfies the empty set axiom. Conversely assuming $\forall y.\, y \notin x$ yields $x = \emptyset$ by using the extensionality axiom also satisfied by $\mathcal{M}$. ◀

Lemma 38. *Given $\mathcal{M} \vDash \mathsf{Z}'$, $\varphi : \mathbb{F}$, and $\rho : \mathbb{N} \to \mathcal{M}$ we have $\rho \vDash_{\mathcal{M}} \varphi$ iff $\rho \vDash_{\tilde{\mathcal{M}}} \tilde{\varphi}$.*

Proof. By induction on φ with ρ generalised, all cases but atoms are directly inductive. Considering the case $t \in t'$, we first need to show that if $\hat{\rho}\, t \in \hat{\rho}\, t'$, then there are x and x' with $x \in x'$ satisfying φ_t and $\varphi_{t'}$, respectively. By Lemma 37 the choice $x := \hat{\rho}\, t$ and $x' := \hat{\rho}\, t'$ is enough. Now conversely, if there are such x and x', by Lemma 37 we know that $x = \hat{\rho}\, t$ and $x' = \hat{\rho}\, t'$ and thus conclude $\hat{\rho}\, t \in \hat{\rho}\, t'$. The case of $t \equiv t'$ is analogous. ◀

Then the desired semantic implication follows since pruned models $\tilde{\mathcal{M}}$ satisfy $\tilde{\mathsf{Z}}'$:

Lemma 39. *If $\mathcal{M} \vDash \mathsf{Z}'$ then $\tilde{\mathcal{M}} \vDash \tilde{\mathsf{Z}}'$.*

Proof. We only need to consider the axioms concerned with set operations, where we instantiate the existential quantifiers introduced in $\tilde{\mathsf{Z}}'$ with the respective operations available in $\mathcal{M}$. For instance, to show $\tilde{\mathcal{M}} \vDash \exists u. \forall x.\, x \notin u$ it suffices to show that $\forall x.\, x \notin \emptyset$ in $\tilde{\mathcal{M}}$, which is exactly the empty set axiom satisfied by $\mathcal{M}$. ◀

Fact 40. $\tilde{\mathsf{Z}}' \vDash \tilde{\varphi}$ *implies* $\mathsf{Z}' \vDash \varphi$.

Proof. Straightforward by Lemmas 38 and 39. ◀

We now turn to the more involved deductive verification of the translation, beginning with the fact that $\tilde{\mathsf{Z}}'$ proves the unique existence of sets satisfying the term characterisations:

Lemma 41. *For all $t : \mathbb{T}$ we have $\tilde{\mathsf{Z}}' \vdash \exists\, \varphi_t$ and $\tilde{\mathsf{Z}}' \vdash \varphi_t[x] \to \varphi_t[x'] \to x \equiv x'$.*

Proof. Both claims are by induction on t, the latter with x and x' generalised. The former is immediate for variables and $\emptyset$, we discuss the case of $\mathcal{P}(t)$. By induction we know $\tilde{\mathsf{Z}}' \vdash \exists\, \varphi_t$ yielding a set x simulating t and need to show $\tilde{\mathsf{Z}}' \vdash \exists \exists\, \varphi_t[\mathsf{x}_0; \uparrow^2] \wedge \forall \mathsf{x}_0 \in \mathsf{x}_2 \leftrightarrow \mathsf{x}_0 \subseteq \mathsf{x}_1$. After instantiating the first quantifier with the set u guaranteed by the existential power set axiom for the set x and the second quantifier with x itself, it remains to show $\varphi_t[x]$ and $\forall \mathsf{x}_0 \in u \leftrightarrow \mathsf{x}_0 \subseteq x$ which are both straightforward by the choice of x and u.

The second claim follows from extensionality given that the characterisation φ_t specifies its satisfying sets exactly by their elements. So in fact the axioms concerning the set operations are not even used in the proof of uniqueness. ◀

During translation, substitution of terms can be simulated by substitution of variables:

Lemma 42. *Forall $\varphi : \mathbb{F}$ and $t : \mathbb{T}$ we have $\tilde{\mathsf{Z}}' \vdash \varphi_t[x] \to (\tilde{\varphi}[x] \leftrightarrow \widetilde{\varphi[t]})$.*

Proof. By induction on φ, all cases but the atoms are straightforward, relying on the fact that the syntax translation interacts well with variable renamings in the quantifier cases. The proof for atoms relies on a similar lemma for terms stating that $\varphi_s[y; x]$ and $\varphi_{s[t]}[y]$ are interchangeable whenever $\varphi_t[x]$, then the rest is routine. ◀

The previous lemma is the main ingredient to verify the desired proof transformation:

Fact 43. $\mathsf{Z}' \vdash \varphi$ *implies* $\tilde{\mathsf{Z}}' \vdash \tilde{\varphi}$.

Proof. We prove the more general claim that $\Gamma +\!\!+ \mathsf{Z}' \vdash \varphi$ implies $\tilde{\Gamma} +\!\!+ \tilde{\mathsf{Z}}' \vdash \tilde{\varphi}$ by induction on the first derivation. All rules but the assumption rule (A), $\forall$-elimination (AE), and $\exists$-elimination (EE) are straightforward, we explain the former two.

- If $\varphi \in \Gamma +\!\!+ \mathsf{Z}'$, then either $\varphi \in \Gamma$ or $\varphi \in \mathsf{Z}'$. In the former case we have $\tilde{\varphi} \in \tilde{\Gamma}$, so $\tilde{\Gamma} +\!\!+ \tilde{\mathsf{Z}}' \vdash \tilde{\varphi}$ by (A). Regarding the latter case, we can verify $\tilde{\mathsf{Z}}' \vdash \tilde{\varphi}$ for all $\varphi \in \mathsf{Z}'$ by rather tedious derivations given the sheer size of some axiom translations.
- If $\Gamma +\!\!+ \mathsf{Z}' \vdash \varphi[t]$ was derived from $\Gamma +\!\!+ \mathsf{Z}' \vdash \forall\varphi$, then by the inductive hypothesis we know $\tilde{\Gamma} +\!\!+ \tilde{\mathsf{Z}}' \vdash \forall\tilde{\varphi}$. Given Lemma 41 we may assume $\varphi_t[x]$ for a fresh variable x. Then by instantiating the inductive hypothesis to x via (AE) we obtain $\tilde{\Gamma} +\!\!+ \tilde{\mathsf{Z}}' \vdash \tilde{\varphi}[x]$ and conclude the claim $\tilde{\Gamma} +\!\!+ \tilde{\mathsf{Z}}' \vdash \widetilde{\varphi[t]}$ with Lemma 42. ◀

Now the undecidability of the symbol-free axiomatisations can be established.

Theorem 44. CE *implies both* $\mathsf{PCP} \preceq \tilde{\mathsf{Z}}'$ *and* $\mathsf{PCP} \preceq \tilde{\mathsf{Z}}$, *and* $\mathsf{CE} \land \mathsf{TD}$ *implies* $\mathsf{PCP} \preceq \widetilde{\mathsf{ZF}}$.

Proof. Similar to Theorem 10 based on Facts 40 and 43 and the reduction from Section 5. ◀

We conclude this section with a brief observation concerning the further reduced signature $\check{\Sigma} := (\_ \in \_)$, full detail can be found in the Coq development. Since equality is expressible in terms of membership by $x \equiv y := \forall z.\, x \in z \leftrightarrow y \in z$, we can rephrase the above translation to yield formulas $\check{\varphi} : \mathbb{F}_{\check{\Sigma}}$ satisfying the same properties as stated in Facts 40 and 43 for a corresponding axiomatisation $\check{\mathsf{Z}}'$. Moreover, since $\check{\mathsf{Z}}'$ does not refer to primitive equality, we can freely interpret it with the fully constructive model given in Theorem 4.6 of [18] and therefore obtain $\mathsf{PCP} \preceq \check{\mathsf{Z}}'$ without assumptions. This allows us to deduce the undecidability of the Entscheidungsproblem in its sharpest possible form:

Theorem 45. *First-order logic with a single binary relation symbol is undecidable.*

Proof. By Fact 11 and the reduction $\mathsf{PCP} \preceq \check{\mathsf{Z}}'$. ◀

7 Discussion

7.1 General Remarks

In this paper, we have described a synthetic approach to the formalisation and mechanisation of undecidability and incompleteness results in first-order logic. The general approach was then instantiated in two case-studies, one concerned with arithmetic theories in the family of PA as the typical systems considered in the investigation of incompleteness, and another one regarding fragments of ZF set theory as one of the standard foundations of mathematics. The chosen strategy complements the considerably harder to mechanise proofs relying on Gödel sentences, and for ZF the choice of PCP as seed problem instead of H_{10} or PA itself is a slight simplification since only a single recursion needs to be simulated. We use this section for some additional remarks based on the helpful feedback by the anonymous reviewers.

As formally stated in Definition 8, we only consider incompleteness as a property of the *classical* deduction system. This is simply owing to the fact that much of the literature on incompleteness seems focused on classical logic, with a notable exception of the more agnostic treatment in [26]. Although likely weaker in general, incompleteness of the *intuitionistic* deduction system can also be considered a meaningful property and follows in an analogous way. Concretely, a corresponding version of Fact 9 holds for the intuitionistic notion, yielding variants of Theorems 26 and 36 provable without LEM.

In alignment with [11] but in contrast to [12], we define semantic entailment $\mathcal{T} \vDash \varphi$ without restricting to *classical models*, i.e. models that satisfy all first-order instances of LEM. In our constructive meta-theory this relaxation is necessary to be able to use the standard models of PA and ZF, which would only be classical in a classical meta-theory. Leaving $\mathcal{T} \vDash \varphi$ in this sense constructively underspecified seems like a reasonable trade for a more economical usage of LEM.

Similarly, we leave it underspecified whether PA and ZF are seen as classical theories or their intuitionistic counterparts, namely Heyting arithmetic and a variant of intuitionistic set theory, respectively. By the choice not to distinguish these explicitly by LEM as a first-order axiom scheme, we leave it to the deduction system to discriminate between both views while the Tarski-style semantics emphasises the classical interpretation (especially in the presence of LEM). For simplicity, we decided to only speak of PA and ZF in the main body of the text, especially since a discussion of intuitionistic set theories would involve choosing a particular system. While IZF is an extension of Z' close to ZF with collection instead of replacement, the more predicative CZF does not have power sets as included in Z'.

7.2 Coq Mechanisation

Our axiom-free mechanisation contributes 5300loc to the Coq Library of Undecidability Proofs [14], on top of about 1300loc that could be reused from previous developments [12, 18]. Remarkably, the reduction from H_{10} to PA consists of only 700loc while already the initial reduction from PCP to ZF in the skolemised signature is above 1600loc. The remaining 3000loc mostly concern the technically more challenging translations to the sparse signatures of $\tilde{\mathsf{Z}}'$ and $\check{\mathsf{Z}}'$ as well as the use of intensional setoid models for the elimination of CE. By the latter, the given reductions can be verified constructively up to Z while the local assumption of TD remains necessary for full ZF. The development is available on our project page (see link in header) and all statements and some highlighted notations in the PDF version of this paper are systematically hyperlinked with HTML documentation of the code.

Our mechanisation of first-order logic unifies ideas from previous versions [11, 12, 17] and is general enough to be reused in other use cases. Notably, we refrained from including equality as a syntactic primitive to treat both intensional and extensional interpretations without changing the underlying signature. On the other hand, with primitive equality, the extensionality of models would hold definitionally and the deduction system could be extended with the Leibniz rule, making the additional axiomatisation of equality obsolete.

Furthermore, manipulating deductive goals of the form $\Gamma \vdash \varphi$ benefitted a lot from custom tactics, mostly to handle substitution and the quantifier rules. The former tactics approximate the automation provided by the Autosubst 2 framework unfortunately relying on functional extensionality [37] and the latter are based on the named reformulations of (AI) and (EE) given in Section 2.3. We are currently working on a more scalable proof mode for deductive goals including a HOAS input language hiding de Bruijn encodings, implementing a two-level approach in comparison to the one-level compromise proposed by Laurent [21].

7.3 Related Work

We report on other mechanisations concerned with incompleteness and undecidability results in first-order logic. Regarding the former, a fully mechanised proof of Gödel's first incompleteness theorem was first given by Shankar [32] using the Nqthm prover. O'Connor [24] implements the same result fully constructively in Coq, and Paulson [25] provides an Isabelle/HOL mechanisation of both incompleteness theorems using the theory of hereditarily finite sets instead of a fragment of PA. Moreover, there are several partial mechanisations [29, 5, 33], and Popescu and Traytel [26] investigate the abstract preconditions of the incompleteness theorems using Isabelle/HOL. With the independence of the continuum hypothesis, Han and van Doorn [15] mechanise a specific instance of incompleteness for ZF in Lean. None of these mechanisations approach incompleteness via undecidability.

Turning to undecidability results, Forster, Kirst, and Smolka [11] mechanise the undecidability of the Entscheidungsproblem in Coq, using a convenient signature to encode PCP, and Kirst and Larchey-Wendling [17] give a Coq mechanisation of Trakhtenbrot's theorem [40], stating the undecidability of finite satisfiability. They also begin with a custom signature for the encoding of PCP but provide the transformations necessary to obtain the undecidability result for the minimal signature containing a single binary relation symbol. We are not aware of any previous mechanisations of the undecidability of PA or ZF.

7.4 Future Work

There are two ways how our incompleteness results (Theorems 26 and 36) could be strengthened. First, the assumption of LEM is only due to the fact that we need soundness, for instance to deduce $\mathsf{Q}' \vDash \varphi_{p,q}$ from $\mathsf{Q}' \vdash_c \varphi_{p,q}$. As done previously [11], it should be possible to employ a Friedman translation to extract $\mathsf{Q}' \vdash_i \varphi_{p,q}$ from $\mathsf{Q}' \vdash_c \varphi_{p,q}$ and hence to obtain $\mathsf{Q}' \vDash \varphi_{p,q}$ constructively. Secondly, that supposed negation-completeness only implies synthetic decidability of a halting problem instead of a provable contradiction could be sharpened by extracting all reduction functions to a concrete model of computation like the weak call-by-value λ-calculus L [13]. Then the actual contradiction of an L-decider for L-halting could be derived.

We plan to continue the work on PA with a constructive analysis of Tennenbaum's theorem [39], stating that no computable non-standard model of PA exists. Translated to the synthetic setting where all functions are computable by construction, this would mean that no non-standard model of PA can be defined in Coq's type theory as long as function symbols are interpreted with type-theoretic functions. It will be interesting to investigate which assumptions are necessary to derive this as a theorem in Coq.

Regarding the reductions to ZF, it should be possible to eliminate the infinite set ω used to simplify the accumulation of partial solutions. Then the fully constructive and extensional standard model of hereditarily finite sets [34] would be available. Further eliminating the power set axiom, segments of this model could be used to obtain a more direct mechanisation of Trakhtenbrot's theorem than the previous one using signature transformations [17].

In general, it would be interesting to find a more elementary characterisation of an undecidable binary relation usable for the sharp formulations of the Entscheidungsproblem and Trakhtenbrot's theorem. This might well work without an intermediate axiomatisation of set theory and express an undecidable decision problem more primitively.

Moreover, by a straightforward extension of the translation in Section 6, one could deduce the conservativity of ZF over $\widetilde{\mathsf{ZF}}$, i.e. that if $\mathsf{ZF} \vdash \varphi$ for φ free of function symbols, then already $\widetilde{\mathsf{ZF}} \vdash \varphi$. This is an instance of the more general fact that first-order logic with definable symbols is conservative, which would be a worthwhile addition to our development.

Finally, we plan to mechanise similar undecidability and incompleteness results for second-order logic. Since second-order PA is categorical, in particular the incompleteness of any sound and enumerable deduction system for second-order logic would then follow easily.

References

1 Peter Aczel. The type theoretic interpretation of constructive set theory. In *Studies in Logic and the Foundations of Mathematics*, volume 96, pages 55–66. Elsevier, 1978.

2 Bruno Barras. Sets in Coq, Coq in sets. *Journal of Formalized Reasoning*, 3(1):29–48, 2010.

3 Andrej Bauer. First steps in synthetic computability theory. *Electronic Notes in Theoretical Computer Science*, 155:5–31, 2006.

4 Thomas Braibant and Damien Pous. An efficient Coq tactic for deciding Kleene algebras. In *International Conference on Interactive Theorem Proving*, pages 163–178. Springer, 2010.

5 Alan Bundy, Fausto Giunchiglia, Adolfo Villafiorita, and Toby Walsh. An incompleteness theorem via abstraction, 1996. Technical report.

6 Alonzo Church et al. A note on the Entscheidungsproblem. *J. Symb. Log.*, 1(1):40–41, 1936.

7 Nicolaas G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972.

8 John Doner and Wilfrid Hodges. Alfred Tarski and decidable theories. *The Journal of symbolic logic*, 53(1):20–35, 1988.

9 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 *LIPIcs*, pages 21:1–21:19, Dagstuhl, Germany, 2021.

10 Yannick Forster, Edith Heiter, and Gert Smolka. Verification of PCP-related computational reductions in Coq. In *International Conference on Interactive Theorem Proving*, pages 253–269. Springer, 2018.

11 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.

12 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, 2021.

13 Yannick Forster and Fabian Kunze. A certifying extraction with time bounds from Coq to call-by-value lambda calculus. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *LIPIcs*, pages 17:1–17:19, Dagstuhl, Germany, 2019.

14 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*, New Orleans, LA, United States, 2020. URL: `https://github.com/uds-psl/coq-library-undecidability`.

15 Jesse Han and Floris van Doorn. A formal proof of the independence of the continuum hypothesis. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 353–366, 2020.

16 David Hilbert and Wilhelm Ackermann. *Grundzüge der theoretischen Logik*. Springer, 1928.

17 Dominik Kirst and Dominique Larchey-Wendling. Trakhtenbrot's theorem in Coq: a constructive approach to finite model theory. In *International Joint Conference on Automated Reasoning (IJCAR 2020), Paris, France*, Paris, France, 2020. Springer.

18 Dominik Kirst and Gert Smolka. Large model constructions for second-order ZF in dependent type theory. *Certified Programs and Proofs - 7th International Conference, CPP 2018, Los Angeles, USA, 2018*, January 2018.

19 Georg Kreisel. Church's thesis: a kind of reducibility axiom for constructive mathematics. In *Studies in Logic and the Foundations of Mathematics*, volume 60, pages 121–150. Elsevier, 1970.

20 Dominique Larchey-Wendling and Yannick Forster. Hilbert's tenth problem in Coq. In *4th International Conference on Formal Structures for Computation and Deduction*, volume 131 of *LIPIcs*, pages 27:1–27:20, February 2019.

21 Olivier Laurent. An anti-locally-nameless approach to formalizing quantifiers. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 300–312, 2021.

22 Petar Maksimović and Alan Schmitt. HOCore in Coq. In *International Conference on Interactive Theorem Proving*, pages 278–293. Springer, 2015.

23 John Myhill. Some properties of intuitionistic Zermelo-Frankel set theory. In *Cambridge Summer School in Mathematical Logic*, pages 206–231. Springer, 1973.

24 Russell O'Connor. Essential incompleteness of arithmetic verified by Coq. In Joe Hurd and Tom Melham, editors, *Theorem Proving in Higher Order Logics*, pages 245–260, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

25 Lawrence C. Paulson. A mechanised proof of Gödel's incompleteness theorems using Nominal Isabelle. *Journal of Automated Reasoning*, 55(1):1–37, 2015.

26 Andrei Popescu and Dmitriy Traytel. A formally verified abstract account of Gödel's incompleteness theorems. In *International Conference on Automated Deduction*, pages 442–461. Springer, 2019.

27 Emil L. Post. Recursively enumerable sets of positive integers and their decision problems. *bulletin of the American Mathematical Society*, 50(5):284–316, 1944.

28 Mojżesz Presburger and Dale Jabcquette. On the completeness of a certain system of arithmetic of whole numbers in which addition occurs as the only operation. *History and Philosophy of Logic*, 12(2):225–233, 1991.

29 Art Quaife. Automated proofs of Löb's theorem and Gödel's two incompleteness theorems. *Journal of Automated Reasoning*, 4(2):219–231, 1988.

30 Fred Richman. Church's thesis without tears. *The Journal of symbolic logic*, 48(3):797–803, 1983.

31 Steven Schäfer, Gert Smolka, and Tobias Tebbi. Completeness and decidability of de Bruijn substitution algebra in Coq. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, pages 67–73. ACM, 2015.

32 Natarajan Shankar. *Proof-checking metamathematics.* The University of Texas at Austin, 1986. PhD Thesis.

33 Wilfried Sieg and Clinton Field. Automated search for Gödel's proofs. In *Deduction, Computation, Experiment*, pages 117–140. Springer, 2008.

34 Gert Smolka and Kathrin Stark. Hereditarily finite sets in constructive type theory. In *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-27, 2016*, volume 9807 of *LNCS*, pages 374–390. Springer, 2016.

35 Raymond M. Smullyan and Melvin Fitting. *Set theory and the continuum problem.* Dover Publications, 2010.

36 Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, 2020.

37 Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions. In *International Conference on Certified Programs and Proofs*, pages 166–180. ACM, 2019.

38 The Coq Development Team. The Coq Proof Assistant, version 8.12.0, 2020. `doi:10.5281/zenodo.4021912`.

39 Stanley Tennenbaum. Non-Archimedean models for arithmetic. *Notices of the American Mathematical Society*, 6(270):44, 1959.

40 Boris A. Trakhtenbrot. The impossibility of an algorithm for the decidability problem on finite classes. *Dokl. Akad. Nok. SSSR*, 70(4):569–572, 1950.

41 Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

42 Benjamin Werner. Sets in types, types in sets. In *Theoretical Aspects of Computer Software*, pages 530–546. Springer, Berlin, Heidelberg, 1997.

A Deduction Systems

Intuitionistic natural deduction $\Gamma \vdash_i \varphi$ is defined inductively by the following rules:

$$\frac{\varphi \in \Gamma}{\Gamma \vdash \varphi}\ \mathrm{C} \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi}\ \mathrm{E} \qquad \frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\ \mathrm{II} \qquad \frac{\Gamma \vdash \varphi \to \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \varphi}\ \mathrm{IE}$$

$$\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \land \psi}\ \mathrm{CI} \qquad \frac{\Gamma \vdash \varphi \land \psi}{\Gamma \vdash \varphi}\ \mathrm{CE}_1 \qquad \frac{\Gamma \vdash \varphi \land \psi}{\Gamma \vdash \psi}\ \mathrm{CE}_2$$

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \lor \psi}\ \mathrm{DI}_1 \qquad \frac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \lor \psi}\ \mathrm{DI}_2 \qquad \frac{\Gamma \vdash \varphi \lor \psi \quad \Gamma, \varphi \vdash \theta \quad \Gamma, \psi \vdash \theta}{\Gamma \vdash \theta}\ \mathrm{DE}$$

$$\frac{\Gamma[\uparrow] \vdash \varphi}{\Gamma \vdash \forall \varphi}\ \mathrm{AI} \qquad \frac{\Gamma \vdash \forall \varphi}{\Gamma \vdash \varphi[t]}\ \mathrm{AE} \qquad \frac{\Gamma \vdash \varphi[t]}{\Gamma \vdash \exists \varphi}\ \mathrm{EI} \qquad \frac{\Gamma \vdash \exists \varphi \quad \Gamma[\uparrow], \varphi \vdash \psi[\uparrow]}{\Gamma \vdash \psi}\ \mathrm{EE}$$

The classical variant $\Gamma \vdash_c \varphi$ adds all instances of the Peirce rule $((\varphi \to \psi) \to \varphi) \to \varphi$.

B Axioms of Set Theory

We list the ZF axioms over the signature $\Sigma := (\emptyset,\ \{\_,\_\},\ \bigcup\_,\ \mathcal{P}(\_),\ \omega\ ;\ \_ \equiv \_,\ \_ \in \_)$:

Structural axioms

Extensionality:	$\forall xy.\, x \subseteq y \to y \subseteq x \to x \equiv y$

Set operations

Empty set:	$\forall x.\, x \notin \emptyset$
Unordered pair:	$\forall xyz.\, z \in \{x, y\} \leftrightarrow x \equiv y \lor x \equiv z$
Union:	$\forall xy.\, y \in \bigcup x \leftrightarrow \exists z \in x.\, y \in z$
Power set:	$\forall xy.\, y \in \mathcal{P}(x) \leftrightarrow y \subseteq x$
Infinity:	$(\emptyset \in \omega \land \forall x.\, x \in \omega \to x \cup \{x\} \in \omega)$
	$\land\ (\forall y.\, (\emptyset \in y \land \forall x.\, x \in y \to x \cup \{x\} \in y) \to \omega \subseteq y)$

Axiom schemes

Separation:	$\lambda\varphi.\, \forall x.\, \exists y.\, \forall z.\, z \in y \leftrightarrow z \in x\ \land \varphi[x]$
Replacement	$\lambda\varphi.\, (\forall xyy'.\, \varphi[x, y] \to \varphi[x, y'] \to y \equiv y')$
	$\to \forall x.\, \exists y.\, \forall z.\, z \in y \leftrightarrow \exists u \in x.\, \varphi[u, z]$

Equality axioms

Reflexivity:	$\forall x.\, x \equiv x$
Symmetry:	$\forall xy.\, x \equiv y \to y \equiv x$
Transitivity:	$\forall xyz.\, x \equiv y \to y \equiv z \to x \equiv z$
Congruence:	$\forall xx'yy'.\, x \equiv x' \to y \equiv y' \to x \in y \to x' \in y'$

The core axiomatisation Z' contains extensionality and the set operation axioms, Z adds the separation scheme, and ZF also adds the replacement scheme. The equality axioms are added when working with the deduction system or in an intensional model.

Complete Bidirectional Typing for the Calculus of Inductive Constructions

Meven Lennon-Bertrand
LS2N, Université de Nantes – Gallinette Project Team, Inria, France

Abstract

This article presents a bidirectional type system for the Calculus of Inductive Constructions (CIC). The key property of the system is its completeness with respect to the usual undirected one, which has been formally proven in Coq as a part of the MetaCoq project. Although it plays an important role in an ongoing completeness proof for a realistic typing algorithm, the interest of bidirectionality is wider, as it gives insights and structure when trying to prove properties on CIC or design variations and extensions. In particular, we put forward constrained inference, an intermediate between the usual inference and checking judgements, to handle the presence of computation in types.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Bidirectional Typing, Calculus of Inductive Constructions, Coq, Proof Assistants

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.24

Supplementary Material *Software (Formalization)*: `https://github.com/MevenBertrand/metacoq/tree/itp-artefact`; archived at `swh:1:dir:d5e5b711b7c45fdfb18f02bf91684f771e0c8b65`

Acknowledgements Many thanks to Matthieu Sozeau for his help with MetaCoq and its nasty inductives, and to Chantal Keller, Nicolas Tabareau and the anonymous reviewers for their helpful comments on earlier versions of this article.

1 Introduction

In logical programming, an important characteristic of judgements is the *mode* of the objects involved, i.e., which ones are considered inputs or outputs. When examining this distinction for a typing judgement $\Gamma \vdash t : T$, both the term t under inspection and the context Γ of this inspection are usually known, they are thus inputs (although some depart from this, see [12]). The mode of the type T, however, may vary: should it be inferred based upon Γ and t, or do we merely want to check whether t conforms to a given T? Both are sensible approaches, and in fact typing algorithms for complex type systems usually alternate between them during the inspection of a single term/program. The bidirectional approach makes this difference between modes explicit, by decomposing undirected[1] typing $\Gamma \vdash t : T$ into two separate judgments $\Gamma \vdash t \triangleright T$ (inference) and $\Gamma \vdash t \triangleleft T$ (checking)[2], that differ only by modes. This decomposition allows theoretical work on practical typing algorithms, but also gives a finer grained structure to typing derivations, which can be of purely theoretical interest even without any algorithm in sight.

[1] We call anything related to the $\Gamma \vdash t : T$ judgement undirected by contrast with the bidirectional typing.
[2] We chose $\triangleright$ and $\triangleleft$ rather than the more usual $\Rightarrow$ and $\Leftarrow$ to avoid confusion with implication on paper, and with the Coq notation for functions in the development.

Although this seems appealing, and despite advocacy by McBride [15, 16] to adopt this approach when designing type systems, most of the theoretical work on dependent typing to this day remains undirected. Others have described on paper bidirectional algorithms for dependent types, starting with Coquand [11] and continuing with Norell [17] or Abel [5]. However, all of these consider unannotated λ-abstractions, and use bidirectional typing as a way to remedy this lack of annotations. This is sensible when looking for lightness of the input syntax, but poses an inherent completeness problem, as a term like $(\lambda\, x.x)\ 0$ does not type-check against type $\mathbb{N}$ in those systems. Very few have considered the case of annotated abstractions, apart from Asperti and the Matita team [7], who however concentrate on specific problems pertaining to unification and implementation of the Matita elaborator, without giving a general bidirectional framework. They also do not consider the problem of completeness with respect to a given undirected system, as it would fail in their setting due to the undecidability of higher order unification.

Thus, we wish to fill a gap in the literature, by describing a bidirectional type system that is complete with respect to the (undirected) Calculus of Inductive Constructions (CIC). By completeness, we mean that any term that is typable in the undirected system should also infer a type in the bidirectional one. This feature is very desirable when implementing kernels for proof assistants, whose algorithms should correspond to their undirected specification, never missing any typable term. The bidirectional systems we describe thus form intermediate steps between actual algorithms and undirected type systems. This step has proven useful in an ongoing completeness proof of MetaCoq's [23] type-checking algorithm[3]: rather than proving the algorithm complete directly, the idea is to prove it equivalent to the bidirectional type system, separating the implementation problems from the ones regarding the bidirectional structure.

But the interest of having a bidirectional type system equivalent to the undirected one is not limited to the link with algorithms, it is also purely theoretical. First, the structure of a bidirectional derivation is more constrained than that of an undirected one, especially regarding the uses of computation. This finer structure can make proofs easier, while the equivalence ensures that they can be transported to the undirected world. For instance, in a setting with cumulativity/subtyping, the inferred type for a term t should by construction be smaller than any other types against which t checks. This provides an easy proof of the existence of principal types in the undirected system. The bidirectional structure also provides a better base for new type systems. This was actually the starting point for this investigation: in [13], we quickly describe a bidirectional variant of CIC, as the usual undirected CIC is unfit for the gradual extension we envision due to the too high flexibility of a free-standing conversion rule. This is the system we wish to thoroughly describe and investigate here.

Outline. We start by giving in Section 2 a general roadmap in the simple setting of pure type systems, including the introduction of a constrained inference judgement that had not been clearly singled out in previous works. With the ideas set clear, we go on to the real thing: a bidirectional type system proven equivalent to the Predicative Calculus of Cumulative Inductive Constructions – PCUIC, the extension of CIC nowadays at the heart of Coq. This equivalence has been formalised on top of MetaCoq [24][4] We next turn back to less technical

[3] A completeness bug in that algorithm – also present in the Coq kernel – has already been found, see Section 3 for details.

[4] A version frozen as described in this article is available in the following git branch: `https://github.com/MevenBertrand/metacoq/tree/itp-artefact`.

considerations, as we believe that the bidirectional structure is of general theoretical interest. Section 4 thus describes the value of basing type systems on the bidirectional system directly rather than on the equivalent undirected one. Finally Section 5 investigates related work, and in particular tries and identify the implicit presence of constrained inference in various earlier articles, showing how making it explicit clarifies those.

2 Warming up with CCω

2.1 Undirected CCω

As a starting point, let us consider the system CCω. It is the backbone of CIC, and we can already illustrate most of our methodology on it. CCω belongs to the wider class of pure type systems (PTS), that has been thoroughly studied and described, see for instance [8]. Since there are many presentational variations, let us first give a precise account of our conventions. *Terms* in CCω are given by the grammar

$$t ::= x \mid \square_i \mid \Pi\, x : t.t \mid \lambda\, x : t.t \mid t\ t$$

where the letter x denotes a variable (so will letters y and z), and the letter i is an integer (we will also use letters j, k and l for those). All other Latin letters will be used for terms, with the upper-case ones used to suggest the corresponding terms should be though of as types – although this is not a syntactical separation. We abbreviate $\Pi\, x : A.B$ by $A \to B$ when B does not depend on x, as is customary. On those terms, *reduction* $\rightsquigarrow$ is defined as the least congruence such that $(\lambda\, x : T.t)\ u \rightsquigarrow t[x := u]$, where $t[x := u]$ denotes *substitution*. *Conversion* $\equiv$ is the symmetric, reflexive, transitive closure of reduction. Finally, *contexts* are lists of variable declarations $x : t$ and are denoted using capital Greek letters. We write $\cdot$ for the empty list, $\Gamma, x : T$ for concatenation, and $(x : T) \in \Gamma$ if $(x : T)$ appears in Γ. Combining those, we can define *typing* $\Gamma \vdash t : T$ as in Figure 1, where $i \vee j$ denotes the maximum of i and j. We say a context Γ is *well-formed* if $\vdash \Gamma$, a type T is *well-formed* in a context Γ if there exists i such that $\Gamma \vdash T : \square_i$, and a term is *well-formed* in a context Γ if there exists T such that $\Gamma \vdash t : T$. We also use *well-typed* for the latter, and leave the context implicit for the last two when it is clear from context. These rules differ from more usual PTS presentations such as [8] on the Var and Sort rules so as to avoid general weakening (which is however admissible) and single out the context well-formedness judgment. Premises are not minimal in order to provide more generous inductive hypotheses when doing proofs by induction on derivations. However, this presentation can easily be seen to be equivalent to that of [8].

As any PTS, CCω has many desirable properties. We summarize the ones we rely on here. Detailed proofs in the context of PTS can be found in [8], and formalisation of the corresponding properties for PCUIC are an important part of MetaCoq.

▶ **Proposition 1** (Properties of CCω). *The type system CCω as just described enjoys the following properties:*

Confluence *Reduction $\rightsquigarrow$ is confluent. As a direct consequence, two terms are convertible just when they have a common reduct: $t \equiv u$ if and only if there exists t' such that $t \rightsquigarrow^* t'$ and $u \rightsquigarrow^* t'$.*

Transitivity *Conversion is transitive.*

Subject reduction *If $\Gamma \vdash t : T$ and $t \rightsquigarrow t'$ then $\Gamma \vdash t' : T$.*

Validity *If $\Gamma \vdash t : T$ then T is well-formed, e.g. there exists some i such that $\Gamma \vdash T : \square_i$.*

$\boxed{\vdash \Gamma}$

$$\frac{}{\vdash \cdot}\ \textsc{Empty} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash A : \square_i}{\vdash \Gamma, x : A}\ \textsc{Ext}$$

$\boxed{\Gamma \vdash t : T}$

$$\frac{\vdash \Gamma}{\Gamma \vdash \square_i : \square_{i+1}}\ \textsc{Sort} \qquad \frac{\vdash \Gamma \qquad (x : A \in \Gamma)}{\Gamma \vdash x : A}\ \textsc{Var} \qquad \frac{\Gamma \vdash A : \square_i \qquad \Gamma, x : A \vdash B : \square_j}{\Gamma \vdash \Pi x : A.B : \square_{i \vee j}}\ \textsc{Prod}$$

$$\frac{\Gamma \vdash \Pi x : A.B : \square_i \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : \Pi x : A.B}\ \textsc{Abs} \qquad \frac{\Gamma \vdash t : \Pi x : A.B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B[x := u]}\ \textsc{App}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : \square_i \qquad A \equiv B}{\Gamma \vdash t : B}\ \textsc{Conv}$$

Figure 1 Undirected typing for CCω.

2.2 Turning CCω Bidirectional

McBride's discipline. To design our bidirectional type system, we follow a discipline exposed by McBride [15, 16]. The central point is to distinguish in a judgment between the subject, whose well-formedness is under scrutiny, from inputs, whose well-formedness is a condition for the judgment to behave well, and outputs, whose well-formedness is a consequence of the judgment. For instance, in inference $\Gamma \vdash t \triangleright T$, the subject is t, Γ is an input and T is an output. This means that one should consider whether $\Gamma \vdash t \triangleright T$ only in cases where $\vdash \Gamma$ is already known, and if the judgment is derivable it should be possible to conclude that both t and T are well-formed. All inference rules are to preserve this invariant. This means that inputs to a premise should be well-formed whenever the inputs to the conclusion and outputs and subjects of previous premises are. Similarly the outputs of the conclusion should be well-formed if the inputs of the conclusion and the subjects and outputs of the premises are assumed to be so.

This distinction also applies to the computation-related judgments, although those have no subject. For conversion $T \equiv T'$ both T and T' are inputs, and thus should be known to be well-formed beforehand. For reduction $T \rightsquigarrow^* T'$, T is an input and T' is an output, so only T needs to be well-formed, with the subject reduction property of the system ensuring that the output T' is also well-formed.

Constrained inference. Beyond the already described inference and checking judgements another one appears in the bidirectional typing rules of Figure 2: *constrained inference*, written $\Gamma \vdash t \triangleright_h T$, where h is either Π or $\square$ – and will be extended once we introduce more type formers. Constrained inference is a judgement – or, rather, a family of judgements indexed by h – with the exact same modes as inference, but where the type output is not completely free. Rather, as the name suggests, a constraint is imposed on it, namely that its head constructor can only be the corresponding element of h. This is needed to handle the behaviour absent in simple types that some terms might not have a desired type "on the nose", as exemplified by the first premise $\Gamma \vdash t \triangleright_\Pi \Pi x : A.B$ of the App rule for $t\ u$. Indeed,

$\boxed{\text{Inference: } \Gamma \vdash t \rhd T}$

$$\frac{}{\Gamma \vdash \square_i \rhd \square_{i+1}}\ \textsc{Sort} \qquad \frac{(x : T) \in \Gamma}{\Gamma \vdash x \rhd T}\ \textsc{Var} \qquad \frac{\Gamma \vdash A \rhd_{\square} \square_i \qquad \Gamma, x : A \vdash B \rhd_{\square} \square_j}{\Gamma \vdash \Pi\, x : A.B \rhd \square_{i \vee j}}\ \textsc{Prod}$$

$$\frac{\Gamma \vdash A \rhd_{\square} \square_i \qquad \Gamma, x : A \vdash t \rhd B}{\Gamma \vdash \lambda\, x : A.t \rhd \Pi\, x : A.B}\ \textsc{Abs} \qquad \frac{\Gamma \vdash t \rhd_{\Pi} \Pi\, x : A.B \qquad \Gamma \vdash u \lhd A}{\Gamma \vdash t\ u \rhd B[x := u]}\ \textsc{App}$$

$\boxed{\text{Checking: } \Gamma \vdash t \lhd T}$

$$\frac{\Gamma \vdash t \rhd T' \qquad T' \equiv T}{\Gamma \vdash t \lhd T}\ \textsc{Check}$$

$\boxed{\text{Constrained inference: } \Gamma \vdash t \rhd_h T}$

$$\frac{\Gamma \vdash t \rhd T \qquad T \rightsquigarrow^* \square_i}{\Gamma \vdash t \rhd_{\square} \square_i}\ \textsc{Sort-Inf} \qquad \frac{\Gamma \vdash t \rhd T \qquad T \rightsquigarrow^* \Pi\, x : A.B}{\Gamma \vdash t \rhd_{\Pi} \Pi\, x : A.B}\ \textsc{Prod-Inf}$$

Figure 2 Bidirectional typing for CCω.

it would be too much to ask t to directly infer a Π-type, as some reduction might be needed on T to uncover this Π. Checking also cannot be used, because the domain and codomain of the tentative Π-type are not known at that point: they are to be inferred from t.

Structural rules. To transform the rules of Figure 1 to those of Figure 2, we start by recalling that we wish to obtain a complete bidirectional type system. Therefore any term should infer a type, and thus all structural rules (i.e. all rules where the subject of the conclusion starts with a term constructor) should give rise to an inference rule. It thus remains to choose the judgements for the premises, which amounts to determining their modes. If a term in a premise appears as input in the conclusion or output of a previous premise, then it can be considered an input, otherwise it must be an output. Moreover, if a type output is unconstrained, then inference can be used, otherwise we must resort to constrained inference.

This applies straightforwardly to most rules but the PTS-style Abs. Indeed, neither $\Gamma \vdash \Pi\, x : A.B : \square_i$ nor $\Gamma, x : A \vdash t : B$ can be taken as the first bidirectional premise: the first one because B is not known from inputs to the conclusion, and the second because context $\Gamma, x : A$ is not known to be well-formed from the conclusion. For general PTS, this is quite problematic, as demonstrated by Pollack [19]. For CCω, however, the solution is simple. Replacing $\Gamma \vdash \Pi\, x : A.B : \square_i$ by the equivalent $\Gamma \vdash A : \square_j$ and $\Gamma, x : A \vdash B : \square_{j'}$, the former can become the first premise, ensuring that type inference for t is done in a well-formed context, and the latter can be dropped based upon the invariant that outputs – here the type B inferred for t – are well-formed.

Similarly, as the context is always supposed to be well-formed as an input to the conclusion, it is not useful to re-check it, and thus the premise of Sort and Var can be safely dropped. This is in line with implementations, where the context is not re-checked at leaves of a derivation tree, with performance issues in mind. The well-formedness invariants then ensure that any derivation starting with the (well-formed) empty context will only handle well-formed contexts.

Computation rules. We are now left with the non-structural conversion rule. As we observed, there are two possible modes for computation: if both sides are inputs, conversion can be used, but if only one is known one must resort to reduction, and the other side becomes an output instead. Rule CHECK corresponds to the first case, while rules PROD-INF and SORT-INF both are in the second case. This difference in turn introduces the need to separate between checking, that calls for the first rule, and constrained inference, that requires the others.

The need to split the conversion rule into a (weak-head) reduction and conversion subroutine depending on the mode is known to the implementors of proof assistants [2]. However, we wish to de-emphasize the role devoted specifically to reduction in the description of those algorithms, instead putting constrained inference forward. Indeed, reduction is only a means to determine whether a certain term fits into a certain category of types. In the setting of CCω, there is mainly one way to do so, which is to reduce its type until its head constructor is exposed. However, as soon as conversion is extended, for instance with unification [7], coercions [7, 22] or graduality [13], reduction is not enough any more. Singling out constrained inference then makes the required modification to the typing rules clearer. We come back to this more in length in Section 5.1.

2.3 Properties

Let us now state the two main properties relating the bidirectional system to the undirected one: it is both correct (terms typable in the bidirectional system are typable in the undirected system) and complete (all terms typable in the undirected system are also typable in the bidirectional system).

2.3.1 Correctness

A bidirectional derivation can be seen as a refinement of an undirected derivation. Indeed, the bidirectional structure can be erased – replacing each bidirectional rule with the corresponding undirected rule – to obtain an undirected derivation, but for missing sub-derivations, which can be retrieved using the invariants on well-formedness of inputs and outputs. Thus, we get the following correctness theorem – note how McBride's discipline manifests as well-formedness hypothesis on inputs.

▶ **Theorem 2** (Correctness of bidirectional typing for CCω). *If Γ is well-formed and $\Gamma \vdash t \triangleright T$ or $\Gamma \vdash t \triangleright_h T$ then $\Gamma \vdash t : T$. If Γ and T are well-formed and $\Gamma \vdash t \triangleleft T$ then $\Gamma \vdash t : T$.*

Proof. The proof is by mutual induction on the bidirectional typing derivation.

Each rule of the bidirectional system can be replaced by the corresponding rule of the undirected system, with all three CHECK, PROD-INF and SORT-INF replaced by CONV, ABS using an extra PROD rule. In all cases, the induction hypothesis can be used on sub-derivations of the bidirectional judgment because the context extensions and checking are done with types that are known to be well-formed by induction hypothesis on previous premises and validity.

Some sub-derivations of the undirected rules that have no counterpart in the bidirectional ones are however missing. In rules SORT and VAR the hypothesis that $\vdash \Gamma$ is enough to get the required premise. For rule CHECK, the well-formedness hypothesis on the type is needed to get the second premise of rule CONV. As for PROD-INF and SORT-INF, that second premise is obtained using the induction hypothesis, validity and subject reduction. Finally, the missing premise on the codomain of the product type in rule ABS is obtained by validity.

Uses of validity could alternatively be handled by strengthening the theorem to incorporate the well-formedness of types when they are outputs. ◀

2.3.2 Completeness

Let us now state the most important property of our bidirectional system: it does not miss any undirected derivation.

▶ **Theorem 3** (Completeness of bidirectional typing for CCω). *If $\Gamma \vdash t : T$ then there exists T' such that $\Gamma \vdash t \triangleright T'$ and $T' \equiv T$.*

Proof. The proof is by induction on the undirected typing derivation.

Rules Sort and Var are base cases, and can be replaced by the corresponding rules in the bidirectional world. Rule Conv is a direct consequence of the induction hypothesis on its first premise, together with transitivity of conversion.

For rule Prod, we need the intermediate lemma that if T is a term such that $T \equiv \square_i$, then also $T \rightsquigarrow^* \square_i$. This is a consequence of confluence of reduction. In turn, it implies that if $\Gamma \vdash t \triangleright T$ and $T \equiv \square_i$ then $\Gamma \vdash t \triangleright_{\square} \square_i$, and is enough to conclude for that rule.

In rule Abs, the induction hypothesis gives $\Gamma \vdash \Pi\, x : A.B \triangleright T$ for some T, and an inversion on this gives $\Gamma \vdash A \triangleright_{\square} \square_i$ for some i. Combined with the second induction hypothesis, we get $\Gamma \vdash \lambda\, x : A.t \triangleright \Pi\, x : A.B'$ for some B' such that $B \equiv B'$, and thus $\Pi\, x : A.B \equiv \Pi\, x : A.B'$ as desired.

We are finally left with the App rule. We know that $\Gamma \vdash t \triangleright T$ with $T \equiv \Pi\, x : A.B$. Confluence then implies that $T \rightsquigarrow^* \Pi\, x : A'.B'$ for some A' and B' such that $A \equiv A'$ and $B \equiv B'$. Thus $\Gamma \vdash t \triangleright_{\Pi} \Pi\, x : A'.B'$. But by induction hypothesis we also know that $\Gamma \vdash u \triangleright A''$ with $A'' \equiv A$ and so by transitivity of conversion $\Gamma \vdash u \triangleleft A'$. We can thus apply App to conclude. ◀

Contrarily to correctness, which kept a similar derivation structure, completeness is of a different nature. Because in bidirectional derivations the conversion rules are much less liberal than in undirected derivations, the crux of the proof is to ensure that conversions can be permuted with structural rules, in order to concentrate them in the places where they are authorized in the bidirectional derivation. In a way, composing completeness with conversion gives a kind of normalization procedure that produces a canonical undirected derivation by pushing all conversions down as much as possible.

2.3.3 Reduction strategies

The judgements of Figure 2 are syntax-directed, in the sense that there is always at most one rule that can be used to derive a certain typing judgements. But with the rules as given there is still some indeterminacy. Indeed when appealing to reduction no strategy is fixed, thus two different reducts give different uses of the rule, resulting in different inferred types – although those are still convertible. However, a reduction strategy can be imposed to completely eliminate indeterminacy in typing, leading to the following.

▶ **Proposition 4** (Reduction strategy). *If $\rightsquigarrow^*$ is replaced by weak-head reduction in rules Sort-Inf and Prod-Inf, then given a well-formed context Γ and a term t there is at most one derivation of $\Gamma \vdash t \triangleright T$ and $\Gamma \vdash t \triangleright_h T$, and so in particular such a T is unique. Similarly, given well-formed Γ and T and a term t there is at most one derivation of $\Gamma \vdash t \triangleleft T$. Moreover, the existence of those derivations is decidable.*

The algorithm for deciding the existence of the derivations is straightforward from the bidirectional rules, it amounts to structural recursion on the subject.

3 From CCω to PCUIC

CCω is already a powerful system, but today's proof assistants rely on much more complex features. There are two main areas of differences between CCω and the Predicative Calculus of Cumulative Inductive Constructions (PCUIC), the type theory nowadays behind the Coq proof assistant. Adapting to those is a good stress test for the bidirectional approach: seamlessly doing so is a good sign that the general methodology we presented is robust.

The first area of difference are the universes. While on paper those are simply integer, to handle typical ambiguity and polymorphic (co)-inductive types, PCUIC uses algebraic universes, containing level variables, algebraic $\vee$ and $+1$ operators, and a special level for the sort Prop. Moreover, those universes are cumulative, that is they behave as if smaller universes were included in larger ones. The precise handling of the algebraic universes is abstracted away in MetaCoq, and quite similar in the directed and undirected systems, so it did not prove too difficult to handle. Cumulativity, however, introduces some not-so-small differences with the previous presentation, so we spend some time on it in Section 3.1.

The second is the addition of new base type and term constructors. We describe the treatment of inductive types in Section 3.2. Co-inductive types and records behave very similarly to inductive types at the level of typing, so we do not dwell on them. The difference lies mainly at the level of reduction/conversion, but as our type system treats those as black boxes the differences have a negligible impact.

The interplay between those two areas is quite subtle, and we were able to uncover an incompleteness bug in the current kernel of Coq regarding pattern-matching of cumulative inductive types. This bug had gone unnoticed until our formalisation, but was causing subject reduction failures in some corner cases[5]. We try and give an intuition of the problem in Section 3.2.

3.1 Cumulativity

PCUIC incorporates a limited form of subtyping, corresponding to the intuition that smaller universes are included in larger ones. Conversion $\equiv$ is therefore replaced by *cumulativity* $\preceq$, the main difference being that the constraint on universes is relaxed. For conversions $\square_i \equiv \square_j$ only when $i = j$, but for cumulativity $\square_i \preceq \square_j$ whenever $i \leq j$ – and this extends by congruence through most constructors. The conversion rule is accordingly replaced by the following cumulativity rule

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : \square_i \qquad A \preceq B}{\Gamma \vdash t : B}\ \textsc{Cumul}$$

This reflects the view that universes $\square_i$ should be included one in the next when going up in the hierarchy. In CCω, all types for a given term t in a fixed context Γ are equally good, as they are all convertible. This is not the case any more in presence of cumulativity, as we can have $T \preceq T'$ but not $T \equiv T'$. Of particular interest are principal types, defined as follows.

▶ **Definition 5** (Principal type). *The term T is called a* principal type *for term t in context Γ if it is a least type for t in Γ, that is if $\Gamma \vdash t : T$ and for any T' such that $\Gamma \vdash t : T'$ we have $T \preceq T'$.*

[5] The precise issue in the kernel is described in this git issue: `https://github.com/coq/coq/issues/13495`.

The existence of such principal types is no so easy to prove directly but quite useful, as they are in a sense the best types for any terms. Indeed, if T is a principal type for t in Γ and T' is any other type for t, the CUMUL rule can be used to deduce $\Gamma \vdash t : T'$ from $\Gamma \vdash t : T$, which in general is not the case if T is not principal. Similarly, if T and T' are two types for a term t, then they are not directly related, but the existence of principal types ensures that there exists some T'' that is a type for t and such that $T \preceq T'$ and $T \preceq T''$, indirectly relating T' and T''.

Reflecting this modification in the bidirectional system of course calls for an update to the computation rules. The change to the CHECK rule is direct: simply replace conversion with cumulativity.

$$\frac{\Gamma \vdash t \triangleright A \qquad A \preceq B}{\Gamma \vdash t \triangleleft B}\ \textsc{Cumul}$$

As to the constrained inference rules, there is no need to modify them. Intuitively, this is because there is no reason to degrade a type to a larger one when it is not needed. We only resort to cumulativity when it is forced by a given input. In that setting, completeness becomes the following:

▶ **Theorem 6** (Completeness with cumulativity). *If $\Gamma \vdash t : T$ using rules of Figure 1 replacing* CONV *with* CUMUL*, then $\Gamma \vdash t \triangleright T'$ is derivable with rules of Figure 2 replacing* CHECK *with* CUMUL *for some T' such that $T' \preceq T$.*

In that setting, even without fixing a reduction strategy as in Proposition 4, there is a weaker uniqueness property for inferred types.

▶ **Proposition 7** (Uniqueness of inferred type). *If Γ is well-formed, $\Gamma \vdash t \triangleright T$ and $\Gamma \vdash t \triangleright T'$ then $T \equiv T'$. Similarly if Γ is well-formed, $\Gamma \vdash t \triangleright_h T$ and $\Gamma \vdash t \triangleright_h T'$ then $T \equiv T'$.*

Proof. Mutual induction on the first derivation. It is key that constrained inference rules only reduce a type, so that the type in the conclusion is convertible to the type in the premise, rather than merely related by cumulativity. ◀

In particular, combining those two properties with a correctness property akin to Theorem 2, we can prove that any inferred type is principal, and so that they both exist and are computable since the bidirectional judgement can still be turned into an algorithm in the spirit of Proposition 4.

▶ **Proposition 8** (Principal types). *If Γ is well-formed and $\Gamma \vdash t \triangleright T$ then T is a principal type for t in Γ.*

Proof. If $\Gamma \vdash t : T'$, then by completeness there exists some T'' such that $\Gamma \vdash t \triangleright T''$ and moreover $T'' \preceq T'$. But by uniqueness $T \equiv T'' \preceq T'$ and thus $T \preceq T'$, and T is indeed a principal type for t in Γ. ◀

Reasoning on the bidirectional derivation thus makes proofs easier, while the correctness and completeness properties ensure they can be carried to the undirected system. Another way to understand this is that seeing completeness followed by correction as a normalization procedure on derivations, the produced canonical derivation is more structured and thus more amenable to proofs. Here for instance the uniqueness of the inferred type translates to the existence of principal types via correctness, and the normalization of the derivations optimizes it to derive a principal type.

$$\frac{\Gamma \vdash A : \square_i \qquad \Gamma, x : A \vdash B : \square_j}{\Gamma \vdash \Sigma\, x : A.B : \square_{i \vee j}}\ \Sigma\text{-TYPE}$$

$$\frac{\Gamma \vdash A : \square_i \qquad \Gamma, x : A \vdash B : \square_j \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B[x := a]}{\Gamma \vdash (a, b)_{A,x.B} : \Sigma\, x : A.B}\ \Sigma\text{-CONS}$$

$$\frac{\Gamma, z : \Sigma\, x : A.B \vdash P : \square_i \qquad \Gamma, x : A, y : B \vdash b : P[z := (x, y)] \qquad \Gamma \vdash s : \Sigma\, x : A.B}{\Gamma \vdash \mathrm{rec}_\Sigma(z.P, x.y.p, s) : P[z := s]}\ \Sigma\text{-REC}$$

Figure 3 Undirected sum type.

$\boxed{\Gamma \vdash t \triangleright T}$

$$\frac{\Gamma \vdash A \triangleright_\square \square_i \qquad \Gamma, x : A \vdash B \triangleright_\square \square_j}{\Gamma \vdash \Sigma\, x : A.B \triangleright \square_{i \vee j}}\ \Sigma\text{-TYPE}$$

$$\frac{\Gamma \vdash A \triangleright_\square \square_i \qquad \Gamma, x : A \vdash B \triangleright_\square \square_j \qquad \Gamma \vdash a \triangleleft A \qquad \Gamma \vdash b \triangleleft B[x := a]}{\Gamma \vdash (a, b)_{A,x.B} \triangleright \Sigma\, x : A.B}\ \Sigma\text{-CONS}$$

$$\frac{\begin{array}{c}\Gamma \vdash s \triangleright_\Sigma \Sigma\, x : A.B \qquad \Gamma, z : \Sigma\, x : A.B \vdash P \triangleright_\square \square_i \\ \Gamma, x : A, y : B \vdash b \triangleleft P[z := (x, y)]\end{array}}{\Gamma \vdash \mathrm{rec}_\Sigma(z.P, x.y.b, s) \triangleright P[z := s]}\ \Sigma\text{-REC}$$

$\boxed{\Gamma \vdash t \triangleright_h T}$

$$\frac{\Gamma \vdash t \triangleright T \qquad T \rightsquigarrow^* \Sigma\, x : A.B}{\Gamma \vdash t \triangleright_\Sigma \Sigma\, x : A.B}\ \Sigma\text{-INF}$$

Figure 4 Bidirectional sum type.

3.2 Inductive Types

Sum type. Before we turn to the general case of inductive types of the formalisation, let us present a simple inductive type: dependent sums. The undirected rules are given in Figure 3, and are inspired from the theoretical presentation of such dependent sums, such at the one of the Homotopy Type Theory book [26]. In particular, we use the same convention to write $y.P$ when variable y is bound in P. Note however that contrarily to [26], some typing information is kept on the pair constructor. Exactly as for the abstraction, this is to be able to infer a unique, most general type in the bidirectional system. Indeed, without that information a pair (a, b) could inhabit multiple types $\Sigma\, x : A.B$ because there are potentially many incomparable types B such that $B[x := a]$ is a type for b, as even if $B[x := a]$ and $B'[x := a]$ are convertible B and B' may be quite different, depending of which instances of a in $B[x := a]$ are abstracted to x.

To obtain the bidirectional rules of Figure 4, first notice that all undirected rules are structural and must thus become inference rules if we want the resulting system to be complete, just as in Section 2. The question therefore is again to know which modes to choose for the premises. For Σ-TYPE and Σ-CONS this is straightforward: when the type appears in the conclusion, use checking, otherwise (constrained) inference. The case of the destructors is somewhat more complex. Handling the subterms of the destructor in the order in which they usually appear (predicate, branches and finally scrutinee) is not possible, as the parameters of the inductive type are needed to construct the context in which the predicate is typed. However those parameters can be inferred from the scrutinee. Thus, a type for the scrutinee is first obtained using a new constrained inference judgment, forcing the inferred type to be a Σ-type, but leaving its parameters free. Next, the obtained arguments can be used to construct the context to type the predicate. Finally, once the predicate is known to be well-formed, it can be used to type-check the branch.

$\boxed{\Gamma \vdash t \triangleright T}$

$$\frac{}{\Gamma \vdash \mathbb{N} \triangleright \square_0} \qquad \frac{}{\Gamma \vdash 0 \triangleright \mathbb{N}} \qquad \frac{\Gamma \vdash n \triangleleft \mathbb{N}}{\Gamma \vdash \mathrm{S}(n) \triangleright \mathbb{N}}$$

$$\frac{\begin{array}{c}\Gamma \vdash s \triangleright_{\mathbb{N}} \mathbb{N} \\ \Gamma, z : \mathbb{N} \vdash P \triangleright_{\square} \square_i \qquad \Gamma \vdash b_0 \triangleleft P[z := 0] \qquad \Gamma, x : \mathbb{N}, p : P[z := x] \vdash b_{\mathrm{S}} \triangleleft P[z := \mathrm{S}(x)]\end{array}}{\Gamma \vdash \mathrm{rec}_{\mathbb{N}}(z.P, b_0, x.p.b_{\mathrm{S}}, s) \triangleright P[z := s]}$$

$$\frac{\Gamma \vdash A \triangleright_{\square} \square_i \qquad \Gamma \vdash a \triangleleft A \qquad \Gamma \vdash a' \triangleleft A}{\Gamma \vdash \mathrm{Id}_A\ a\ a' \triangleright \square_i} \qquad \frac{\Gamma \vdash A \triangleright_{\square} \square_i \qquad \Gamma \vdash a \triangleleft A}{\Gamma \vdash \mathrm{refl}_A\ a \triangleright \mathrm{Id}_A\ a\ a}$$

$$\frac{\Gamma \vdash s \triangleright \mathrm{Id}_A\ a\ a' \qquad \Gamma, x : A, z : \mathrm{Id}_A\ a\ x \vdash P \triangleright_{\square} \square_i \qquad \Gamma \vdash b \triangleleft P[z := \mathrm{Id}_A\ a\ a][x := a]}{\Gamma \vdash \mathrm{rec}_{\mathrm{Id}}(x.z.P, b, s) \triangleright P[z := s][x := a']}$$

$\boxed{\Gamma \vdash t \triangleright_h T}$

$$\frac{\Gamma \vdash t \triangleright T \qquad T \rightsquigarrow^* \mathbb{N}}{\Gamma \vdash t \triangleright_{\mathbb{N}} \mathbb{N}} \qquad \frac{\Gamma \vdash t \triangleright T \qquad T \rightsquigarrow^* \mathrm{Id}_A\ a\ a'}{\Gamma \vdash t \triangleright_{\mathrm{Id}} \mathrm{Id}_A\ a\ a'}$$

Figure 5 Other bidirectional inductive types.

This same approach can be readily extended to other usual inductive types, with recursion or indices posing no specific problems, see Figure 5. The choice to use $\triangleright_{\square}$ rather than $\triangleleft$ for types is guided by the intuition that the universe level of e.g. A in $\mathrm{Id}_A\ a\ a'$ is free, similarly to what happens for sum types.

Polymorphic, Cumulative Inductive Types. The account of general inductive types in PCUIC is quite different from the one we just gave. The main addition is universe polymorphism [25], which means that inductive types and constructors come with explicit universe levels. The Σ-type of the previous paragraph, for instance, would contain an explicit universe level i, and both A and B would be checked against $\square_i$ rather than having their level inferred. This makes the treatment of complex inductive types possible by using checking uniformly – rather than relying on constrained inference to infer universe levels – at the cost of possibly needless annotations, as here with Σ-types. To make that polymorphism more seamless, those polymorphic inductive types are also cumulative [27]: in much the same way as $\square_i \preceq \square_j$ if $i \leq j$, also $\mathbb{N}^{@i} \preceq \mathbb{N}^{@j}$, where $\mathbb{N}^{@i}$ denotes the polymorphic inductive $\mathbb{N}$ at universe level i.

$\boxed{\Gamma \vdash t \triangleright T}$

$$\frac{}{\Gamma \vdash I^{@\mathbf{i}} \triangleright \Pi(\mathbf{x} : \mathbf{X}^{@\mathbf{i}})(\mathbf{y} : \mathbf{Y}^{@\mathbf{i}}), \square_{l^{@\mathbf{i}}}}\ \textsc{Ind} \qquad \frac{}{\Gamma \vdash c_k^{@\mathbf{i}} \triangleright \Pi(\mathbf{x} : \mathbf{X}^{@\mathbf{i}})(\mathbf{y} : \mathbf{Y_k}^{@\mathbf{i}}), I^{@\mathbf{i}}\ \mathbf{x}\ \mathbf{u}_\mathbf{k}^{@\mathbf{i}}}\ \textsc{Cons}$$

$$\frac{\begin{array}{c}\Gamma \vdash s \triangleright_I I^{@\mathbf{i}'}\ \mathbf{a}\ \mathbf{b} \qquad \Gamma \vdash \mathbf{p}_k \triangleleft \mathbf{X}_k^{@\mathbf{i}}[\mathbf{x} := \mathbf{p}] \\ \Gamma, \mathbf{y} : \mathbf{Y}^{@\mathbf{i}}[\mathbf{x} := \mathbf{p}], z : I^{@\mathbf{i}}\ \mathbf{p}\ \mathbf{y} \vdash P \triangleright_\square \square_j \\ I^{@\mathbf{i}'}\ \mathbf{a}\ \mathbf{b} \preceq I^{@\mathbf{i}}\ \mathbf{p}\ \mathbf{b} \qquad \Gamma, \mathbf{y} : \mathbf{Y_k}^{@\mathbf{i}}[\mathbf{x} := \mathbf{p}] \vdash \mathbf{t}_k \triangleleft P[\mathbf{y} := \mathbf{u_k}^{@\mathbf{i}}][z := c_k^{@\mathbf{i}}\ \mathbf{p}\ \mathbf{y}]\end{array}}{\Gamma \vdash \mathtt{match}\, s\, \mathtt{in}(I, \mathbf{i}, \mathbf{p})\, \mathtt{return}\, P\, \mathtt{with}[\mathbf{t}] \triangleright P[\mathbf{y} := \mathbf{b}][z := s]}\ \textsc{Match}$$

$$\frac{\Gamma \vdash T \triangleright_\square \square_i \qquad \Gamma, f : T \vdash t \triangleleft T \qquad \text{guard condition}}{\Gamma \vdash \mathtt{fix}\, f : T := t \triangleright T}\ \textsc{Fix}$$

$\boxed{\Gamma \vdash t \triangleright_I T}$

$$\frac{\Gamma \vdash t \triangleright T \qquad T \rightsquigarrow I\ \mathbf{a}\ \mathbf{b}}{\Gamma \vdash t \triangleright_I I\ \mathbf{a}\ \mathbf{b}}$$

Figure 6 Bidirectional inductive type – PCUIC style.

This enables lifting from a lower universe to a higher one, so that for instance $\vdash 0^{@i} : \mathbb{N}^{@j}$ if $i \leq j$. PCUIC as presented in MetaCoq also presents constructors and inductive types as functions, rather than requiring them to be fully applied, and it separates recursors into a pattern-matching and a fixpoint construct, the latter coming with a specific guard condition to keep the normalization property enjoyed by a system with recursors.

A sketch of the bidirectional rules is given in Figure 6, for a generic inductive I. We use bold characters to denote lists – for instance $\mathbf{a}$ is a list of terms – and indexes to denote a specific element – so that $\mathbf{a}_k$ is the k-th element of the previous. The considered inductive I has parameters of type $\mathbf{X}$, indices of type $\mathbf{Y}$ and inhabits some universe $\square_l$. Its constructors c_k are of types $\Pi(\mathbf{x} : \mathbf{X})(\mathbf{y} : \mathbf{Y_k}), I\ \mathbf{x}\ \mathbf{u}$, with $\mathbf{u_k}$ terms possibly depending on both $\mathbf{x}$ and $\mathbf{y}$. Since we are considering a polymorphic inductive type, all of those actually have to be instantiate with universe levels, an operation we denote with ${}^{@\mathbf{i}}$.

The two rules Ind, Cons are similar to those for variables, with the types pulled out of a global environment – not represented in our rules – rather than of the context. In particular, this presentation completely fixes the universe levels of the arguments. In rule Fix, the type of the fixpoint is checked to be well-formed, and then the body is checked against it. The guard condition, although complex, does not vary between the directed and undirected systems, and we thus do not dwell on it. The formalised version of this rule is complicated by the need to consider mutual (co-)fixpoints, but follows the same pattern.

Last but not least, Match follows the same structure as in Figures 4 and 5: first, the type of the scrutinee is inferred, then the predicate is verified to be a type and finally the branches are checked. An important point is how much information can be retrieved from the scrutinee s. Indeed, the universe levels $\mathbf{i}$ and the parameters $\mathbf{p}$ used to build the context in which the predicate P and branches $\mathbf{t}$ are typed are stored in the match constructor. For cumulative inductive types, this is crucial to retain equivalence between the undirected and bidirectional system, and wrongly building the context from the type inferred for the scrutinee led to the bug we discovered. The idea is that the match construction might need to be typed "higher" than the type of inferred for s to be able to type P and $\mathbf{t}$. Subsequently, a cumulativity check not appearing in the examples above is needed to ensure that the scrutinee checks against the type constructed using $\mathbf{i}$ and $\mathbf{p}$. In contrast with parameters, the inferred indices $\mathbf{b}$ can safely be used in the return type, but proving this is the most subtle part of the correctness proof.

3.3 The formalisation

Let us now go over the formalisation file by file.

BDEnvironmentTyping.v & BDTyping.v. The first file refines a few definitions on contexts from EnvironmentTyping.v in order to account for the difference between checking and constrained inference. We expect this to eventually replace the less precise definitions.

The second contains the definition of the bidirectional type system as a mutually defined inductive type whose main part is `infering`. The best way to understand this inductive is probably to compare it with `typing`, the inductive predicate for undirected typing.

We then go on to proving by hand a custom induction principle, by first introducing a notion of size of a derivation. This induction principle is not as strong as we might expect, as it does not provide the extra induction hypothesis on context and type that would go with McBride's discipline. We did not try to go and prove such a strong induction principle, as we did not need it. Instead, reflecting the discipline in the choice of the predicates proven by induction was enough. But the main reason was that proving such an induction principle effectively corresponds to an inline proof of validity, a property that required quite an important amount of work to get. We still conjecture that such a strong induction principle should be provable, by reproducing some of the lemmas on the undirected typing, with extra care taken to the size of the obtained typing derivations, so as to be able to use e.g. substitutivity of typing together with an induction hypothesis.

BDToPCUIC.v & BDFromPCUIC.v. The next two files prove the equivalence between both type system. Correctness (akin to Theorem 2) is `infering_typing` for inference and the following theorems for the other judgements. Completeness (akin to Theorem 6) is theorem `typing_infering`.

The bulk of both proofs is an induction on typing derivation whose most challenging part is the handling of the case constructor, especially the subtle issues around indices described in Section 3.2. Similarly to Section 2, correctness relies on the strong properties of validity and subject reduction to reconstruct missing premises, while completeness mostly requires transitivity of conversion and confluence of reduction.

BDUnique.v. This last file contains the proofs of Proposition 7 – `uniqueness_inferred` – and Proposition 8 – `principal_type`. Apart from some lemmas on conversion that were only proved for cumulativity in MetaCoq, the induction itself is quite straightforward thanks to the bidirectional structure.

4 Beyond PCUIC

The use of our bidirectional structure is not limited to CIC or PCUIC. On the contrary, we found it crucial to have such a bidirectional type system when designing a gradual extension to CIC [13], for multiple reasons we try and detail below.

But let us first give a bit of context about this extension. The aim was to adapt the ideas of gradual typing [21] to CIC. Gradual typing aims at incorporating some level of dynamic typing into a static typing discipline. To do so, a new type constructor ? is introduced to represent dynamic type information. At typing time, this ? should be seen as a wildcard that represents "any type" that is to be treated optimistically. In particular, ? should be considered convertible to any type. Conversion is thus replaced by a new relation, called consistency, that corresponds to this intuition. In effect it behaves somewhat similarly to unification, with each ? corresponding to a unification variable. This means in particular that consistency is *not* transitive, as if it were it would be useless: since any type T is consistent with ?, if consistency were transitive any two types would be related.

Localized computation. The free-standing conversion rule Conv is powerful, but sometimes too much.

This was our first use for the bidirectional structure. Indeed, multiple uses of a consistency in a row would have allowed to change the type of a term to any other arbitrary type by going through ? using two conversion rules in a row. Thus, any term would have been typable! Being able to use the conversion rule unrestricted was too much. Instead, the bidirectional system enforces a more localized use of conversion: only once, at the interface between inference and checking. This restriction was enough to make the conversion rule meaningful again.

More generally, since the equivalence between the undirected and directed variants relies on the properties listed in Proposition 1, when one of these fails the equivalence is endangered. When one envisons a system where this would be the case, the bidirectional approach might be worth considering, as it could stay viable while its undirected counterpart might not.

Modes for the conversion rule. The observation made in Section 2.2 that the unique Conv rule serves two different roles, which is clarified by the separation between checking and constrained inference, is an important one when toying with computation. Indeed, those two different aspects must be accounted for if one wishes to modify conversion and/or reduction. In particular, modifying the definition of conversion without accounting for the specific role of reduction would make rules for checking and constrained inference come out of sync, bringing trouble down the road.

Taking again the example of [13], the Check is modified by directly replacing conversion with the consistency relation usual in gradual typing. But this is not enough, because constrained inference must be handled as well. This is done by supplementing rule Sort-Inf by another rule to treat the case when the inferred type reduces to the wildcard ?, that can be used as a type – with some care taken. The same happens for all constrained inference rules.

Bidirectional elaboration. In works such as [20, 7, 13], the procedure described is not typing but rather elaboration: the subject of the derivation t is in a kind of source syntax and the aim is not only to inspect t, but also to output a corresponding t' in another target syntax. The term t' is a more precise account of term t, for instance with solved meta-variables, inserted coercions, and so on. The bidirectional structure readily adapts to those settings, with the extra term t' simply considered as an output of all judgements. As such, McBride's discipline as described in Section 2.2 demands that when, in a context Γ, the subject t elaborates to t' while inferring type T, we should have $\Gamma \vdash t' : T$ – and similarly for all other typing judgements. Having all rules locally preserve this invariant ensures that elaborated terms are always well-typed.

5 Related work

5.1 Constrained inference

Traces of constrained inference in diverse seemingly ad-hoc workarounds can be found in various works around typing for CIC, illustrating that this notion, although overlooked, is of interest.

In [19], $\Gamma \vdash t : T$ is used for what we write $\Gamma \vdash t \triangleright T$, but another judgment written $\Gamma \vdash t :\geq T$ and denoting type inference followed by reduction is used to effectively inline the two hypothesis of our constrained inference rules. Checking is similarly inlined.

Saïbi [20] describes an elaboration mechanism inserting coercions between types. Those are inserted primarily in checking, when both types are known. However he acknowledges the presence of two special classes to handle the need to cast a term to a sort or a function type without more informations, exactly in the places where we resort to constrained inference rather than checking.

More recently, Sozeau [22] describes a system where conversion is augmented to handle coercion between subset types. Again, $\Gamma \vdash t : T$ is used for inference, and the other judgments are inlined. Of interest is the fact that reduction is not enough to perform constrained inference, because type head constructors can be hidden by the subset construction: a term of subset type such as $\{f : \mathbb{N} \to \mathbb{N} \mid f\ 0 = 0\}$ should be usable as a function of type $\mathbb{N} \to \mathbb{N}$. An erasure procedure is therefore required on top of reduction to remove subset types in the places where we use constrained inference.

Abel and Coquand [4] use a judgement written $\Delta \vdash V\delta \Uparrow \mathrm{Set} \rightsquigarrow i$, where a type V is checked to be well-formed, but with its exact level i free. This corresponds very closely to our use of $\triangleright_{\square}$.

Traces can also be found in the description of Matita's elaboration algorithm [7]. Indeed, the presence of meta-variables on top of coercions makes it even clearer that specific treatment of what we identified as constrained inference is required. The authors introduce a special judgement they call type-level enforcing corresponding to our $\triangleright_{\square}$ judgement. As for $\triangleright_{\Pi}$, they have two rules to apply a function, one where its inferred type reduces to a product, corresponding to PROD-INF, and another one to handle the case when the inferred type instead reduces to a meta-variable. As Saïbi, they also need a special case for coercions of terms in function and type position. However, their solution is different. They rely on unification, which is available in their setting, to introduce new meta-variables for the domain and codomain of a product type whenever needed. For $\triangleright_{\square}$ though this solution is not viable, as one would need a kind of universe meta-variable. Instead, they rely on backtracking to test multiple possible universe choices.

Finally, we have already mentioned [13] in Section 4, where the bidirectional structure is crucial in describing a gradual extension to CIC. In particular, and similarly to what happens with meta-variables in [7], all constrained inference rules are duplicated: there is one rule when the head constructor is the desired one, and a second one to handle the gradual wildcard.

5.2 Completeness

Quite a few articles tackle the problem of bidirectional typing in a setting with an untyped – so called Curry-style – abstraction. This is the case of early work by Coquand [11], the type system of Agda as described in [17], the systems considered by Abel in many of his papers [3, 4, 2, 5], and much of the work of McBride [14, 15, 16] on the topic. In such systems,

λ-abstractions can only be checked against a given type, but cannot infer one, so that only terms with no β-redexes are typable. Norell [17] argues that such β-redexes are uncommon in real-life programs, so that being unable to type them is not a strong limitation in practice. To circumvent this problem, McBride also adds the possibility of typing annotations to retain the typability of a term during reduction.

While this approach is adapted to programming languages, where the emphasis is on lightweight syntax, it is not tenable for a proof assistant kernel, where all valid terms should be accepted. Indeed, debugging a proof that is rejected because the kernel fails to accept a perfectly well-typed term the user never wrote – as most proofs are generated rather than written directly – is simply not an option.

In a setting with typed – Church-style – abstraction, if one wishes to give the possibility for seemingly untyped abstraction, another mechanism has to be resorted to, typically meta-variables. This is what is done in Matita [7], where the authors combine a rule similar to Abs – where the type of an abstraction is inferred – with another one, similar to the Curry-style one – where abstraction is checked – looking like this:

$$\frac{T \rightsquigarrow^* \Pi\, x : A'.B \qquad \Gamma \vdash A \triangleright_{\Box} \Box_i \qquad A \equiv A' \qquad \Gamma, x : A \vdash t \triangleleft B}{\Gamma \vdash \lambda\, x : A.t \triangleleft T}$$

While such a rule would make a simple system such as that of Section 2 "over-complete", it is a useful addition to enable information from checking to be propagated upwards in the derivation. This is crucial in systems where completeness is lost, such as Matita's elaboration. Similar rules are described in [7] for let-bindings and constructors of inductive types.

Although only few authors consider the problem of a complete bidirectional algorithm for type-checking dependent types, we are not the first to attack it. Already Pollack [19] does, and the completeness proof for CCω of Section 2 is very close to one given in his article. Another proof of completeness for a more complex CIC-like system can be found in [22]. None of those however tackle as we do the whole complexity of PCUIC.

5.3 Inputs and outputs

We already credited the discipline we adopt on well-formedness of inputs and outputs to McBride [15, 16]. A similar idea has also appeared independently in [9]. Bauer and his co-authors introduce the notions of a (weakly) presuppositive type theory [9, Def. 5.6] and of well-presented premise-family and rule-boundary [9, Def. 6.16 and 6.17] to describe a discipline similar to ours, using what they call the boundary of a judgment as the equivalent of our inputs and outputs. Due to their setting being undirected, this is however more restrictive, because they are not able to distinguish inputs from outputs and thus cannot relax their condition to only demand inputs to be well-formed but not outputs.

6 Conclusion

We have described a judgmental presentation of the bidirectional structure of typing algorithms in the setting of dependent types. In particular, we identified a new family of judgements we called constrained inference. Those have no counterpart in the non-dependent setting, as they result from a choice of modes for the conversion rule, which is specific to the dependent setting. We proved our bidirectional presentation equivalent to an undirected one, both on paper on the simple case of CCω, and formally in the much more complex and realistic setting of PCUIC. Finally, we gave various arguments for the usefulness of our presentation as a way to ease proofs, an intermediate between undirected type-systems and typing algorithms, a solid basis to design new type systems, and a tool to re-interpret previous work on type systems in a clearer way.

Regarding future work, a type-checking algorithm is already part of MetaCoq, and we should be able to use our bidirectional type system to give a pleasant completeness proof by separating the concerns pertaining to bidirectionality from the algorithmic problems, such as implementation of an efficient conversion check or proof of termination. More broadly, bidirectional type systems should be an interesting tool in the feat of incorporating in proof assistants features that have been satisfactorily investigated on the theoretical level while keeping a complete and correct kernel, avoiding the pitfall of cumulative inductive type's incomplete implementation in Coq. A first step would be to investigate the discrepancies between the two kinds of presentations of inductive types Section 3, and in particular if all informations currently stored in the match node are really needed or if a more concise presentation can be given. But we could go further and study how to handle cubical type theory [28], rewrite rules [10], setoid type theory [6], exceptional type theory [18], η-conversion… There might also be an interesting link to make with the current work on normalization by evaluation [1] as an alternative to weak-head reduction for constrained inference. Finally, we hope that our methodology will be adapted as a base for other theoretical investigations. As a way to ease this adoption, studying it in a general setting such as that of [9] might be a strong argument for adoption.

References

1 Andreas Abel. Towards normalization by evaluation for the $\beta\eta$-calculus of constructions. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*, volume 6009 of *Lecture Notes in Computer Science*, pages 224–239. Springer-Verlag, 2010. doi:10.1007/978-3-642-12251-4.

2 Andreas Abel and Thorsten Altenkirch. A partial type checking algorithm for type:type. *Electronic Notes in Theoretical Computer Science*, 229(5):3–17, 2011. Proceedings of the Second Workshop on Mathematically Structured Functional Programming (MSFP 2008). doi:10.1016/j.entcs.2011.02.013.

3 Andreas Abel and Thierry Coquand. Untyped algorithmic equality for Martin-Löf's logical framework with surjective pairs. *Fundamenta Informaticae*, 77(4):345–395, 2007. TLCA'05 special issue.

4 Andreas Abel, Thierry Coquand, and Peter Dybjer. Verifying a semantic $\beta\eta$-conversion test for Martin-Löf type theory. In Philippe Audebaud and Christine Paulin-Mohring, editors, *Mathematics of Program Construction*, pages 29–56, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

5 Andreas Abel, Joakim Öhman, and Andrea Vezzosi. Decidability of conversion for type theory in type theory. *Proc. ACM Program. Lang.*, December 2017. doi:10.1145/3158111.

6 Thorsten Altenkirch, Simon Boulier, Ambrus Kaposi, and Nicolas Tabareau. Setoid type theory - a syntactic translation. In *MPC 2019 - 13th International Conference on Mathematics of Program Construction*, volume 11825 of *LNCS*, pages 155–196. Springer, October 2019. doi:10.1007/978-3-030-33636-3_7.

7 Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A Bi-Directional Refinement Algorithm for the Calculus of (Co)Inductive Constructions. *Logical Methods in Computer Science*, Volume 8, Issue 1, 2012. doi:10.2168/LMCS-8(1:18)2012.

8 Henk Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*. Oxford Universiy Press, 1992.

9 Andrej Bauer, Philipp G. Haselwarter, and Peter LeFanu Lumsdaine. A general definition of dependent type theories. Preprint, 2020. arXiv:2009.05539.

10 Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. The Taming of the Rew: A Type Theory with Computational Assumptions. *Proceedings of the ACM on Programming Languages*, 2021. doi:10.1145/3434341.

11 Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1), 1996. doi:10.1016/0167-6423(95)00021-6.

12 Trevor Jim. What are principal typings and what are they good for? In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 42–53, New York, NY, USA, 1996. Association for Computing Machinery. doi:10.1145/237721.237728.

13 Meven Lennon-Bertrand, Kenji Maillard, Nicolas Tabareau, and Éric Tanter. Gradualizing the calculus of inductive constructions. Preprint, 2020. arXiv:2011.10618.

14 Conor McBride. I got plenty o' nuttin'. In Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella, editors, *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 207–233. Springer International Publishing, 2016. doi:10.1007/978-3-319-30936-1_12.

15 Conor McBride. Basics of bidirectionalism. Blog post, August 2018. URL: https://pigworker.wordpress.com/2018/08/06/basics-of-bidirectionalism/.

16 Conor McBride. Check the box! In *25th International Conference on Types for Proofs and Programs*, June 2019.

17 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

18 Pierre-Marie Pédrot and Nicolas Tabareau. Failure is not an option an exceptional type theory. In *ESOP 2018 - 27th European Symposium on Programming*, volume 10801 of *LNCS*, pages 245–271, Thessaloniki, Greece, 2018. Springer. doi:10.1007/978-3-319-89884-1_9.

19 R. Pollack. Typechecking in Pure Type Systems. In *Informal Proceedings of the 1992 Workshop on Types for Proofs and Programs, Båstad, Sweden*, pages 271–288, June 1992. URL: http://homepages.inf.ed.ac.uk/rpollack/export/BaastadTypechecking.ps.gz.

20 Amokrane Saïbi. Typing algorithm in type theory with inheritance. *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '97*, 1997. doi:10.1145/263699.263742.

21 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 274–293. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/LIPIcs.SNAPL.2015.274.

22 Matthieu Sozeau. Subset coercions in coq. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs*, pages 237–252, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. doi:10.1007/978-3-540-74464-1_16.

23 Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, February 2020. doi:10.1007/s10817-019-09540-0.

24 Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq. *Proceedings of the ACM on Programming Languages*, pages 1–28, January 2020. doi:10.1145/3371076.

25 Matthieu Sozeau and Nicolas Tabareau. Universe polymorphism in Coq. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, pages 499–514, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-08970-6_32.

26 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

27 Amin Timany and Matthieu Sozeau. Cumulative Inductive Types In Coq. In Hélène Kirchner, editor, *3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018)*, volume 108 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1–16, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.FSCD.2018.29`.

28 Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: A dependently typed programming language with univalence and higher inductive types. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. `doi:10.1145/3341691`.

A Mechanized Proof of the Max-Flow Min-Cut Theorem for Countable Networks

Andreas Lochbihler
Digital Asset (Switzerland) GmbH, Zürich, Switzerland

Abstract

Aharoni et al. [3] proved the max-flow min-cut theorem for countable networks, namely that in every countable network with finite edge capacities, there exists a flow and a cut such that the flow saturates all outgoing edges of the cut and is zero on all incoming edges. In this paper, we formalize their proof in Isabelle/HOL and thereby identify and fix several problems with their proof. We also provide a simpler proof for networks where the total outgoing capacity of all vertices other than the source is finite. This proof is based on the max-flow min-cut theorem for finite networks.

2012 ACM Subject Classification Mathematics of computing → Network flows; Theory of computation → Higher order logic; Theory of computation → Logic and verification

Keywords and phrases flow network, optimization, infinite graph, Isabelle/HOL

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.25

Related Version A full version with informal proofs and more counterexamples is available:
Full Version: `http://www.andreas-lochbihler.de/pub/lochbihler2021itpl.pdf` [18]

Supplementary Material The formalization is available in the Archive of Formal Proofs:
Software (Formalization): `http://www.isa-afp.org/entries/MFMC_Countable.shtml` [16]

Funding Swiss National Science Foundation grant 153217 "Formalising Computational Soundness for Protocol Implementations". This work was partially done while the author was at ETH Zurich.

Acknowledgements We thank Ron Aharoni and Eli Berger for helping to clarify the weaknesses in the original proofs. S. Reza Sefidgar and the anonymous reviewers helped to improve the presentation.

1 Introduction

The max-flow min-cut (MFMC) theorem for finite networks [10] has wide-spread applications: network analysis, optimization, scheduling, etc. Aharoni et al. [3] have generalized this theorem to countable networks, i.e., graphs with countably many vertices and edges, as follows:

▶ **Theorem 1.** *Let $\Delta = (V, E, s, t, c)$ be a directed graph with countably many edges $E \subseteq V \times V$, vertices s and t and a capacity function $c :: E \to \mathbb{R}_{\geq 0}$. There exists a flow f and an s-t-cut C such that f saturates all outgoing edges e of C, i.e. $f(e) = c(e)$, and is 0 on all incoming edges.*

The countable MFMC theorem is used, e.g., in probability [22] and programming language theory [17], privacy [7], and for random walks [21]. Here, we formalize this theorem in Isabelle.

Traditionally, the max-flow min-cut theorem is stated in terms of equality of values: The value of the maximum flow is equal to the value of the minimum cut. Here, a flow $f :: E \Rightarrow \mathbb{R}_{\geq 0}$ assigns values to the edges of Δ such that the incoming and outgoing amounts in every vertex are the same, except for the source s and the sink t. The value $|f|$ is the amount that leaves the source s, i.e., $|f| = \sum_{x \in \mathrm{OUT}(s)} f(s, x)$ where $\mathrm{OUT}(x) = \{y \mid (x, y) \in E\}$. Dually, an s-t-cut partitions the vertices into two sets $(C, V - C)$ such that C contains the source s but not the sink t. Its value $|C|$ is the total capacity of the edges that leave C: $|C| = \sum_{e \in \mathrm{OUT}(C)} c(e)$ where $\mathrm{OUT}(C) = \{(x, y) \in E \mid x \in C \wedge y \notin C\}$.

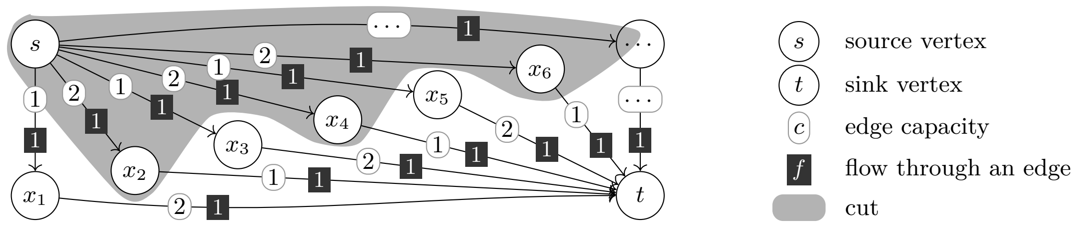


Figure 1 A countable network with a flow and a cut of infinite value.

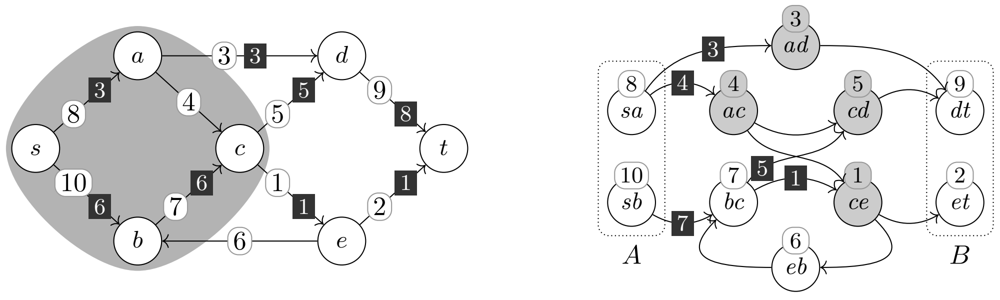
For finite networks, the equality-of-values condition $|f| = |C|$ is equivalent to the flow f saturating the cut C. In infinite networks, the saturation condition is preferable. For example, Fig. 1 shows a network with source s and sink t and countably many vertices x_i. The edge capacities are given as white rounded rectangles on the edges. The black rectangles denote a flow f and the vertices in the grey area form a cut C. The flow f saturates the outgoing edges of C and we have $|f| = \infty = |C|$. However, there is another flow g given by $g(e) = {}^1\!/\!{}_2 f(e)$ that sends only half the amount of f. Still, $|g| = \infty = |C|$. So the equality-of-values condition does not distinguish between f and g. Yet, we should consider only f a maximum flow, not g, as one can obviously increase g on some edges. The cut-saturation condition achieves this as it compares the finite capacities of individual edges with the flow through them.

This subtlety highlights the main challenge in proving the max-flow min-cut theorem for countable networks: avoiding infinite summations. Aharoni et al.'s proof performs an elaborate dance around this problem, transforming the network several times on the way. Our formalization follows these steps through all the transformations (Sect. 3) until the problem is reduced to finding some sort of matching in an infinite bipartite graph. The original proof then jumps back to arbitrary networks. Our proof forks into two proofs: The first takes a shortcut to a significantly simpler argument based on the max-flow min-cut theorem for finite networks (Sect. 4.1). This shortcut works only for networks where the sum of the capacities of the outgoing edges of any vertex other than the source is finite. This condition is met in some applications [7, 17]. The second proof follows the original (Sect. 4.2).

Our main contributions are as follows:

- We have formalized Aharoni et al.'s strong version of the max-flow min-cut theorem for countable networks in Isabelle/HOL. The resulting formalization is usable in other formalizations; e.g., we have applied it to the problem of proving parametricity of a probabilistic programming language with recursion [17]. The formalization has clarified the definitions and theorems and has revealed several problems in the original proofs (Sect. 6), which we have fixed. In particular, the reduction to bipartite graphs did not work as expected and required more general theorems.
- We give an alternative proof for the case when every inner vertex of a network has only finite total outgoing capacity. This local boundedness assumption allows us to reuse Lammich and Sefidgar's formalization of the max-flow min-cut theorem for finite networks [14] by applying a majorised convergence argument. This proof is considerably simpler and suffices for some use cases in programming languages and privacy [7, 17].

Neither of the two proofs requires a large background theory; basic notions like infinite summations, monotone and majorised convergence, and fixpoints of increasing functions suffice. The formalization therefore does not rely on specific Isabelle/HOL features and could have been done similarly in other systems like HOL4 and Coq.



■ **Figure 2** Example of a network (left) and a flow (values of 0 are omitted) with an orthogonal cut, and the corresponding web (right) with a maximal wave (black rectangles) and its set of terminal vertices (grey circles). Capacities and weights are shown as labels in rounded rectangles.

The formalization started in 2015 and a first version was published in the Archive of Formal Proofs in 2016. This paper describes the cleaned-up version for Isabelle2021 [16], which also includes the simpler proof for the bounded case. This paper first presents the corrected proof using conventional mathematical notation (Sects. 2–4). We discuss the formalization aspects in Sect. 5 and the problems with the original proof in Sect. 6.

2 Graphs, Networks, and Webs

In this section, we introduce the relevant notions for graphs, networks, and webs. The terminology and notation follows [3] to ease the comparison and make the presentation accessible to mathematicians. Formalization considerations will be discussed in Sect. 5.

▶ **Definition 2** (Graph). *A* (directed) graph $G = (V, E)$ *consists of a set of vertices* V *and a set of directed edges* $E \subseteq V \times V$. *A graph is countable iff its set of edges is countable. The neighbours of a vertex* $x \in V$ *are given by* $\mathrm{OUT}_G(x) = \{\, y \mid (x, y) \in E \,\}$ *and* $\mathrm{IN}_G(x) = \{\, y \mid (y, x) \in E \,\}$. *If the graph* G *is obvious from the context, we drop the subscript* G.

Given a function $f :: E \to \mathbb{R}_{\geq 0}$, *the* in-degree $d^-_f :: V \to \mathbb{R}^\infty_{\geq 0}$ *of* f *given by* $d^-_f(x) = \sum_{y \in \mathrm{IN}(x)} f(y, x)$ *assigns to each vertex* $x \in V$ *the sum of* f *over all incoming edges to* x. *Analogously,* $d^+_f(x) = \sum_{y \in \mathrm{OUT}(x)} f(x, y)$ *denotes* f*'s* out-degree *of* $x \in V$. *If* $d^+_f(x) = 0$, *then* x *is a* sink *for* f. *The set* $\mathrm{SINK}(f)$ *denotes the set of sinks for* f.

▶ **Definition 3** (Network). *A* network $\Delta = (V, E, s, t, c)$ *is a graph* (V, E) *with two dedicated vertices, the source* s *and the sink* t, *and a capacity function* $c :: E \to \mathbb{R}_{\geq 0}$. *A network is countable iff the graph is countable.*

▶ **Definition 4** (Flow). *For a network* $\Delta = (V, E, s, t, c)$, *a* flow $f :: E \to \mathbb{R}_{\geq 0}$ *in* Δ *satisfies*
1. *(Capacity restriction)* $f(x, y) \leq c(x, y)$ *for all* $(x, y) \in E$, *and*
2. *(Kirchhoff's 1st law)* $d^-_f(x) = d^+_f(x)$ *for all* $x \in V - \{\, s, t \,\}$.

The value $|f|$ *of a flow* f *is* f*'s out-degree of* s: $|f| = d^+_f(s)$.

▶ **Definition 5** (Orthogonal cut). *In a network* $\Delta = (V, E, s, t, c)$, *a set of vertices* C *is a* cut *iff* $s \in C$ *and* $t \notin C$. *A cut* C *is* orthogonal *to a flow* f *iff* f *saturates all edges going out of* C *(i.e.,* $f(x, y) = c(x, y)$ *for all* $(x, y) \in E$ *with* $x \in C$ *and* $y \notin C$*) and* f *is zero on all edges entering* C *(i.e.,* $f(x, y) = 0$ *for all* $(x, y) \in E$ *with* $x \notin C$ *and* $y \in C$*).*

We have already seen an orthogonal pair of a flow of infinite value and a cut in Fig. 1. Another example of an orthogonal flow-cut pair of value 9 is shown in Fig. 2 on the left.

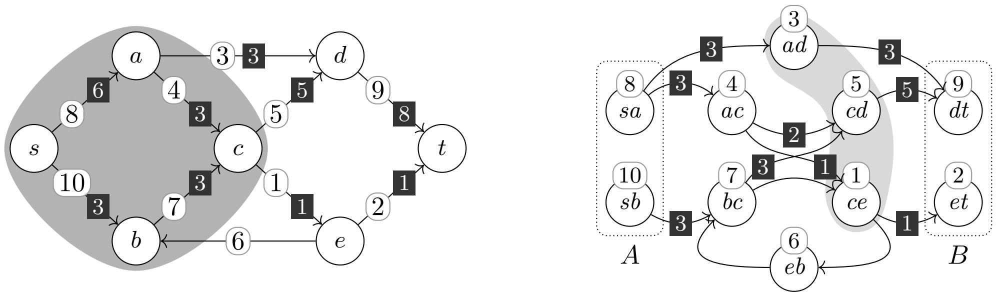


Figure 3 The network and web from Fig. 2 with a different flow (left) and a web-flow (right).

A network constrains the capacities of the edges in a graph, but the throughput of a vertex is unconstrained. So the sums on the two sides of Kirchhoff's first law may be infinite. To avoid such infinite sums, a web constrains the throughput of a vertex and leaves the edge capacity unconstrained. Section 3.1 explains how to convert between networks and webs.

▶ **Definition 6** (Web). *A web $\Gamma = (V, E, A, B, w)$ is a graph (V, E) with two sets of vertices $A, B \subseteq V$ (the sides A and B) and a weight function $w :: V \to \mathbb{R}_{\geq 0}$. We refer to the components of Γ by V_Γ, E_Γ, A_Γ, B_Γ, and w_Γ.*

The two vertex sets A and B correspond to the source and sink of a network, respectively. Currents in a web take the role of flows in a network. The difference is that vertices may leak some of the incoming current (condition 2), i.e., they need not preserve the current.

▶ **Definition 7** (Current). *Given a web $\Gamma = (V, E, A, B, w)$, a* current *$f :: E \to \mathbb{R}_{\geq 0}$ satisfies*
1. *(weight restriction) $d_f^-(x) \leq w(x)$ and $d_f^+(x) \leq w(x)$ for all $x \in V$,*
2. *(flow reflection) $d_f^-(x) \geq d_f^+(x)$ for all $x \in V - A$, and*
3. *(side restriction) $d_f^-(x) = 0$ for $x \in A$ and $d_f^+(y) = 0$ for $y \in B$.*

A current f is called a web-flow *if $d_f^-(x) = d_f^+(x)$ for all $x \in V - (A \cup B)$. If $d_f^+(x) \geq w(x)$, then f* exhausts *x. If $x \in A$ or $d_f^-(x) \geq w(x)$, then f* saturates *x. A saturated sink x is called* terminal. *The set of saturated vertices is written as $\mathrm{SAT}(f)$ and the set of terminal vertices as $\mathrm{TER}(f) = \mathrm{SAT}(f) \cap \mathrm{SINK}(f)$.*

Figure 2 shows an example web on the right where the weight of the vertices are shown in rounded rectangles. It is derived from the network on the left as we will see in Sect. 3.1. The black rectangles specify a current f whose terminal vertices $\mathrm{TER}(f)$ are shown in grey. It exhausts none of the vertices. The current f is not a web-flow because some vertices are leaking, e.g., $d_f^-(bc) = 7 > 6 = d_f^+(bc)$.

Figure 3 shows a different flow and current for same network and web, respectively. The flow on the left differs from the one in Fig. 2 only in that three units are routed through (s, a) and (a, c) instead of through (s, b) and (b, c). So the vertex c now mixes the units coming from a with the three units coming from b and outputs five of them to d and one to e. On the right, a web-flow is shown, which refines the flow on the left as will be explained in Sect. 3.1. The light-grey area contains the exhausted vertices, namely ad, cd, and ce. There are no terminal vertices as the three sinks dt, et, and eb are disjoint from the saturated vertices sa, sb, ad, cd, and ce.

▶ **Definition 8** (Essential vertex). *Given sets of vertices S and B in a graph $G = (V, E)$, a vertex $x \in S$ is* essential *in S iff there is a path from x to a vertex in B which does not contain a vertex in $S - \{x\}$. The set of essential vertices of S is written as $\mathcal{E}_{G,B}(S)$.*

▶ **Definition 9** (Separation and roofing). *A set S of vertices in graph G* separates *a vertex x from a set of vertices B iff every path from x to a vertex in B contains a vertex in S. The set S is said to* separate *a set of vertices A from B iff it separates every vertex in A from B.*

The roofing *of S and B (notation $\mathrm{RF}_{G,B}(S)$) consists of all vertices which S separates from B. The* strict roofing *excludes essential vertices: $\mathrm{RF}^{\circ}_{G,B}(S) = \mathrm{RF}_{G,B}(S) - \mathcal{E}_{G,B}(S)$.*

In a web $\Gamma = (V, E, A, B, w)$, S is A-B-separating *iff it separates A and B. If f is a current in Γ, we abbreviate $\mathcal{E}(f) = \mathcal{E}_{\Gamma,B}(\mathrm{TER}(f))$ and $\mathrm{RF}(f) = \mathrm{RF}_{\Gamma,B}(\mathrm{TER}(f))$ and $\mathrm{RF}^{\circ}(f) = \mathrm{RF}^{\circ}_{\Gamma,B}(\mathrm{TER}(f))$.*

In the web in Fig. 2, the grey vertices $\mathrm{TER}(f)$ separate A from B. The vertex ac is not essential in $\mathrm{TER}(f)$ as all paths from ac to B pass either through cd or ce, which are both in $\mathrm{TER}(f)$. The roofing $\mathrm{RF}(f)$ contains all the vertices to the left of ad, cd, and ce, inclusive, i.e., $\mathrm{RF}(f) = \{sa, sb, ac, bc, ad, eb, cd, ce\}$. The strict roofing $\mathrm{RF}^{\circ}(f)$ excludes the essential vertices ad, eb, and ce. Since ac is not essential in $\mathrm{TER}(f)$, the strict roofing includes ac.

▶ **Lemma 10** ([2, Lemma 2.14]). *If S separates A from B in G, so does $\mathcal{E}_{G,B}(S)$.*

The key tool for the proof is the concept of a wave. Waves are currents whose terminal vertices separate A from B and which are zero outside of the roofing of the terminal vertices. Intuitively, a wave's essential terminal vertices identify a bottleneck in the web: since the wave saturates them, all other separating sets between the A side and the terminal vertices must allow at least the same current.

▶ **Definition 11** (Wave). *A current f in Γ is a* wave *iff $\mathrm{TER}(f)$ is A-B-separating and $d^{+}_{f}(x) = 0$ for $x \notin \mathrm{RF}(f)$.*

In Fig. 2, the current f is 0 outside of $\mathrm{RF}(f)$, i.e., on the edges entering B. So f is a wave. Conversely, the web-flow g in Fig. 3 is not a wave as $\mathrm{TER}(g) = \{\}$ does not separate A from B.

3 From Networks to Bipartite Webs and Back

Aharoni et al.'s proof proceeds in four steps [3]:

1. Transform the network into a web.
2. Find a maximal wave in the web. Its roofing determines the cut.
3. Trim the wave, i.e., reduce the wave such that strictly roofed vertices preserve the current.
4. Extend the wave to a web-flow. This uses a reduction to bipartite webs in which every current is a web-flow by definition.

In this section, we cover these steps up to the reduction to bipartite webs. The next section takes care of actually finding a suitable current in the bipartite web.

3.1 From Networks to Webs

The first step reduces a network Δ to a web, which we denote by web(Δ). Every edge e becomes a vertex of web(Δ) with weight $c(e)$. Every two incident edges (x, y) and (y, z) in the network induce an edge between the vertices (x, y) and (y, z) in web(Δ). The side A consists of the edges leaving s and B of the edges entering t. Formally:

$$V_{\mathrm{web}(\Delta)} = E_{\Delta} \qquad w_{\mathrm{web}(\Delta)}(e) = c(e) \qquad A_{\mathrm{web}(\Delta)} = \{(s, y) \mid (s, y) \in E_{\Delta}\}$$
$$E_{\mathrm{web}(\Delta)} = \{((x, y), (y, z)) \mid (x, y) \in E_{\Delta} \wedge (y, z) \in E_{\Delta}\} \qquad B_{\mathrm{web}(\Delta)} = \{(x, t) \mid (x, t) \in E_{\Delta}\}$$

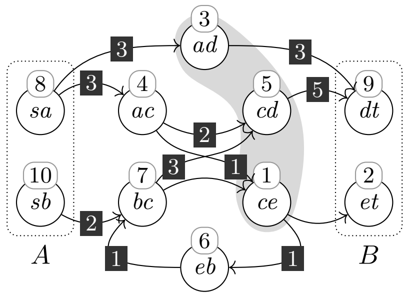


Figure 4 A separating set (grey area) that is not orthogonal to the shown web-flow.

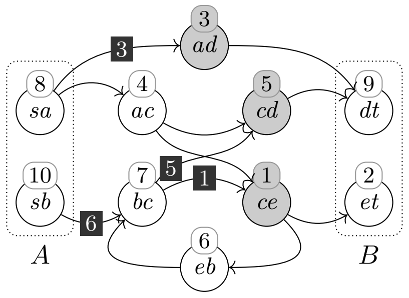


Figure 5 A trimming of the wave from Fig. 2.

For example, Figs. 2 and 3 show the same network Δ on the left and the corresponding web web(Δ) on the right. Webs have the advantage over networks that the current makes explicit how the incoming flow is split up into the outgoing edges of a vertex. In Fig. 3, e.g., the web-flow on the right specifies that the three units flowing from *sa* to *ac* split up into two units going to *cd* and one unit going to *ce*. The flow in the network on the left cannot express this detail: the vertex c mixes the two incoming flows of 3 units each and distributes somehow into five and one outgoing units.

Webs therefore allow us to capture flow preservation more precisely than networks. For if a flow f through a network vertex x is infinite, then flow preservation at x merely states that both sums are infinite: $d^-_f(x) = d^+_f(x) = \infty$. This creates problems if we want to subtract two infinite flows f and g from one another because $d^-_f(x) - d^-_g(x) = \infty - \infty$ is not meaningful. So even if both f and g satisfy Kirchhoff's first law at a vertex, it is not clear that their difference $f - g$ satisfies it. In the corresponding web, in contrast, a web-flow g specifies precisely the finite amount each incoming edge contributes to each outgoing edge. So for a web-flow or current g, the sums $d^-_g(x)$ and $d^+_g(x)$ are finite because they are bounded by the finite vertex weights, i.e., the edge capacities in the network. Accordingly, subtraction of flows has nice algebraic properties such as $d^-_f(x) - d^-_g(x) = d^-_{f-g}(x)$ if $f \geq g$.

We next transfer the orthogonality notion from networks to webs. We show that an A-B-separating set S and an orthogonal web-flow f in web(Δ) induce a cut $\hat{S}$ and an orthogonal flow $\hat{f}$ in the original network Δ. Figure 3 illustrates the reduction: The flow $\hat{f}$ in the network Δ on the left corresponds to the web-flow f in web(Δ) on the right. The set $\mathcal{E}(\mathrm{SAT}(f))$ in grey on the right is orthogonal to the web-flow f and yields the cut $\hat{S}$ on the left.

▶ **Definition 12** (Orthogonal current). *Let $\Gamma = (V, E, A, B, w)$ be a web. A set of vertices S is* orthogonal *to a current f iff*

(i) $d^-_f(x) = w(x)$ *for* $x \in S - A$,
(ii) $d^+_f(x) = w(x)$ *for* $x \in (S \cap A) - B$, *and*
(iii) $f(x, y) = 0$ *for* $x \in V - \mathrm{RF}^\circ(S)$ *and* $y \in \mathrm{RF}(S)$.

Intuitively, an orthogonal current exhausts the vertices in S unless the vertex belongs to both sides. Condition (iii) ensures that nothing flows back into the roofed vertices. For example, the web-flow in Fig. 4 is not orthogonal to the vertices in the grey area, because one unit flows from the essential vertex *ce* back to the roofed vertex *eb*.

▶ **Lemma 13** (Reduction from networks to webs). *Let $\Delta = (V, E, s, t, c)$ be a network with $s \neq t$ and no outgoing edge from t and no direct edge from s to t. Suppose that all edges have positive capacity, i.e., $c(e) > 0$ for $e \in E$.*

(a) *Let f be a web-flow in* $\mathrm{web}(\Delta)$. *Define $\hat{f}$ by $\hat{f}(e) = \max(d_f^+(e), d_f^-(e))$ for $e \in E$. Then, $\hat{f}$ is a flow in Δ.*

(b) *Let S be an A-B-separating set in* $\mathrm{web}(\Delta)$. *Define $\hat{S} = \mathrm{RF}_{\Delta,\{t\}}(\{x \mid \exists y.\, (x, y) \in \mathcal{E}(S)\})$. Then $\hat{S}$ is a cut in Δ.*

(c) *Let an A-B-separating set S be orthogonal to a web-flow f. Then $\hat{S}$ is orthogonal to $\hat{f}$.*

By this lemma, to find a cut and an orthogonal flow in a network Δ, it suffices to find a separating set of vertices in $\mathrm{web}(\Delta)$ and an orthogonal web-flow f. In the next section, we focus on finding a suitable separating set, namely the terminal vertices of a maximal wave.

3.2 Maximal Waves and Trimmings

Waves and currents can be ordered pointwise: if f and g are waves or currents in $\Gamma = (V, E, A, B, w)$, then $f \leq g$ iff $f(e) \leq g(e)$ for all $e \in E$. The waves in a countable web form a chain-complete partial order (ccpo), and so do the currents. Therefore, every countable web contains a maximal wave [3, Cor. 4.4] by Zorn's lemma.

Recall that a wave's terminal vertices describe a bottleneck in the web. Intuitively, the maximal wave identifies a narrowest bottleneck in the web: Roughly speaking, the roofed part cannot contain a tighter bottleneck because if so, the current could not saturate the terminal vertices due to the flow reflection condition. Conversely, if a separating set beyond the terminal vertices formed a tighter bottleneck, then we could extend the wave and saturate that smaller bottleneck, which contradicts maximality. Here, it is crucial that a wave may partially leak the incoming current of some vertices, i.e., they need not preserve the current.

A trimming of a wave reduces the current such that the incoming current is preserved on the strict roofing. For example, the wave in Fig. 2 on the right is maximal. Its trimming is shown in Fig. 5. The current is reduced on the edge from *sb* to *bc* from 7 to 6 and on the edge from *sa* to *ac* from 4 to 0.

▶ **Definition 14** (Trimming). *Let f be a wave in $\Gamma = (V, E, A, B, w)$. A wave g is called a* trimming *of f iff*

(i) *$g \leq f$,*

(ii) *$d_g^+(x) = d_g^-(x)$ for all $x \in \mathrm{RF}^\circ(f) - A$, and*

(iii) *$\mathcal{E}(\mathrm{TER}(g)) - A = \mathcal{E}(\mathrm{TER}(f)) - A$.*

▶ **Lemma 15** ([3, Lemma 4.8]). *Every wave in a countable web has a trimming.*

Proof. The trimming for a wave f is constructed as the transfinite fixpoint iteration of the one-step trimming function trim_1 starting at f. For a wave g, $\mathit{trim}_1(g)$ picks some strictly roofed vertex z where Kirchhoff's first law does not hold, i.e., $z \in \mathrm{RF}^\circ(g) - A \wedge d_g^+(z) \neq d_g^-(z)$. Then, trim_1 reduces the current on z's incoming edges by the factor $\frac{d_g^+(z)}{d_g^-(z)}$ so that Kirchhoff's first law holds at z afterwards.

$$\mathit{trim}_1(g)(y, x) = \begin{cases} g(y, x) & \text{if } g \text{ is a trimming} \\ \text{if } x = z \text{ then } \frac{d_g^+(z)}{d_g^-(z)} * g(y, x) \text{ else } g(y, x) & \text{if such a } z \text{ exists} \end{cases}$$

The fixpoint exists by Bourbaki-Witt's fixpoint theorem [8] as trim_1 is decreasing, i.e., $\mathit{trim}_1(g) \leq g$, and the set of waves g with $g \leq f$ is a chain-complete partial order w.r.t. $\geq$. The proof that the fixpoint satisfies the trimming conditions relies on d^+ and d^- being point-wise order-continuous, which holds by monotone convergence as the web is countable. ◀

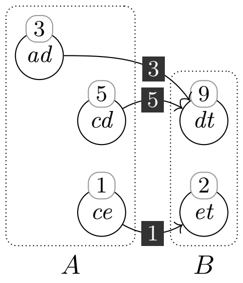


Figure 6 The quotient of the web and wave of Fig. 2 with a linkage.

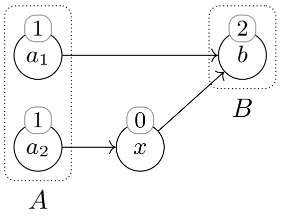


Figure 7 A web that contains no non-zero wave, but the zero wave is a hindrance.

3.3 A Linkage in the Quotient of a Web

The trimming of a maximal wave f describes the first half of the web-flow we are looking for (Fig. 5). For the second half, we consider the residual web beyond f's terminal vertices, which is called the quotient Γ/f. Figure 6 shows the quotient for the web and wave f from Fig. 2. The essential terminal vertices of the wave become the side A. The quotient does not include the roofed vertex eb even though it is reachable from $\mathcal{E}(\mathrm{TER}(f))$ as we want to construct an orthogonal current and nothing may flow back into roofed vertices. The formal definition is a bit complicated so that it also works when there are edges between vertices in $\mathcal{E}(\mathrm{TER}(f))$ or when $\mathcal{E}(\mathrm{TER}(f))$ contains vertices from B. The details are discussed in Sect. 6.

▶ **Definition 16** (Quotient). *Let $\Gamma = (V, E, A, B, w)$ and f be a wave in Γ. The quotient Γ/f is the following web:*

- $E_{\Gamma/f} = \{(x, y) \in E \mid x \notin \mathrm{RF}^{\circ}_{\Gamma}(f) \wedge y \notin \mathrm{RF}_{\Gamma}(f)\}$
- $A_{\Gamma/f} = \mathcal{E}_{\Gamma}(\mathrm{TER}_{\Gamma}(f)) - (B - A)$ *and* $B_{\Gamma/f} = B$
- $w_{\Gamma/f}(x) = w(x)$ *for* $x \in V - (\mathrm{RF}^{\circ}_{\Gamma}(f) \cup (\mathrm{TER}_{\Gamma}(f) \cap B))$ *and* $w_{\Gamma/f}(x) = 0$ *for* $x \in \mathrm{TER}_{\Gamma}(f) \cap B$.

In the quotient Γ/f, we now look for a web-flow g that saturates all vertices in A, i.e., $\mathrm{TER}(f)$. Such a web-flow is called a linkage. Then, the web-flow in Γ is given by the trimming of f plus g. Figure 6 shows such a linkage; together with the trimmed wave from Fig. 5, they form the orthogonal web-flow whose reduction (Lemma 13) yields the network flow shown in Fig. 2.

▶ **Definition 17** (Linkage [3, Def. 4.1]). *A web-flow f in a web $\Gamma = (V, E, A, B, w)$ is called a* linkage *iff f exhausts all vertices in A, i.e., $d^{+}_{f}(a) = w(a)$ for all $a \in A$.*

Under what conditions does a web Γ contain a linkage? Certainly, there must not be a bottleneck beyond the A side. Waves describe such bottlenecks. So if the zero wave is the only wave in Γ, then the A side is the only bottleneck. Moreover, we need that all vertices in A are essential for separation unless their weight is 0. For example, the web in Fig. 7 contains only the zero wave, but not a linkage. The problem is that the vertex a_2 with weight 1 is bottlenecked by the zero-weight vertex $x \in \mathcal{E}(\mathrm{TER}(\mathbf{0}))$. Such a situation is called a hindrance.

▶ **Definition 18** (Hindrance, looseness, [3, Def. 4.5]). *A wave f in a web $\Gamma = (V, E, A, B, w)$ is a* $>\varepsilon$-hindrance *iff there is a vertex $a \in A - \mathcal{E}(\mathrm{TER}(f))$ such that $\varepsilon < w(a) - d^{+}_{f}(a)$. Also, f is a* hindrance *iff there exists a $\varepsilon > 0$ such that f is a $>\varepsilon$-hindrance. A web is called* hindered *(respectively* $>\varepsilon$-hindered*) iff it contains a hindrance (respectively a $>\varepsilon$-hindrance). A web is called* loose *iff it contains no non-zero wave and the zero wave is not a hindrance.*

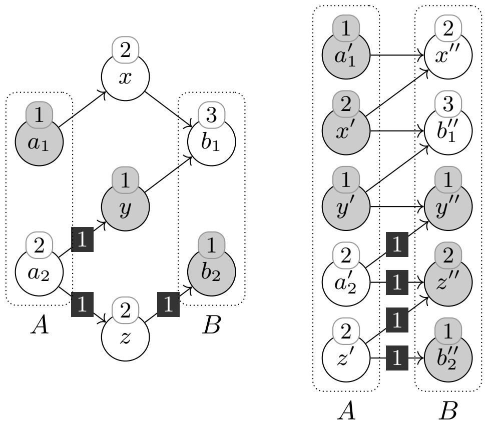


Figure 8 An unhindered web Γ (left) and its bipartite reduction bp(Γ) (right). The wave f in bp(Γ) induces the wave $\tilde{f}$ in Γ.

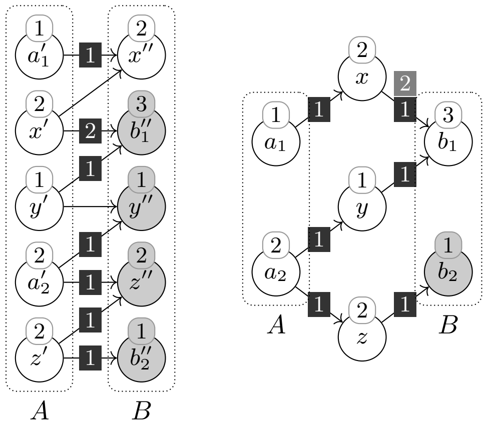


Figure 9 A linkage g in bp(Γ) (left) that yields a linkage (right) in the web Γ from Fig. 8 by trimming $\tilde{g}$ at vertex x.

▶ **Lemma 19** ([3]). *If f is a maximal wave in the web $\Gamma = (V, E, A, B, w)$, then Γ/f is loose.*

3.4 Reduction to Bipartite Webs

To find linkages in countable loose webs, Aharoni et al. [3] transform webs into bipartite webs. A web $\Omega = (V, E, A, B, w)$ is *bipartite* iff there are only edges from nodes in A to nodes in B, i.e., iff $V = A \cup B$ and $A \cap B = \emptyset$ and $E \subseteq A \times B$.

We briefly review the transformation described in [1]; Fig. 8 shows an example. In this section, we always assume that the web $\Gamma = (V, E, A, B, w)$ has no incoming edges to vertices in A, no outgoing edges from vertices in B, no loops, and that A and B are disjoint. In the bipartite web bp(Γ), there are two copies x' and x'' for every vertex $x \in V - (A \cup B)$. Vertices $x \in A$ and $y \in B$ only have one copy x' and y'', respectively. The edges are $E_{\mathrm{bp}(\Gamma)} = \{(x', y'') \mid (x, y) \in E\} \cup \{(x', x'') \mid x \in V - (A \cup B)\}$ and the sides $A_{\mathrm{bp}(\Gamma)} = \{x' \mid x \in V - B\}$ and $B_{\mathrm{bp}(\Gamma)} = \{x'' \mid x \in V - A\}$ and the weight function $w(x') = w(x)$ for $x \in V - B$ and $w(x'') = w(x)$ for $x \in V - A$.

An A-B-separating set S in bp(Γ) induces an A-B-separating set $\widetilde{S}$ in Γ given by $\widetilde{S} = (A_S \cap B_S) \cup (A \cap A_S) \cup (B \cap B_S)$ where $A_S = \{v \mid v' \in S\}$ and $B_S = \{v \mid v'' \in S\}$ [1]. Moreover, a wave f in bp(Γ) induces a wave $\tilde{f}$ in Γ given by $\tilde{f}(x, y) = f(x', y'')$ for $(x, y) \in E$ with $\mathrm{TER}_\Gamma(\tilde{f}) = \widetilde{\mathrm{TER}_{\mathrm{bp}(\Gamma)}(f)}$ [3, Lemma 6.3].

▶ **Lemma 20.** *If Γ is loose, then* bp(Γ) *is unhindered.*

Aharoni et al. wrongly claimed the stronger statement that if Γ is loose then bp(Γ) is loose [3, below Thm. 6.5]. We provide a counterexample in Sect. 6. Note that the reduction bp does not preserve unhinderedness either.

Conversely, a linkage g in bp(Γ) yields a linkage in Γ as illustrated in Fig. 9: For $\tilde{g}$ as defined above, we have $d^+_{\tilde{g}}(a) = d^+_g(a') = w(a)$ for $a \in A_\Gamma$ and $d^+_{\tilde{g}}(x) \geq d^-_{\tilde{g}}(x)$ for all $x \notin B$. So the out-flow of some vertices may surpass the in-flow, e.g., x in Fig. 9. Analogously to the trimming of waves, we can trim $\tilde{g}$ using a fixpoint iteration to obtain the linkage in Γ.

▶ **Lemma 21** ([3]). *If* bp(Γ) *contains a linkage and* Γ *is countable, then* Γ *contains a linkage.*

4 Linkability in unhindered bipartite webs

By the results in Sect. 3, the max-flow min-cut theorem for the countable case (Thm. 1) follows from the following theorem, which we prove in this section.

▶ **Theorem 22** (Bipartite linkability). *A countable unhindered bipartite web contains a linkage.*

In fact, we present two ways how to construct such a linkage in an unhindered bipartite web. Both ways enumerate the vertices in $A = \{a_1, a_2, a_3, \ldots\}$ and construct a sequence of web-flows f_i that exhaust $\{a_1, \ldots, a_i\}$ so that the limit f exhausts all of A. The difference is in how the f_i are constructed and in the limit argument. In Sect. 4.1, each f_i is constructed independently as the limit of maximum flows in a finite network; the existence and the linkage property of the limit for these f_i themselves is shown using diagonalization and majorised convergence. Unfortunately, this construction only works if the neighbours of any a_i vertex have finite total weight.

In contrast, f_{i+1} in Sect. 4.2 saturates a_{i+1} by extending the previous web-flow f_i with a sequence of augmenting flows in the so-called residual network, similar to how classic max-flow algorithms for finite networks work [9]. This construction avoids taking infinite summations and thus yields a proof of Thm. 22 without additional assumptions. However, the proof is more involved than in the bounded case.

4.1 The Bounded Case

We first prove Thm. 22 for the case where the neighbours of each vertex in A have only bounded total weight, i.e., $\sum_{y \in \mathrm{OUT}(x)} w(y) < \infty$ for all $x \in A$. The general case is shown in the next section.

The next lemma states the crucial property of unhindered bipartite webs, namely that the total weight of any finite set of A vertices is at most the total weight of their neighbours in B.

▶ **Lemma 23.** *Let $\Omega = (V, E, A, B, w)$ be a countable unhindered bipartite web and $X \subseteq A$ be finite. Then, $\sum_{x \in X} w(x) \le \sum_{y \in E[X]} w(y)$ where $E[X] = \{y \mid \exists x \in X.\ (x, y) \in E\}$ denotes the neighbours of X.*

This lemma allows us to understand a linkage in an unhindered bipartite web as an $A \times B$ matrix over the reals where the weights on A are the row sums of the countable matrix and the edges describe the matrix elements that may be non-zero. In the proof below, we will use the following result about the existence of a countable matrix with given marginals. It is a corollary of a theorem by Kellerer [12, Satz 4.1]. In the formalization, we have proved the corollary directly by adapting Kellerer's proof to this special case. This proof uses the max-flow min-cut theorem for *finite* networks.

▶ **Proposition 24** (Matrix with given marginals). *Let $f : A \to \mathbb{R}_{\ge 0}$ and $g : B \to \mathbb{R}_{\ge 0}$ for countable sets A, B such that $\sum_{i \in A} f(i) = \sum_{j \in B} g(j) < \infty$, and let $R \subseteq A \times B$. Assume that $\sum_{i \in X} f(x) \le \sum_{j \in R[X]} g(j)$ for all $X \subseteq A$. Then, there exists a function $h : A \times B \to \mathbb{R}_{\ge 0}$ such that for all $i \in A$ and $j \in B$:*

- *$h(i, j) = 0$ if $(i, j) \notin R$,*
- *$f(i) = \sum_{j \in \mathbb{N}} h(i, j)$, and*
- *$g(j) = \sum_{i \in \mathbb{N}} h(i, j)$.*

We can now prove bipartite linkability in the bounded case. The proof starts with a sequence of increasing finite subsets A_n of A that converge to A, and suitable, possibly infinite subsets B_n of their neighbours in B. For these subsets, we obtain a $A_n \times B_n$ matrix h_n with the right marginals. This sequence h_n converges and its limit yields the desired linkage, using a majorised convergence argument with the bound on the neighbours.

▶ **Theorem 25** (Bounded bipartite linkability). *A countable unhindered bipartite web $\Omega = (V, E, A, B, w)$ contains a linkage if $\sum_{y \in \mathrm{OUT}(x)} w(y) < \infty$ for all $x \in A$.*

Together with the reduction from Sect. 3, this yields a proof for Thm. 1 when only the source s in the network $\Delta = (V, E, s, t, c)$ may have outgoing edges whose total capacity is infinite, i.e., $d^+_c(x) < \infty$ for $x \in V - \{s\}$. The MFMC use cases in probability theory [22] and privacy [7] satisfy this condition.

4.2 The Unbounded Case

We now show that Thm. 22 holds even when the neighbours of a vertex have infinite total weight. Our proof generalizes Aharoni et al.'s from loose to unhindered bipartite webs. For the remainder of this section, we always assume that $\Omega = (V, E, A, B, w)$ is a countable bipartite web. We write $\Omega \ominus f$ for the bipartite web Ω where the weight of the vertices has been reduced by the current f that flows through them.

▶ **Definition 26** (Residual web). *If $\Omega = (V, E, A, B, w)$ is a bipartite web and f a current in Ω, we write $\Omega \ominus f$ for the web (V, E, A, B, w') where the new weight function w' is given by $w'(x) = w(x) - d^+_f(x)$ for $x \in A$ and $w'(x) = w(x) - d^-_f(x)$ for $x \in B$.*

The proof rests on the following step: If Ω is unhindered, then we can find a current f that saturates some vertex $a \in A$ such that the residual web $\Omega \ominus f$ is unhindered again.

▶ **Lemma 27** (Vertex saturation in unhindered bipartite webs). *If Ω is unhindered and $a \in A$, then there exists a current f in Ω such that $d^+_f(a) = w(a)$ and $\Omega \ominus f$ is unhindered.*

With this lemma, we can now prove that countable unhindered bipartite webs are linkable (Thm. 22). The proof is analogous to [3, Thm. 6.5], but uses our Lemma 27 instead.

Proof of Thm. 22. Enumerate the vertices in A as $a_1, a_2, \ldots$. Recursively define a family f_n of currents in Ω as follows:

(i) f_0 is the zero current.

(ii) For $n > 0$, pick a current g_n in $\Omega \ominus f_{n-1}$ such that $d^+_g(a_n) = w_{\Omega \ominus f_{n-1}}(a_n)$ and $\Omega \ominus f_{n-1} \ominus g$ is unhindered. Set $f_n = f_{n-1} + g$.

A simple induction on n shows that f_n is a well-defined current in Ω and $\Omega \ominus f_n$ is unhindered for all n; here, Lemma 27 applied to $\Omega \ominus f_{n-1}$ ensures that g_n exists. Set $g(e) = \sup\{f_n(e) \mid n \in \mathbb{N}\}$ for $e \in E$. Then, g is a current in Ω with $d^+_g(x) = w(x)$ for all $x \in A$. As every current in a bipartite web is a web-flow, g is the linkage we are looking for. ◀

The proof of the saturation lemma 27 uses the following theorems and lemmas, which have already been proven by Aharoni et al. [3]. We have formalized all of them and fixed the glitches in the original statements and proofs.

▶ **Theorem 28** (Flow attainability [3, Thm. 5.1]). *Let $\Delta = (V, E, s, t, c)$ be a countable network with $s \neq t$, no loops and no incoming edges to s, and such that for all $x \in V - \{t\}$, the sum of capacities of the incoming edges to x or the sum of capacities of the outgoing edges from x is finite, i.e., $d^-_c(x) < \infty$ or $d^+_c(x) < \infty$. Then there exists a flow f in Δ such that $d^+_f(s) = \sup\{|g| \mid g \text{ is a flow in } \Delta\}$ and $d^-_f(x) \leq |f|$ for all $x \in V$.*

▶ **Lemma 29** ([3, Lemma 6.7]). *Let $\Omega = (V, E, A, B, w)$ be a countable bipartite web and let $u :: V \to \mathbb{R}_{\geq 0}$ such that $u(x) = 0$ for $x \in A$, $u(y) \leq w(y)$ for $y \in B$, and $\varepsilon = \sum_{x \in B} u(x) < \infty$. Let $\Omega' = (V, E, A, B, w - u)$ be the web Ω with w reduced by u. If Ω' is $>\varepsilon$-hindered, then Ω is hindered.*

▶ **Lemma 30** ([3, Cor. 6.8]). *Let g be a current in Ω with $\varepsilon := \sum_{b \in B} d^-_g(b) < \infty$. If $\Omega \ominus g$ is $>\varepsilon$-hindered, then Ω is hindered.*

▶ **Lemma 31** ([3, Lem 6.9]). *Let Ω be loose and $b \in B$ with $w(b) > 0$. For every $\delta > 0$, there exists an $\varepsilon > 0$ such that $\varepsilon < \delta$ and Ω with the weight of b reduced by ε is unhindered.*

5 Discussion of the Formalization

We have formalized all definitions, theorems, and proofs mentioned in this paper in Isabelle/HOL. This includes all the lemmas and underlying theory. In this section, we discuss the challenges we faced and the design decisions we made. The issues with the original definitions, theorems, and proofs and their corrections are discussed in the next section.

Graphs are formalized using Isabelle's record package [20] as an extensible record with one field for the edge relation, given as a binary predicate over the vertices of type α. This yields the projection function edge :: α graph $\Rightarrow \alpha \Rightarrow \alpha \Rightarrow$ bool for the edge field.[1] From this, we derive the set E of edges as an abbreviation.

record α graph = edge :: $\alpha \Rightarrow \alpha \Rightarrow$ bool
definition vertex :: α graph $\Rightarrow \alpha \Rightarrow$ bool where vertex $G\ x = (\exists y.$ edge $G\ x\ y \lor$ edge $G\ y\ x)$
type-synonym α edge = $\alpha \times \alpha$
abbreviation E :: α graph $\Rightarrow \alpha$ edge set where $\mathsf{E}_G = \{(x, y).$ edge $G\ x\ y\}$

We derive the set of vertices from edges of the graph rather than modelling them separately. This has the advantage that we encode the condition $E \subseteq V \times V$ in the construction and do not have to carry around this well-formedness condition in our formalization. Conversely, graphs in this model cannot have isolated vertices. This is without loss of generality as isolated vertices cannot contribute to any flow or cut.

Networks are formalized as an extension of the record graph. So all operations on graphs also work for networks. The same applies to webs.

record α network = α graph +
  capacity :: $\alpha \Rightarrow$ ennreal
  source :: α
  sink :: α

record α web = α graph +
  weight :: $\alpha \Rightarrow$ ennreal
  A :: α set
  B :: α set

Records provide a simple and lightweight means for grouping the components of a network or web. Particular properties such as countability, finite capacity and weights, and disjoint sides A and B, are formalized as locales [5]. For example, the locale countable-network below enforces that there are only countably many edges, the source is not the sink, and the capacities are finite and 0 outside of the edges. Using the **(structure)** annotation on a record variable like Δ [4], we can omit the network (or web) as subscripts, e.g., in the

[1] The record package achieves extensibility with structural subtyping by internally generalizing α graph to (α, β) graph-scheme, where β is the extension slot for further fields. For example, β is instantiated with the singleton type unit for graph. All operations on graph are actually defined on graph-scheme so that they also work for all record extensions. We omit this technicality from the presentation.

assumption countable E; Isabelle automatically fills in the corresponding parameter. We use this notational convenience mainly for definitions that need custom syntax anyway, e.g., $\mathcal{E}$, RF, and RF$^{\circ}$. For plain HOL functions without special syntax like capacity and source, it is usually faster to type the record parameter than to enter special syntax.

locale countable-network = **fixes** Δ :: α network (**structure**)
  assumes countable E **and** source $\Delta \neq$ sink Δ
    and $e \notin$ E $\Longrightarrow$ capacity Δ $e = 0$ **and** capacity Δ $e < \infty$

Since flows, cuts, and capacities are always non-negative, we use the extended non-negative reals ennreal from Isabelle/HOL's library everywhere. Summations like the in-degree d^- are expressed using the Lebesgue integral nn-integral over the counting measure count-space A on the set A. So every subset of A is measurable and all points have equal weight. Moreover, every function is integrable and we need not discharge neither integrability nor summability conditions in the proofs. Just the finiteness conditions of the form $\sum_{x \in A} < \infty$ are ubiquitous.

We also formalize capacities and weights as ennreal and explicitly require them being finite in the locales. This avoids coercions from the real numbers real into ennreal, which would complicate the proof formalization. For example, the in-degree $d^-_f(f)$ of y is defined as follows where $\sum_{x \in A} g$ desugars to nn-integral (count-space A) $(\lambda x.\ g)$. We let the summation range over UNIV, the set of all values of α, not only the neighbours of y. Instead, we enforce that f is 0 outside of E, e.g., via the capacity assumption in countable-network. This way, d-IN depends only on f and not on the graph. This simplifies the formalization because when we consider f in the context of different graphs, d-IN f is trivially the same for all of them.

definition d-IN :: (α edge $\Rightarrow$ ennreal) $\Rightarrow$ α $\Rightarrow$ ennreal where d-IN f $y = \sum_{x \in \mathsf{UNIV}} f\ (x, y)$

Regarding the mathematical background theory, we found that most relevant theorems were readily available in the Isabelle/HOL library: limits, infinite summations via the Lebesgue integral, monotone and majorised convergence, lim sup and lim inf. There is even a generic formalization of Cantor's diagonalization argument by Immler [11]. The Bourbaki-Witt fixpoint theorem [8], however, was missing. We therefore ported the Coq formalization by Smolka et al. [23] to Isabelle/HOL. It is now part of Isabelle/HOL's library. We have also contributed many lemmas about ennreal and nn-integral to the library.

Apart from identifying and fixing glitches and mistakes in definitions and proofs (Sect. 6), we faced three main challenges during the formalization. First, the definition and proof principles in the paper are often not suitable for direct formalization. For example, the original proofs construct trimmings, linkages and saturating flows using transfinite iteration and transfinite induction with ordinals. We have replaced them with fixpoints of increasing or decreasing functions in a chain-complete partial order, using Bourbaki-Witt's fixpoint theorem (Lemmas 15, 21, and 27). This way, we did not need to formalize ordinals and their theory.

Second, applying the theorems from the Isabelle library often needs a small twist. The proof for the existence of a maximal wave in Sect. 3.2 demonstrates this. The proof that the least upper bound $\bigsqcup_{i \in I} f_i$ for a chain f_i of currents in a web Γ is a current relies on Beppo Levi's monotone convergence theorem. The challenge here was that the monotone convergence theorem applies only to countable increasing sequences, whereas Isabelle's formalizaton of chain-complete partial orders demands the existence of least upper bounds for arbitrary (uncountable) chains. We bridge the gap by finding a countable subsequence of any such chain, which relies on the currents being non-zero only on the countably many edges.

Third, we often faced the problem that a statement had some precondition that was not met when we wanted to apply it. In an informal proof, these preconditions would be assumed "without loss of generality" or ignored altogether. We deal with them in two ways: either

Table 1 Line counts for different parts of the formalization, not counting empty lines.

	Shared		Bounded	Unbounded
preliminaries	200	matrix for marginals (Prop. 24)	845	
networks & webs	2214	flow attainability (Thm. 28)		1954
reductions	1248	bipartite linkability (Thms. 25 / 22)	589	3158
total	3662		1434	5112

introduce a reduction that ensures the precondition or generalize the definitions and proofs so that they are not needed. Reductions are in general preferable as generalizations often complicate the definitions and proofs. Additional reductions can be seen, e.g., in Lemma 13. It assumes that there is no direct edge from s to t and all edges have positive capacity. The final theorem 1 does not make these assumptions. We therefore introduce another reduction that splits a potential s-t edge by introducing a new vertex and removes all edges with no capacity. Similarly, the reduction to bipartite webs in Sect. 3.4 assumes that the web does not contain loops. These loops would originate from loops in the original network; so we have another reduction that eliminates loops in networks. Reductions are not always feasible though. The example of the quotient web (Def. 16) is discussed in the next section.

On the positive side, reasoning about paths in networks and webs was much less of a pain than we had expected. We formalized a finite path as a list of vertices, which allows us to reuse Isabelle's library for lists to manipulate and reason about paths. For example, the predicate distinct expresses that a path does not contain cycles, and $\pi @ [x] @ \pi'$ denotes the concatenation of the two paths $\pi @ [x]$ and $[x] @ \pi'$. Moreover, we found that $\mathcal{E}$, RF, and RF$^\circ$ are powerful concepts that allow us to avoid explicitly dealing with paths in the main lemmas about flows – once we had proven enough properties about them.

Table 1 shows line counts of the Isabelle theories for different parts of the formalization, as a proxy for the formalization effort. These counts exclude empty lines. The left part lists the material that is used by both linkability proofs for bipartite webs. This covers the concepts of networks, flows, webs, currents, (maximal) waves, and trimmings, as well as the reductions from networks to webs and from webs to bipartite webs. On the right, the line counts are shown for linkability of bounded (Sect. 4.1) and unbounded (Sect. 4.2) countable bipartite webs, together with the line counts for the helper statements 24 and 28. The unbounded case requires about 3.6 times as much space as the bounded case if we include the formalization of the helper statements. If we exclude the helper statements, the ratio is about 5.4. This highlights how much more complicated the general case is.

We have also generated a PDF from the Isabelle theories using Isabelle's document preparation system. The material corresponding to shared and unbounded fill 236 pages. Aharoni et al. need a bit more than 10 pages in [3]. This gives an expansion factor of about 23. This is much higher than for text book mathematics, where the factor is typically well below 10 [6, 24]. We take this as an indication that the original paper is very dense.

6 Problems in the Original Proof

We now discuss the problems we have identified in the original paper during the formalization. We focus on three representative examples here: the reduction to bipartite webs, the definition of quotient webs, and the notion of trimmings. Further problems are given in the report [18].

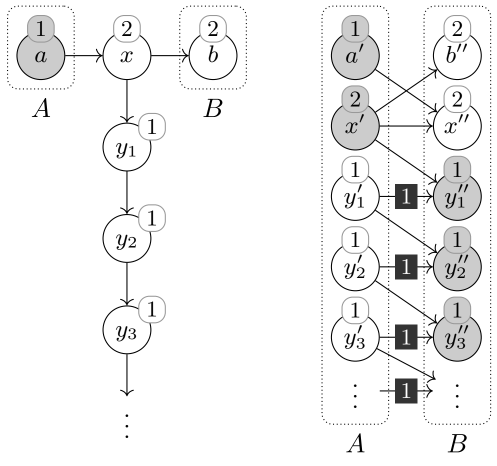


Figure 10 A loose web (left) whose bipartite reduction (right) is not loose as witnessed by the non-zero wave shown.

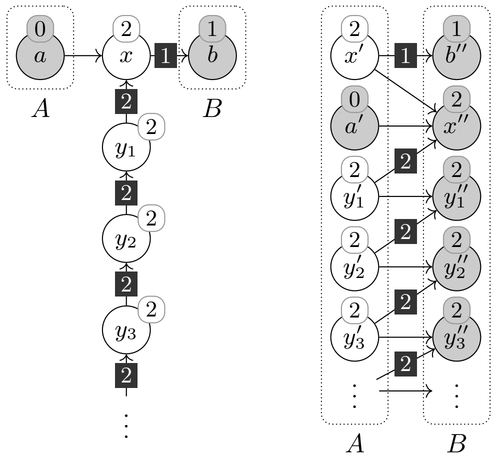


Figure 11 An unhindered web (left) whose bipartite reduction (right) contains a hindrance as witnessed at x'.

Reduction to bipartite webs. This is the main problem we have found. Aharoni et al. [3] claim that the reduction to bipartite webs from Sect. 3.4 preserves looseness, but this is not the case. In Fig. 10, the web Γ on the left is loose, its bipartite transformation $\mathrm{bp}(\Gamma)$ on the right is not loose, because it contains the non-zero wave shown. The problem is that there is no path from the (infinitely many) vertices y_i (where $i \in \mathbb{N}$) to b. In a finite web, we could remove all vertices that cannot reach a vertex in B, because they cannot contribute to a web-flow. In the infinite case, however, we cannot do so easily because such infinite paths do occur in infinite networks and absorb parts of the (maximal) flow; an example is given in the conclusion. So their key theorem [3, Thm. 6.5], namely that every countable loose bipartite web contains a linkage, cannot be used to prove the general case.

Instead, we strengthen the theorem to countable *unhindered* bipartite webs (Thm. 22). The induction invariant now is $\Omega \ominus f_n$ being unhindered rather than being loose, and the induction step (Lemma 27) must also be generalized. Fortunately, the original high-level ideas carry over; our proof composes the lemmas 29, 30 and 31 in a different order. We regain looseness from unhinderedness by first finding a maximal wave and reducing the weights, similar to what is happening in Lemma 19. Note that the reduction bp does not preserve unhinderedness either, as the example in Fig. 11 shows. The web on the left is not loose as it contains the shown wave.

Quotient webs. Quotient webs (Def. 16) are an example where the definition had to be changed. This change propagates to the proofs of the basic properties of quotient webs. In detail, the original definition sets the edges as $E_{\Gamma/f} = \{(x, y) \in E \mid x \notin \mathrm{RF}^{\circ}_{\Gamma}(f) \wedge y \notin \mathrm{RF}^{\circ}_{\Gamma}(f)\}$, i.e., an edge may point to one of f's essential terminal vertices. Our Definition 16 excludes these edges. The difference is illustrated in Fig. 12. The quotient Γ/f on the right of the web Γ and the wave f on the left contains the edge (z, x) only with the original definition. This edge invalidates a number of statements, e.g., that $f + g \upharpoonright (\Gamma/f)$ is a current or a wave if g is a current or a wave in Γ, where $g \upharpoonright (\Gamma/f)$ restricts g to the vertices of Γ/f. Take, e.g., $g(a, z) = 2$, $g(z, x) = g(z, y) = 1$, and $g(e) = 0$ otherwise.

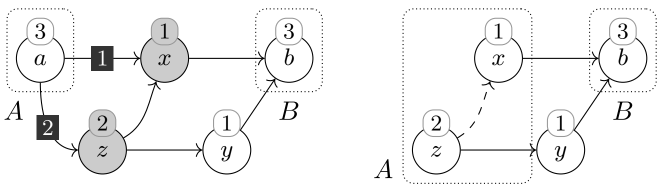


Figure 12 A wave f in a web Γ (left) and the quotient web Γ/f (right). The quotient contains the edge (z, x) only in [3].

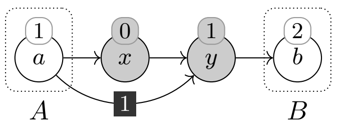


Figure 13 Wave f in a web none of whose trimmings g satisfies Aharoni et al.'s condition $\text{TER}(g) - A = \mathcal{E}(\text{TER}(f)) - A$.

Our definition therefore excludes this edge. And while we were at it, we also changed the definition of $A_{\Gamma/f}$ and the weights so that the two sides of the quotient are always disjoint and vertices without edges have weight 0. These changes ensure that the quotient web meets the assumptions of the reduction to bipartite webs (Sect. 3.4). Accordingly, we had to adapt the existing proofs about the quotient web's properties or find new ones.

Trimmings. The definition of trimmings (Def. 14) is an example of a small glitch that affects proofs only minimally. For trimmings, Aharoni et al. [3] require the stronger condition $\text{TER}(g) - A = \mathcal{E}(\text{TER}(f)) - A$ instead of $\mathcal{E}(\text{TER}(g)) - A = \mathcal{E}(\text{TER}(f)) - A$. The two are equivalent only if there are no vertices with weight 0, but webs may contain such vertices. So Lemma 15 need not hold for such webs. For example, Fig. 13 shows a wave f that does not have a trimming according to Aharoni et al.'s definition [3, Def. 4.7]. Every wave g has $x \in \text{TER}(g)$ because x has weight 0, but $x \notin \mathcal{E}(\text{TER}(f)) - A = \{y\}$.

7 Related work

Lee [15] and Lammich and Sefidgar [13, 14] have formalized the MFMC theorem for *finite* networks in Mizar and Isabelle/HOL, respectively. Lammich and Sefidgar additionally formalize and verify several max-flow algorithms. We reused Lammich and Sefidgar's formalization in our proof of Prop. 24. We make no algorithmic considerations, as countable networks are infinite objects that lie beyond the reach of traditional notions of algorithms.

Lyons and Peres [19, Thm. 3.1] consider countable locally finite networks, where every vertex has only finitely many neighbours, and without a sink. They show that the maximum flow's value equals the value of a minimum cut, where a cut here contains an edge of every infinite simple path that starts at the source. Like our proof for the bounded case, their proof extends the MFMC theorem for finite networks using majorised convergence. Since their graphs are locally finite, all summations of interest are finite by construction.

8 Conclusion

In this paper, we have formalized a strong max-flow min-cut theorem for countable networks in Isabelle/HOL. To rule out anomalities due to the network being infinite, the theorem statement avoids imprecise infinite sums and instead compares the saturation edge by edge. During the formalization, we have discovered and fixed a number of problems in the original proof [3].

Arguably, this statement still does not capture the intuition fully. For example, the infinite network in Fig. 14 has a cut of value 4 with an orthogonal flow. This is the cut that the proof of Thm. 1 constructs. Yet, this cut is not minimal: The cut that separates the upper nodes from the lower nodes would be saturated by a flow of 2 units (not shown).

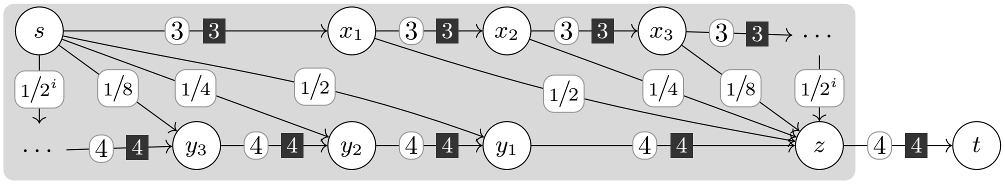


Figure 14 An infinite network with an orthogonal pair of a cut and a flow.

This illustrates the intricacies of infinite networks: The out-flow from the source s of value 3 drains away in the infinite ray $s \to x_1 \to x_2 \to x_3 \to \dots$. Conversely, the in-flow to the sink t of value 4 is pulled in via the infinite path $\dots \to y_3 \to y_2 \to y_1 \to z \to t$. So this network shows that the outflow from the source may exceed the capacity of a cut and yet not saturate it.

Aharoni et al. [3, Sects. 7–8] study two restrictions on networks that avoid such anomalies: networks without infinite edge-disjoint paths and locally-finite networks. We have not yet formalized these results. Neither result applies to the network in Fig. 14. So finding a more intuitive statement of the max-flow min-cut theorem for countable networks is still an open problem.

References

1 Ron Aharoni. Menger's theorem for graphs containing no infinite paths. *European Journal of Combinatorics*, 4:201–204, 1983. `doi:10.1016/S0195-6698(83)80012-2`.

2 Ron Aharoni and Eli Berger. Menger's theorem for infinite graphs. *Inventiones mathematicae*, 176(1):1–62, 2009. `doi:10.1007/s00222-008-0157-3`.

3 Ron Aharoni, Eli Berger, Agelos Georgakopoulos, Amitai Perlstein, and Philipp Sprüssel. The max-flow min-cut theorem for countable networks. *Journal of Combinatorial Theory, Series B*, 101:1–17, 2010. `doi:10.1016/j.jctb.2010.08.002`.

4 Clemens Ballarin. Locales and locale expressions in Isabelle/Isar. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs (TYPES 2003)*, volume 3085 of *LNCS*, pages 34–50. Springer Berlin Heidelberg, 2004. `doi:10.1007/978-3-540-24849-1_3`.

5 Clemens Ballarin. Locales: A module system for mathematical theories. *Journal of Automated Reasoning*, 52:123–153, 2014. `doi:10.1007/s10817-013-9284-7`.

6 Clemens Ballarin. Exploring the structure of an algebra text with locales. *Journal of Automated Reasoning*, 64:1093–1121, 2020. `doi:10.1007/s10817-019-09537-9`.

7 Gilles Barthe, Thomas Espitau, Justin Hsu, Tetsuya Sato, and Pierre-Yves Strub. *-liftings for differential privacy. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *International Colloquium on Automata, Languages, and Programming (ICALP 2017)*, volume 80 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 102:1–102:12, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.ICALP.2017.102`.

8 N. Bourbaki. Sur le théorème de Zorn. *Archiv der Mathematik*, 2(6):434–437, 1949.

9 Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972. `doi:10.1145/321694.321699`.

10 L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956. `doi:10.4153/CJM-1956-045-5`.

11 Fabian Immler. Generic construction of probability spaces for paths of stochastic processes in Isabelle/HOL. Master's thesis, Fakultät für Informatik, Technische Universität München, 2012.

12 Hans G. Kellerer. Funktionen auf Produkträumen mit vorgegebenen Marginal-Funktionen. *Mathematische Annalen*, 144:323–344, 1961. doi:10.1007/BF01470505.

13 Peter Lammich and S. Reza Sefidgar. Formalizing the Edmonds-Karp algorithm. In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving (ITP 2016)*, volume 9807 of *LNCS*, pages 219–234. Springer, 2016. doi:10.1007/978-3-319-43144-4_14.

14 Peter Lammich and S. Reza Sefidgar. Formalizing network flow algorithms: A refinement approach in Isabelle/HOL. *Journal of Automated Reasoning*, 62:261–280, 2019. doi:10.1007/s10817-017-9442-4.

15 Gilbert Lee. Correctnesss of Ford-Fulkerson's maximum flow algorithm. *Formalized Mathematics*, 13(2):305–314, 2005. URL: https://fm.mizar.org/2005-13/pdf13-2/glib_005.pdf.

16 Andreas Lochbihler. A formal proof of the max-flow min-cut theorem for countable networks. *Archive of Formal Proofs*, 2016. http://www.isa-afp.org/entries/MFMC_Countable.shtml, Formal proof development.

17 Andreas Lochbihler. Probabilistic functions and cryptographic oracles in higher-order logic. In Peter Thiemann, editor, *Programming Languages and Systems (ESOP 2016)*, volume 9632 of *LNCS*, pages 503–531. Springer, 2016. doi:10.1007/978-3-662-49498-1_20.

18 Andreas Lochbihler. A mechanized proof of the max-flow min-cut theorem for countable networks. http://www.andreas-lochbihler.de/pub/lochbihler2021itpl.pdf, 2021.

19 Russell Lyons and Yuval Peres. *Probability on Trees and Networks.* Cambridge University Press, New York, 2017. doi:10.1017/9781316672815.

20 Wolfgang Naraschewski and Markus Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher Order Logics (TPHOLs 1998)*, volume 1479 of *LNCS*, pages 349–366. Springer, 1998. doi:10.1007/BFb0055146.

21 Christophe Sabot and Laurent Tournier. Random walks in Dirichlet environment: an overview. *Annales de la Faculté des sciences de Toulouse: Mathématiques*, Ser. 6, 26(2):463–509, 2017. doi:10.5802/afst.1542.

22 Joshua Sack and Lijun Zhang. A general framework for probabilistic characterizing formulae. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI 2012)*, volume 7148 of *LNCS*, pages 396–411. Springer, 2012. doi:10.1007/978-3-642-27940-9_26.

23 Gert Smolka, Steven Schäfer, and Christian Doczkal. Transfinite constructions in classical type theory. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving (ITP 2015)*, volume 9236 of *LNCS*, pages 391–404. Springer, 2015. doi:10.1007/978-3-319-22102-1_26.

24 Freek Wiedijk. The de Bruijn factor. https://www.cs.ru.nl/~freek/factor/factor.pdf, 2000.

A Formal Proof of Modal Completeness for Provability Logic

Marco Maggesi
University of Florence, Italy

Cosimo Perini Brogi
University of Genoa, Italy

Abstract

This work presents a formalized proof of modal completeness for Gödel-Löb provability logic (GL) in the HOL Light theorem prover. We describe the code we developed, and discuss some details of our implementation. In particular, we show how we adapted the proof in the Boolos' monograph according to the formal language and tools at hand. The strategy we develop here overcomes the technical difficulty due to the non-compactness of GL, and simplify the implementation. Moreover, it can be applied to other normal modal systems with minimal changes.

2012 ACM Subject Classification Theory of computation → Higher order logic; Theory of computation → Modal and temporal logics; Theory of computation → Automated reasoning

Keywords and phrases Provability Logic, Higher-Order Logic, Mechanized Mathematics, HOL Light Theorem Prover

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.26

Related Version *Previous Version*: `https://arxiv.org/abs/2102.05945`

Supplementary Material The implementation described in this paper is hosted in the subdirectory `GL` of the distribution of HOL Light theorem prover.
Software (HOL Light source code, January 2021): `https://github.com/jrh13/hol-light/`
archived at `swh:1:dir:288139a46b5250ca678d928a823173f4f69c8594`

Funding *Marco Maggesi*: Supported by the Italian Ministry of University and Research and by INdAM-GNSAGA.

1 Introduction

In this paper we wish to report on our results and general experience in using HOL Light theorem prover to formally verify some properties of Gödel-Löb provability logic (GL).

Our work starts with a deep embedding of the syntax of propositional modal logic together with the corresponding relational semantics. Next, we introduce the traditional axiomatic calculus $\mathbb{GL}$ and prove the validity of the system w.r.t. irreflexive transitive finite frames.

After doing that, the most interesting part of our work begins with the proof of a number of lemmas *in* $\mathbb{GL}$ that are necessary for our main goal, namely the development of a formal proof of modal completeness for the system.

In order to achieve that, we had to formally verify a series of preliminary lemmas and constructions involving the behaviour of syntactical objects used in the standard proof of the completeness theorem. These unavoidable steps are very often only proof-sketched in wide-adopted textbooks in logic, for they mainly involve "standard" reasoning *within* the proof system we are dealing with. But when we are working in a formal setting like we did with HOL Light, experience reveals that it is generally more convenient to adopt a different line of reasoning and to make a smart use of our formal tools, so that we can succeed in developing alternative (or simpler) proofs, still totally verified by the computer.

In other terms: in order to give a formal proof of the completeness theorem for $\mathbb{GL}$, that is

▶ **Theorem 1.** *For any formula A, $\mathbb{GL} \vdash A$ iff A is true in any irreflexive transitive finite frame.*

we needed to split down the main goal into several subgoals – dealing with both the object- and the meta-level – which might seem trivial in informal reasoning. However, in order to carefully check them it has been more convenient to apply different proof-strategies where HOL Light infrastructure played a more active role.

We can briefly summarise the present paper as follows:

- In Section 2, we introduce the basic ingredients of our development, namely the formal counterparts of the syntax and relational semantics for provability logic, along with some lemmas and general definitions which are useful to handle the implementation of these objects in a uniform way, i.e. without the restriction to the specific modal system we are interested in. The formalization constitutes large part of the file `modal.ml`;
- In Section 3, we formally define the axiomatic calculus $\mathbb{GL}$, and prove in a neat way the validity lemma for this system. Moreover, we give formal proofs of several lemmas *in* $\mathbb{GL}$ ($\mathbb{GL}$-lemmas, for short), whose majority is in fact common to all normal modal logics, so that our proofs might be re-used in subsequent implementations of different systems. This corresponds to contents of our code in `gl.ml`;
- Finally, in Section 4 we give our formal proof of modal completeness of $\mathbb{GL}$, starting with the definition of maximal consistent *lists* of formulae. In order to prove their syntactic properties – and, in particular, the extension lemma for consistent lists of formulae to maximal consistent lists – we use the $\mathbb{GL}$-lemmas and, at the same time, we adapt an already known general proof-strategy to maximise the gain from the formal tools provided by HOL Light – or, informally, from higher-order reasoning. Therefore, the proof we are proposing in that section follows the standard lines presented in e.g. [7] – from extension lemma to modal completeness via truth lemma – but it adopts tools, results, and techniques which are specific to the theorem prover we used, in line with a clear philosophical attitude in computer-aided proof development.
  At the end of the Section, we give the formal definition of bisimilarity for our setup and we prove the associated bisimulation theorem [19, Ch. 11]. Our notion of bisimilarity is polymorphic, in the sense that it can relate classes of frames sitting on different types. With this tool at hand, we can correctly state our completeness theorem in its natural generality (`COMPLETENESS_THEOREM_GEN`) – i.e. for irreflexive, transitive finite frames over any (infinite) type. These results, together with a simple decision procedure for $\mathbb{GL}$, are gathered in the file `completeness.ml`.

Our code is integrated into the current HOL Light distribution, and it is freely available from there.[1] We care to stress that our formalization does not tweak any original HOL Light tools, and it is therefore "foundationally safe". Moreover, since we only used that original formal infrastructure, our results can be easily translated into another theorem prover belonging to the HOL family – or, more generally, endowed with the same automation toolbox.

Before presenting our results, in the forthcoming subsections of this introduction we provide the reader with some background material both on provability logic, and on formal theorem proving in HOL Light: further information about modalities and HOL Light can be found in [12] and [15], respectively.

[1] See "Supplementary Material" on the first page of this paper.

1.1 Developments of Provability Logic

The origin of provability logic dates back to a short paper by Gödel [11] where propositions about provability are formalized by means of a unary operator $\mathtt{B}$ with the aim of giving a classical reading of intuitionistic logic.

The resulting system corresponds to the logic $\mathbb{S}4$, and the proposition $\mathtt{B}p$ is interpreted as "*p* is *informally* provable" – as claimed by Gödel himself. This implies that $\mathbb{S}4$ can be considered a provability logic lacking an appropriate semantics.

At the same time, that work opened the question of finding an adequate modal calculus for the formal properties of the provability predicate used in Gödel's incompleteness theorems. That problem has been settled since 1970s for many formal systems of arithmetic by means of Gödel-Löb logic GL.

The corresponding axiomatic calculus $\mathbb{GL}$ consists of the axiomatic system for classical propositional logic, extended by the distributivity axiom schema K, the necessitation rule NR, and the axiom schema GL (see Section 3).

The schema GL is precisely a formal version of Löb's theorem, which holds for a wide class of arithmetical theories satisfying the so-called Hilbert-Bernays-Löb (HBL) provability conditions.[2]

The semantic counterpart of this calculus is - through the Kripke formalism - the logic of irreflexive transitive finite frames. Moreover, the calculus can be interpreted arithmetically in a sound and complete way. In other terms, $\mathbb{GL}$ solves the problem raised in Gödel's paper by identifying a propositional formal system for provability in all arithmetical theories that satisfies the previously mentioned HBL conditions.

Published in 1976, Solovay's arithmetical completeness theorem [21] is in this sense a milestone result in the fields of proof theory and modal logic. As $\mathbb{GL}$ is arithmetically complete, it is capable of capturing and identifying *all relevant properties of formal provability for arithmetic* in a very simple system, which is decidable and neatly characterised.

Such a deep result, however, uses in an essential way the *modal* completeness of $\mathbb{GL}$: Solovay's technique basically consists of an arithmetization of a relational countermodel for a given formula that is not a theorem of $\mathbb{GL}$, from which it is possible to define an appropriate arithmetical formula that is not a theorem of the mathematical system.

In contemporary research, this is still the main strategy to prove arithmetical completeness for other modalities for provability and related concepts, in particular for interpretability logic. In spite of this, for many theories of arithmetic – including Heyting Arithmetic – this technique cannot be applied, and no alternatives are known.

Therefore, on the one hand, completeness of formal systems w.r.t. the relevant relational semantics is still an unavoidable step in achieving the more substantial result of arithmetical completeness; on the other hand, however, the area of provability logic keeps flourishing and suggesting old and new open problems, closely related to the field of proof theory, but in fact bounded also to seeking a uniform proof-strategy to establish adequate semantics in formal theories of arithmetic having different strengths and flavours.[3]

[2] These properties of the formal predicate for arithmetical provability were isolated first in [16].
[3] The reader is referred to [3] for a survey of open problems in provability logics. As an instance of relevant applications of this kind of formal systems to traditional investigations in proof theory see e.g. [1].

1.2 HOL Light Notation

The HOL Light proof assistant [14] is based on *classical* higher-order logic with polymorphic type variables and where equality is the only primitive notion. From a logical viewpoint, the formal engine defined by the *term-conversions* and *inference rules* underlying HOL Light is the same as that described in [17], extended by an infinity axiom and the classical characterization of Hilbert's choice operator. From a practical perspective, it is a theorem prover privileging a procedural proof style development – i.e. when using it, we have to solve goals by applying *tactics* that reduce them to (eventually) simpler subgoals, so that the interactive aspect of proving is highlighted. Proof-terms can then be constructed by means of *tacticals* that compact the proof into few lines of code evaluated by the machine.

Logical operators – defined in terms of equality – and λ-abstraction are denoted by specific symbols in ASCII: for the reader's sake, we give a partial glossary in the next table. In the third column of the table, we also report the notation used for the object logic GL (introduced at the beginning of Section 2.1).

Informal notation	HOL notation	GL notation	Description
$\bot$	`F`	`False`	Falsity
$\top$	`T`	`True`	Truth
$\neg p$	`~ p`	`Not p`	Negation
$p \wedge q$	`/\`	`&&`	Conjunction
$p \vee q$	`\/`	`||`	Disjunction
$p \Longrightarrow q$	`==>`	`-->`	Implication
$p \Longleftrightarrow q$	`<=>`	`<->`	Biconditional
$\Box p$		`Box p`	Modal Operator
$p_1, \dots p_N \vdash p$	`p1 ... pN |- p`		HOL theorem
$\vdash p$		`|-- p`	Derivability in $\mathbb{GL}$
$\forall x.\, P(x)$	`!x. P(x)`		Universal quantification
$\exists x.\, P(x)$	`?x. P(x)`		Existential quantification
$\lambda x.\, M(x)$	`\x. M(x)`		Lambda abstraction
$x \in s$	`x IN s`		Set membership

We recall that a Boolean function `s :` α `-> bool` is also called a *set on* α in the HOL parlance. The notation `x IN s` is equivalent to `s x` and must not be confused with a type annotation `x :` α.

In the following sections, we will directly state our results as theorems and definitions in the HOL Light syntax. Note that theorems are prefixed by the turnstile symbol, as in `|- 2 + 2 = 4`. We often report a theorem with its associated name, that is, the name of its associated OCaml constant, e.g.

```
ADD_SYM
  |- !m n. m + n = n + m
```

As expository style, we omit formal proofs at all, but the meaning of definitions, lemmas, and theorems in natural language is hopefully clear after the table we have just given.

We warn the reader that the HOL Light printing mechanism omits type information completely. However in this paper we manually add type annotations when they might be useful, or even indispensable, in order to avoid ambiguity – including the case of our main results, `COMPLETENESS_THEOREM` and `COMPLETENESS_THEOREM_GEN`.

As already told in the introduction, our contribution is now part of the HOL Light distribution. The reader interested in performing these results on her machine – and perhaps build further formalization on top of it – can run our code with the command

```
loadt "GL/make.ml";;
```

at the HOL Light prompt.

2 Basics of Modal Logic

As we stated previously, we deal with a logic that extends classical propositional reasoning by means of a single modal operator which is intended to capture the abstract properties of the provability predicate for arithmetic.

To reason about and within this logic, we have to "teach" HOL Light – our meta-language – how to identify it, starting with its syntax – the object-language – and semantics – the interpretation of this very object-language.

We want to keep everything neat and clean from a foundational perspective, therefore we will define *both* the object-language and its interpretation with no relation to the HOL Light environment. In other terms: our formulae and operators are *real* syntactic objects which we keep distinct from their semantic counterpart – and from the logical operators of the theorem prover too.

2.1 Language and Semantics Defined

Let us start by fixing the propositional modal language we will use throughout the present work. We consider *all* classical propositional operators – conjunction, disjunction, implication, equivalence, negation, along with the 0-ary symbols $\top$ and $\bot$ – and we add a modal unary connective $\Box$. The starting point is, as usual, a denumerable infinite set of propositional atoms $a_0, a_1, \cdots$. Accordingly, formulae of this language will have one of the following form

$$a \mid A \wedge B \mid A \vee B \mid A \to B \mid A \leftrightarrow B \mid \neg A \mid \top \mid \bot \mid \Box A \ .$$

The following code extends the HOL system with an the **inductive type of formulae** up to the **atoms** – which we identify with the denumerable type of strings – by using the above **connectives**:

```
let form_INDUCT,form_RECURSION = define_type
  "form = False
        | True
        | Atom string
        | Not form
        | && form form
        | || form form
        | --> form form
        | <-> form form
        | Box form";;
```

Next, we turn to the semantics for our modal language. We use **relational models** – aka Kripke models[4]. Formally, a **Kripke frame** is made of a non-empty set "of possible worlds" `W`, together with a binary relation `R` on `W`. To this, we add a valuation function `V` which assigns to each atom of our language and each world `w` in `W` a Boolean value. This is extended to a **forcing relation** `holds`, defined recursively on the structure of the input formula `p`, that computes the truth-value of `p` in a specific world `w`:

```
let holds = new_recursive_definition form_RECURSION
  `(holds f V False (w:W) <=> F) /\
   (holds f V True w <=> T) /\
   (holds f V (Atom s) w <=> V s w) /\
```

[4] See [9] for the historical development of this notion.

```
    (holds f V (Not p) w <=> ~(holds f V p w)) /\
    (holds f V (p && q) w <=> holds f V p w /\ holds f V q w) /\
    (holds f V (p || q) w <=> holds f V p w \/ holds f V q w) /\
    (holds f V (p --> q) w <=> holds f V p w ==> holds f V q w) /\
    (holds f V (p <-> q) w <=> holds f V p w <=> holds f V q w) /\
    (holds f V (Box p) w <=> !u. u IN FST f /\ SND f w u ==> holds f V p u)`;;
```

In the previous lines of code, `f` stands for a generic Kripke frame – i.e., a pair `(W,R)` of a set of worlds and an accessibility relation – and `V` is an evaluation of propositional variables. Then, the **validity** of a formula `p` with respect to a frame `(W,R)`, and a class of frames `L`, denoted respectively `holds_in (W,R) p` and `L |= p`, are

```
let holds_in = new_definition
  `holds_in (W,R) p <=> !V w. w IN W ==> holds (W,R) V p w`;;

let valid = new_definition
  `L |= p <=> !f. L f ==> holds_in f p`;;
```

The above formalization is essentially the one presented in Harrison's HOL Light Tutorial [15, § 20]. Notice that the usual notion of Kripke frame requires that the set of possible worlds is non-empty: that condition could be imposed by adapting the `valid` relation. We have preferred to stick to Harrison's original definitions in our code, but in the next section, when we define the classes of frames we are dealing with, the requirement on `W` is correctly integrated in the corresponding types.

2.2 Frames for GL

For carrying out our formalization, we are interested in the logic of the (non-empty) frames whose underlying relation R is **transitive** and conversely well-founded – aka **Noetherian** – on the corresponding set of possible worlds; in other terms, we want to study the modal tautologies in models based on an accessibility relation R on W such that

- if xRy and yRz, then xRz; and
- for no $X \subseteq W$ there are infinite R-chains $x_0Rx_1Rx_2\cdots$.

In HOL Light, `WF R` states that `R` is a well-founded relation, so that we express the latter condition as `WF(\x y. R y x)`. Here we see a recurrent motif in logic: defining a system from the semantic perspective requires non-trivial tools from the foundational point of view, for, to express the second condition, a first-order language is not enough. However, that is not an issue here, since our underlying system is natively higher order:

```
let TRANSNT = new_definition
  `TRANSNT (W:W->bool,R:W->W->bool) <=>
   ~(W = {}) /\
   (!x y:W. R x y ==> x IN W /\ y IN W) /\
   (!x y z:W. x IN W /\ y IN W /\ z IN W /\ R x y /\ R y z ==> R x z) /\
   WF(\x y. R y x)`;;
```

We warn the reader that in the previous statement there occur two interrelated mathematical objects both denoted `W` (for convenience): one is the type `W` and the other is the set `W` on the former (in the sense explained in the introduction about the HOL syntax). From a theoretical point of view, moreover, the question has no deep consequences as we can characterize this class of frames by using a *propositional* language extended by a modal operator $\Box$ that satisfies the *Gödel-Löb axiom schema* (GL) : $\Box(\Box A \to A) \to \Box A$. Here is the formal version of our claim:

```
TRANSNT_EQ_LOB
  |- !W:W->bool R:W->W->bool.
       (!x y:W. R x y ==> x IN W /\ y IN W)
       ==> ((!x y z. x IN W /\ y IN W /\ z IN W /\ R x y /\ R y z ==> R x z) /\
            WF (\x y. R y x) <=>
            (!p. holds_in (W,R) (Box(Box p --> p) --> Box p)))
```

The informal proof of the above result is standard and can be found in [7, Theorem 10] and in [19, Theorem 5.7]. The computer implementation of the proof is made easy thanks to Harrison's tactic `MODAL_SCHEMA_TAC` for semantic reasoning in modal logic, documented in [15, § 20.3].

By using this preliminary result, we could say that the frame property of being transitive and Noetherian can be captured by Gödel-Löb modal axiom, without recurring to a higher-order language. Nevertheless, that class of frames is not particularly informative from a logical point of view: a frame in `TRANSNT` can be too huge to be used in practice – for instance, for checking whether a formula is indeed a theorem of our logic. In particular, when aiming at a completeness theorem, one wants to consider models that are useful for further investigations on the properties of the very logic under consideration – in the present case, decidability of GL, which, as for any other normal modal logic, is an easy corollary of the *finite model property* [19, Ch. 13].

To this aim we note that by definition of Noetherianness, our R cannot be reflexive – otherwise $xRxRx\cdots$ would give us an infinite R-chain. This is not enough: following our main reference [7], the frames we want to investigate are precisely those whose W is **finite**, and whose R is both **irreflexive** and **transitive**:

```
let ITF = new_definition
  `ITF (W:W->bool,R:W->W->bool) <=>
   ~(W = {}) /\
   (!x y:W. R x y ==> x IN W /\ y IN W) /\
   FINITE W /\
   (!x. x IN W ==> ~R x x) /\
   (!x y z. x IN W /\ y IN W /\ z IN W /\ R x y /\ R y z ==> R x z)`;;
```

Now it is easy to see that `ITF` is a subclass of `TRANSNT`:

```
ITF_NT
  |- !W R:W->W->bool. ITF(W,R) ==> TRANSNT(W,R)
```

That will be the class of frames whose logic we are now going to define syntactically.

3 Axiomatizing GL

We want to identify the logical system generating all the modal tautologies for transitive Noetherian frames; more precisely, we want to isolate the *generators* of the modal tautologies in the subclass of transitive Noetherian frames which are finite, transitive, and irreflexive. Notice that the lemma `ITF_NT` allows us to derive the former result as a corollary of the latter.

When dealing with the very notion of tautology – or *theoremhood*, discarding the complexity or structural aspects of *derivability* in a formal system – it is convenient to focus on axiomatic calculi. The calculus we are dealing with here is usually denoted by $\mathbb{GL}$.

It is clear from the definition of the forcing relation that for classical operators any axiomatization of propositional classical logic will do the job. Here, we adopt a basic system in which only $\to$ and $\bot$ are primitive – from the axiomatic perspective – and all the remaining classical connectives are defined by axiom schemas and by the inference rule of Modus Ponens imposing their standard behaviour.[5]

As anticipated in the Introduction, to this classical engine we add

- the axiom schema K: $\Box(A \to B) \to \Box A \to \Box B$;
- the axiom schema GL: $\Box(\Box A \to A) \to \Box A$;
- the necessitation rule NR: $\dfrac{A}{\Box A}\,\textsf{NR}$,

where A, B are generic formulae (not simply atoms). Then, here is the complete definition of the **axiom system** $\mathbb{GL}$. The set of axioms is encoded via the inductive predicate `GLaxiom`:

```
let GLaxiom_RULES,GLaxiom_INDUCT,GLaxiom_CASES = new_inductive_definition
  `(!p q. GLaxiom (p --> (q --> p))) /\
   (!p q r. GLaxiom ((p --> q --> r) --> (p --> q) --> (p --> r))) /\
   (!p. GLaxiom (((p --> False) --> False) --> p)) /\
   (!p q. GLaxiom ((p <-> q) --> p --> q)) /\
   (!p q. GLaxiom ((p <-> q) --> q --> p)) /\
   (!p q. GLaxiom ((p --> q) --> (q --> p) --> (p <-> q))) /\
   GLaxiom (True <-> False --> False) /\
   (!p. GLaxiom (Not p <-> p --> False)) /\
   (!p q. GLaxiom (p && q <-> (p --> q --> False) --> False)) /\
   (!p q. GLaxiom (p || q <-> Not(Not p && Not q))) /\
   (!p q. GLaxiom (Box (p --> q) --> Box p --> Box q)) /\
   (!p. GLaxiom (Box (Box p --> p) --> Box p))`;;
```

The judgment $\mathbb{GL} \vdash A$, denoted `|-- A` in the machine code (not to be confused with the symbol for HOL theorems `|-`), is also inductively defined in the expected way:

```
let GLproves_RULES,GLproves_INDUCT,GLproves_CASES = new_inductive_definition
  `(!p. GLaxiom p ==> |-- p) /\
   (!p q. |-- (p --> q) /\ |-- p ==> |-- q) /\
   (!p. |-- p ==> |-- (Box p))`;;
```

3.1 $\mathbb{GL}$-lemmas

As usual, $\mathbb{GL} \vdash A$ denotes the existence of a derivation of A from the axioms of $\mathbb{GL}$; we could also define a notion of derivability from a set of assumptions just by tweaking the previous definitions in order to handle the specific limitations on NR – so that the deduction theorem would hold [13] – but this would be inessential to our intents.

Proving some lemmas in the axiomatic calculus $\mathbb{GL}$ is a technical interlude necessary for obtaining the completeness result.

In accordance with this aim, we denoted the classical axioms and rules of the system as the propositional schemas used by Harrison in the file `Arithmetic/derived.ml` of the HOL Light standard distribution [14] – where, in fact, many of our lemmas relying only on the propositional calculus are already proven there w.r.t. an axiomatic system for first-order classical logic; our further lemmas involving modal reasoning are denoted by names that are commonly used in informal presentations.

Therefore, the code in `gl.ml` mainly consists of the formalized proofs of those lemmas *in* $\mathbb{GL}$ that are useful for the formalized results we present in the next section. This file might be thought of as a “kernel” for further experiments in reasoning about axiomatic calculi by using HOL Light. The lemmas we proved are, indeed, standard tautologies of classical

[5] This is essentially the calculus introduced by Church in [8]

propositional logic, along with specific theorems of minimal modal logic and its extension for transitive frames – i.e. of the systems $\mathbb{K}$ and $\mathbb{K}4$ [19] –, so that by applying minor changes in basic definitions, they are – so to speak – take-away proof-terms for extensions of that very minimal system within the realm of normal modal logics.

More precisely, we have given, whenever it was useful, a "sequent-style natural deduction" characterization of classical operators both in terms of an implicit (or internal) deduction – and in that case we named the lemma with the suffix `_th` –, such as

```
GL_modusponens_th
  |- !p q. |-- ((p --> q) && p --> q)
```

and as a derived rule of the axiomatic system mimicking the behaviour of the connective in Gentzen's formalism, e.g.,

```
GL_and_elim
  |- !p q r. |-- (r --> p && q) ==> |-- (r --> q)  /\ |-- (r --> p)
```

We had to prove about 120 such results of varying degree of difficulty. We believe that this file is well worth the effort of its development, for two main reasons to be considered – along with the just mentioned fact that they provide a (not so) minimal set of internal lemmas which can be moved to different axiomatic calculi at, basically, no cost.

Indeed, on the one hand, these lemmas simplify the subsequent formal proofs involving consistent lists of formulae since they let us work formally within the scope of $\vdash$, so that we can rearrange subgoals according to their most useful equivalent form by applying the appropriate $\mathbb{GL}$-lemma(s).

On the other hand, the endeavour of giving formal proofs of these lemmas of the calculus $\mathbb{GL}$ has been important for checking how much our proof-assistant is "friendly" and efficient in performing this specific task.

As it is known, any axiomatic system fits very well an investigation involving the notion of *theoremhood* for a specific logic, but its lack of naturalness w.r.t. the practice of developing informal proofs makes it an unsatisfactory model for the notion of *deducibility*. In more practical terms: developing a formal proof of a theorem in an axiomatic system *by pencil and paper* can be a dull and uninformative task.

We therefore left the proof-search to the HOL Light toolbox as much as possible. Unfortunately, we have to express mixed feelings on the general experience. In most cases, relying on the automation tools of this specific proof assistant did indeed save our time and resources when trying to give a formal proof in $\mathbb{GL}$. Nevertheless, there has been a number of $\mathbb{GL}$-lemmas for proving which those automation tools did not revealed useful at all. In those cases, actually, we had to perform a tentative search of the specific instances of axioms from which deriving the lemmas,[6] so that interactive proving them had advantages as well as traditional instruments of everyday mathematicians.

Just to stress the general point: it is clearly possible – and actually useful in general – to rely on the resources of HOL Light to develop formal proofs both *about* and *within* an axiomatic calculus for a specific logic, in particular when the lemmas of the object system have relevance or practical utility for mechanizing (meta-)results on it; however, these very resources – and, as far as we can see, the tools of any other general proof assistant – do not look peculiarly satisfactory for pursuing investigations on derivability within axiomatic systems.

[6] The HOL Light tactics for first-order reasoning `MESON` and `METIS` were unable, for example, to instantiate autonomously the obvious middle formula for the transitivity of an implication, or even the specific formulae of a schema to apply to the goal in order to rewrite it.

3.2 Soundness Lemma

At this point, we can prove that 𝔾𝕃 is **sound** – i.e. every formula derivable in the calculus is a tautology in the class of irreflexive transitive finite frames. This is obtained by simply unfolding the relevant definitions and applying theorems `TRANSNT_EQ_LOB` and `ITF_NT` of Section 2.2:

```
GL_TRANSNT_VALID
  |- !p. (|-- p) ==> TRANSNT:(W->bool)#(W->W->bool)->bool |= p

GL_ITF_VALID
  |- !p. |-- p ==> ITF:(W->bool)#(W->W->bool)->bool |= p
```

From this, we get a model-theoretic proof of **consistency for the calculus**

```
GL_consistent
  ~ |-- False
```

Having exhausted the contents of file `gl.ml`, we shall move to consider the most interesting part of our effort, namely the mechanized proof of completeness for the calculus w.r.t. this very same class of frames. That constitutes the remaining contents of our implementation, beside the auxiliary code in `misc.ml` furnishing some general results about lists of items with same type we needed to handle the subsequent constructions (but have a more general utility).

4 Completeness and Decidability

When dealing with normal modal logics, it is common to develop a proof of completeness w.r.t. relational semantics by using the so-called "canonical model method". This can be summarized as a standard construction of countermodels made of maximal consistent sets of formulae and an appropriate accessibility relation [19].

For 𝔾𝕃, we cannot pursue this strategy, since the logic is not compact: maximal consistent sets are (in general) infinite objects, though the notion of derivability involves only a finite set of formulae. We cannot therefore reduce the semantic notion of (in)coherent set of formulae to the syntactic one of (in)consistent set of formulae: when extending a consistent set of formulae to a maximal consistent one, we might end up with a *syntactically* consistent set that nevertheless cannot be *semantically* satisfied.

In spite of this, it is possible to achieve a completeness result by

1. identifying the relevant properties of maximal consistent sets of formulae; and
2. tweaking the definitions so that those properties hold for specific consistent sets of formulae related to the formula we want to find a countermodel to.

That is, basically, the key idea behind the proof given in [7, Ch. 5]. In that monograph, however, the construction of a maximal consistent set from a simply consistent one is only proof-sketched and relies on a syntactic manipulation of formulae. By using HOL Light we do succeed in giving a detailed proof of completeness as direct as that by Boolos. But, as a matter of fact, we can claim something more: we can do that by carrying out in a *very natural way* a tweaked Lindenbaum construction to extend consistent *lists* to maximal consistent ones. This way, we succeed in preserving the standard Henkin-style completeness proofs; and, at the same time, we avoid the symbolic subtleties sketched in [7] that have no real relevance for the argument, but have the unpleasant consequence of making the formalized proof unnecessarily long, so that the implementation would sound rather pedantic – or even dull.

Furthermore, the proof of the main lemma `EXTEND_MAXIMAL_CONSISTENT` is rather general and does not rely on any specific property of $\mathbb{GL}$: our strategy suits all the others normal (mono)modal logics – we only need to modify the subsequent definition of `STANDARD_RELATION` according to the specific system under consideration. Thus, we provide a way for establishing completeness à la Henkin *and* the finite model property without recurring to filtrations [19] of canonical models for those systems.

4.1 Maximal Consistent Lists

Following the standard practice, we need to consider consistent finite sets of formulae for our proof of completeness. In principle, we can employ general sets of formulae in the formalization, but, from the practical view-point, lists without repetitions are better suited, since they are automatically finite and we can easily manipulate them by structural recursion. We define first the operation of finite conjunction of formulae in a list:[7]

```
let CONJLIST = new_recursive_definition list_RECURSION
  `CONJLIST [] = True /\
   (!p X. CONJLIST (CONS p X) = if X = [] then p else p && CONJLIST X)`;;
```

We proceed by proving some properties on lists of formulae and some $\mathbb{GL}$-lemmas involving `CONJLIST`. In particular, since $\mathbb{GL}$ is a normal modal logic – i.e. its modal operator distributes over implication and preserves theoremhood – we have that our $\Box$ distributes over the conjunction of X so that we have `CONJLIST_MAP_BOX`:

$$\mathbb{GL} \vdash \Box \bigwedge X \leftrightarrow \bigwedge \Box X,$$

where $\Box X$ is an abuse of notation for the list obtained by "boxing" each formula in X.

We are now able to define the notion of **consistent list of formulae** and prove the main properties of this kind of objects:

```
let CONSISTENT = new_definition
  `CONSISTENT (l:form list) <=> ~ (|-- (Not (CONJLIST l)))`;;
```

In particular, we prove that:

- a consistent list cannot contain both A and $\neg A$ for any formula A, nor $\bot$ (see theorems `CONSISTENT_LEMMA`, `CONSISTENT_NC`, and `FALSE_IMP_NOT_CONSISTENT`, respectively);
- for any consistent list X and formula A, either $X + A$ is consistent, or $X + \neg A$ is consistent (`CONSISTENT_EM`), where $+$ denotes the usual operation of appending an element to a list.

Our **maximal consistent lists** w.r.t. a given formula A will be consistent lists that do not contain repetitions and that contain, for any subformula of A, that very subformula or its negation:[8]

```
let MAXIMAL_CONSISTENT = new_definition
  `MAXIMAL_CONSISTENT p X <=>
   CONSISTENT X /\ NOREPETITION X /\
   (!q. q SUBFORMULA p ==> MEM q X \/ MEM (Not q) X)`;;
```

[7] Notice that in the this definition we perform a case analysis where the singleton list is treated separately (i.e., we have `CONJLIST [p] = p`). This is slightly uncomfortable in certain formal proof steps: in retrospect, we might have used a simpler version of this function. However, since this is a minor detail, we preferred not to change our code.

[8] Here we define the set of subformulae of A as the reflexive transitive closure of the set of formulae on which the main connective of A operates: this way, the definition is simplified and it is easier to establish standard properties of the set of subformulae by means of general higher-order lemmas in HOL Light for the closure of a given relation.

where `X` is a list of formulae and `MEM q X` is the membership relation for lists. We then establish the main closure property of maximal consistent lists:

```
MAXIMAL_CONSISTENT_LEMMA
  |- !p X A b. MAXIMAL_CONSISTENT p X /\
               (!q. MEM q A ==> MEM q X) /\
               b SUBFORMULA p /\
               |-- (CONJLIST A --> b)
               ==> MEM b X
```

After proving some further lemmas with practical utility – in particular, the fact that any maximal consistent list behaves like a restricted bivalent evaluation for classical connectives (`MAXIMAL_CONSISTENT_MEM_NOT` and `MAXIMAL_CONSISTENT_MEM_CASES`) – we can finally define the ideal (type of counter)model we are interested in. The type `STANDARD_MODEL` consists, for a given formula `p`, of:

1. the set of maximal consistent lists w.r.t. `p` made of *subsentences* of `p` – i.e. its subformulae or their negations – as possible worlds;
2. an irreflexive transitive accessibility relation `R` such that for any subformula `Box q` of `p` and any world `w`, `Box q` is in `w` iff, for any `x` `R`-accessible from `w`, `q` is in `x`;
3. an atomic valuation that gives value `T` (true) to `a` in `w` iff `a` is a subformula of `p`.

After defining the relation of *subsentences* as

```
let SUBSENTENCE = new_definition
  `!p q. q SUBSENTENCE p <=>
        q SUBFORMULA p \/ (?q'. q = Not q' /\ q' SUBFORMULA p)`;;
```

we can introduce the corresponding code:

```
let GL_STANDARD_FRAME = new_definition
  `GL_STANDARD_FRAME p (W,R) <=>
   W = {w | MAXIMAL_CONSISTENT p w /\ (!q. MEM q w ==> q SUBSENTENCE p)} /\
   ITF (W,R) /\
   (!q w. Box q SUBFORMULA p /\ w IN W
          ==> (MEM (Box q) w <=> !x. R w x ==> MEM q x))`;;

let GL_STANDARD_MODEL = new_definition
  `GL_STANDARD_MODEL p (W,R) V <=>
   GL_STANDARD_FRAME p (W,R) /\
   (!a w. w IN W ==> (V a w <=> MEM (Atom a) w /\ Atom a SUBFORMULA p))`;;
```

4.2 Maximal Extensions

What we have to do now is to show that the type `GL_STANDARD_MODEL` is non-empty. We achieve this by constructing suitable maximal consistent lists of formulae from specific consistent ones.

Our original strategy differs from the presentation given in e.g. [7] for being closer to the standard Lindenbaum construction commonly used in proving completeness results. By doing so, we have been able to circumvent both the pure technicalities in formalizing the combinatorial argument sketched in [7, p.79] *and* the problem – apparently inherent to the Lindenbaum extension – due to the non-compactness of the system, as we mentioned before.

The main lemma states then that, from any consistent list X of subsentences of a formula A, we can construct a maximal consistent list of subsentences of A by extending (if necessary) X:

```
EXTEND_MAXIMAL_CONSISTENT
  |- !p X.
       CONSISTENT X /\
       (!q. MEM q X ==> q SUBSENTENCE p)
       ==> ?M. MAXIMAL_CONSISTENT p M /\
               (!q. MEM q M ==> q SUBSENTENCE p) /\
               X SUBLIST M
```

The proof-sketch is as follows: given a formula A, we proceed in a step-by-step construction by iterating over the subformulae B of A not contained in X. At each step, we append to the list X the subformula B – if the resulting list is consistent – or its negation $\neg B$ – otherwise.

This way, we are in the pleasant condition of carrying out the construction by using the HOL Light device efficiently, and, at the same time, we do not have to worry about the non-compactness of $\mathbb{GL}$, since we are working with finite objects – the type `list` – from the very beginning.

Henceforth, we see that – under the assumption that A is not a $\mathbb{GL}$-lemma – the set of possible worlds in `STANDARD_FRAME` w.r.t. A is non-empty, as required by the definition of relational structures:

```
NONEMPTY_MAXIMAL_CONSISTENT
  |- !p. ~ |-- p
         ==> ?M. MAXIMAL_CONSISTENT p M /\
                 MEM (Not p) M /\
                 (!q. MEM q M ==> q SUBSENTENCE p)
```

Next, we have to define an R satisfying the condition 2 for a `STANDARD_FRAME`; the following does the job:

```
let GL_STANDARD_REL = new_definition
  `GL_STANDARD_REL p w x <=>
   MAXIMAL_CONSISTENT p w /\
   (!q. MEM q w ==> q SUBSENTENCE p) /\
   MAXIMAL_CONSISTENT p x /\
   (!q. MEM q x ==> q SUBSENTENCE p) /\
   (!B. MEM (Box B) w ==> MEM (Box B) x /\ MEM B x) /\
   (?E. MEM (Box E) x /\ MEM (Not (Box E)) w)`;;
```

Such an accessibility relation, together with the set of the specific maximal consistent lists we are dealing with, defines a structure in `ITF` with the required properties:

```
ITF_MAXIMAL_CONSISTENT
  |- !p. ~ |-- p
         ==> ITF ({M | MAXIMAL_CONSISTENT p M /\
                       (!q. MEM q M ==> q SUBSENTENCE p)},
                  GL_STANDARD_REL p),

ACCESSIBILITY_LEMMA
  |- !p M w q.
       ~ |-- p /\
       MAXIMAL_CONSISTENT p M /\
       (!q. MEM q M ==> q SUBSENTENCE p) /\
       MAXIMAL_CONSISTENT p w /\
       (!q. MEM q w ==> q SUBSENTENCE p) /\
       MEM (Not p) M /\
       Box q SUBFORMULA p /\
       (!x. GL_STANDARD_REL p w x ==> MEM q x)
       ==> MEM (Box q) w,
```

4.3 Truth Lemma and Completeness

For our ideal model, it remains to reduce the semantic relation of forcing to the more tractable one of membership to the specific world. More formally, we prove – by induction on the complexity of the subformula B of A – that if $\mathbb{GL} \nvdash A$, then for any world w of the standard model, B holds in w iff B is member of w:

```
GL_truth_lemma
  |- !W R p V q.
       ~ |-- p /\
       GL_STANDARD_MODEL p (W,R) V /\
       q SUBFORMULA p
       ==> !w. w IN W ==> (MEM q w <=> holds (W,R) V q w),
```

Finally, we are able to prove the main result: if $\mathbb{GL} \nvdash A$, then the list $[\neg A]$ is consistent, and by applying EXTEND_MAXIMAL_CONSISTENT, we obtain a maximal consistent list X w.r.t. A that extends it, so that, by applying GL_truth_lemma, we have that $X \not\models A$ in our standard model. The corresponding formal proof reduces to the application of those previous results and the appropriate instantiations:

```
COMPLETENESS_THEOREM
  |- !p. ITF:(form list->bool)#(form list->form list->bool)->bool |= p
         ==> |-- p,
```

Notice that the family of frames ITF is polymorphic, but, at this stage, our result holds only for frames on the domain form list, as indicated by the type annotation. This is not an intrinsic limitation: the next section is devoted indeed to generalize this theorem to frames on an arbitrary domain.

By using our EXTEND_MAXIMAL_CONSISTENT lemma, we succeeded in giving a rather quick proof of both completeness and the finite model property for $\mathbb{GL}$.[9]

Indeed, as an immediate corollary, we have that the system $\mathbb{GL}$ is decidable and, in principle, we could implement a decision procedure for it in OCaml. This is a not-so-easy task – especially if one seeks efficiency and completeness – and it is out of the scope of the present work. Nevertheless, we feel like offering a very rough approximation.

We define the tactic GL_TAC and its associated rule GL_RULE that perform the following steps: (1) apply the completeness theorem, (2) unfold some definitions, and (3) try to solve the resulting *semantic* problem using first-order reasoning.

```
let GL_TAC : tactic =
  MATCH_MP_TAC COMPLETENESS_THEOREM THEN
  REWRITE_TAC[valid; FORALL_PAIR_THM; holds_in; holds;
              ITF; GSYM MEMBER_NOT_EMPTY] THEN
  MESON_TAC[];;

let GL_RULE tm = prove(tm, REPEAT GEN_TAC THEN GL_TAC);;
```

The above naive strategy is able to prove automatically some lemmas which are common to normal modal logic, but require some effort when derived in an axiomatic system. As an example consider the following $\mathbb{GL}$-lemma:

[9] For the completeness w.r.t. transitive Noetherian frames, it is common – see [7, 19] – to reason on irreflexive transitive structures and derive the result as a corollary of completeness w.r.t. the latter class.

```
GL_box_iff_th
  |- !p q. |-- (Box (p <-> q) --> (Box p <-> Box q))
```

When developing a proof of it within the axiomatic calculus, we need to "help" HOL Light by instantiating several further 𝔾𝕃-lemmas, so that the resulting proof-term consists of ten lines of code. On the contrary, our rule is able to check it in few steps:

```
# GL_RULE `!p q. |-- (Box (p <-> q) --> (Box p <-> Box q))`;;
  0..0..1..6..11..19..32..solved at 39
  0..0..1..6..11..19..32..solved at 39
val it : thm = |- !p q. |-- (Box (p <-> q) --> (Box p <-> Box q))
```

In spite of this, the automation offered by `MESON` tactic is not enough when the generated semantic problem involves in an essential way the fact that our frames are finite (or Noetherian). So, for instance, our procedure is not able to prove the Gödel-Löb axiom

$$\mathbb{GL} \vdash \Box(\Box A \longrightarrow A) \longrightarrow \Box A\,.$$

This suggests the need for improving `GL_TAC` to handle more general contexts: in the long run, it is a likely-looking outcome of what we reached so far.

4.4 Generalizing via Bisimulation

As we stated before, our theorem `COMPLETENESS_THEOREM` provides the modal completeness for 𝔾𝕃 with respect to a semantics defined using models built on the type `:form list`. It is obvious that the same result must hold whenever we consider models built on any infinite type. To obtain a formal proof of this fact, we need to establish a *correspondence* between models built on different types. It is well-known that a good way to make rigorous such a correspondence is by means of the notion of *bisimulation* [5].

In our context, given two models `(W1,R1)` and `(W2,R2)` sitting respectively on types `:A` and `:B`, each with a valuation function `V1` and `V2`, a **bisimulation** is a binary relation `Z:A->B->bool` that relates two worlds `w1:A` and `w2:B` when they can *simulate* each other. The formal definition is as follows:

```
BISIMIMULATION
  |- BISIMIMULATION (W1,R1,V1) (W2,R2,V2) Z <=>
     (!w1:A w2:B.
        Z w1 w2
        ==> w1 IN W1 /\ w2 IN W2 /\
            (!a:string. V1 a w1 <=> V2 a w2) /\
            (!w1'. R1 w1 w1'  ==> ?w2'. w2' IN W2 /\ Z w1' w2' /\ R2 w2 w2') /\
            (!w2'. R2 w2 w2' ==> ?w1'. w1' IN W1 /\ Z w1' w2' /\ R1 w1 w1'))
```

Then, we say that two worlds are *bisimilar* if there exists a bisimulation between them:

```
let BISIMILAR = new_definition
  `BISIMILAR (W1,R1,V1) (W2,R2,V2) (w1:A) (w2:B) <=>
   ?Z. BISIMIMULATION (W1,R1,V1) (W2,R2,V2) Z /\ Z w1 w2`;;
```

The key fact is that the semantic predicate `holds` respects bisimilarity:

```
BISIMILAR_HOLDS
  |- !W1 R1 V1 W2 R2 V2 w1:A w2:B.
       BISIMILAR (W1,R1,V1) (W2,R2,V2) w1 w2
       ==> (!p. holds (W1,R1) V1 p w1 <=> holds (W2,R2) V2 p w2)
```

From this, we can prove that validity is preserved by bisimilarity. The precise statements are the following:

```
BISIMILAR_HOLDS_IN
  |- !W1 R1 W2 R2.
       (!V1 w1:A. ?V2 w2:B. BISIMILAR (W1,R1,V1) (W2,R2,V2) w1 w2)
       ==> (!p. holds_in (W2,R2) p ==> holds_in (W1,R1) p)

BISIMILAR_VALID
  |- !L1 L2.
       (!W1 R1 V1 w1:A.
          L1 (W1,R1) /\ w1 IN W1
          ==> ?W2 R2 V2 w2:B. L2 (W2,R2) /\
                              BISIMILAR (W1,R1,V1) (W2,R2,V2) w1 w2)
       ==> (!p. L2 |= p ==> L1 |= p)
```

In the last theorem, recall that the statement `L(W,R)` means that `(W R)` is a frame in the class of frames `L`.

Finally, we can explicitly define a bisimulation between `ITF`-models on the type `:form list` and on any infinite type `:A`. From this, it follows at once the desired generalization of completeness for $\mathbb{GL}$:

```
COMPLETENESS_THEOREM_GEN
  |- !p. INFINITE (:A) /\ ITF:(A->bool)#(A->A->bool)->bool |= p ==> |-- p
```

5 Related Work

Our formalization gives a mechanical proof of completeness for $\mathbb{GL}$ in HOL Light which sticks to the original Henkin's method for classical logic. In its standard version, its nature is synthetic and intrinsically semantic [10], and, as we stated before, it is the core of the canonical model construction for most of normal modal logic.

That very approach does not work for $\mathbb{GL}$. Nevertheless, the modified extension lemma we proved in our mechanization introduces an analytic flavour to the strategy – for building maximal consistent lists in terms of components of a given non-provable formula in the calculus – and shows that Henkin's idea can be applied to $\mathbb{GL}$ too modulo appropriate changes.

As far as we know, *no other mechanized proof of modal completeness for* $\mathbb{GL}$ has been given before, despite there exist formalizations of similar results for several other logics, manly propositional and first-order classical and intuitionistic logic.

Formalizations of completeness for *classical logic* define an established trend in interactive theorem proving since [20], where a Hintikka-style strategy is used to define a theoremhood checker for formulae built up by negation and disjunction only.

In fact, a very general treatment of systems for classical propositional logic is given in [18]. There, an axiomatic calculus is investigated along with natural deduction, sequent calculus, and resolution system in Isabelle/HOL, and completeness is proven by Hintikka-style method for sequent calculus first, to be lifted then to the other formalisms by means of translations of each system into the others. Their formalization is more ambitious than ours, but, at the same time, it is focused on a very different aim. A similar overview of meta-theoretical results for several calculi formalized in Isabelle/HOL is given in [6], where, again, a more general investigation – unrelated to modal logics – is provided.

Concerning the area of intuitionistic modalities, [2] gives a constructive proof of completeness of IS4 w.r.t. a specific relational semantics verified in Agda, but it uses natural deduction and apply modal completeness to obtain a normalization result for the terms of the associated λ-calculus.

A Henkin-style completeness proof for $\mathbb{S}5$ is formalised in Lean [4]. That work applies the standard method of canonical models – since $\mathbb{S}5$ is compact – but does not prove the finite model property for the logic.

More recently, Xu and Norrish in [22] have used the HOL4 theorem prover for a general treatment of model theory of modal systems. As a future work, it might be interesting to make use of the formalization therein along with the main lines of our implementation of axiomatic calculi to merge the two presentations – syntactic and semantic – in an exhaustive way.

References

1 Juan Aguilera and David Fernández-Duque. Strong completeness of provability logic for ordinal spaces. *The Journal of Symbolic Logic*, 82:608–628, November 2017. doi:10.1017/jsl.2017.3.

2 Miëtek Bak. Introspective Kripke models and normalisation by evaluation for the $\lambda^{\Box}$-calculus. 7th Workshop on Intuitionistic Modal Logic and Applications (IMLA 2017). https://github.com/mietek/imla2017/blob/master/doc/imla2017.pdf, 2017.

3 Lev Beklemishev and Albert Visser. Problems in the logic of provability. In *Mathematical Problems from Applied Logic I*, pages 77–136. Springer, 2006.

4 Bruno Bentzen. A Henkin-style completeness proof for the modal logic S5. *CoRR*, abs/1910.01697, 2019. arXiv:1910.01697.

5 Patrick Blackburn and Johan Van Benthem. Modal logic: a semantic perspective. In *Handbook of modal logic*, volume 3, pages 1–84. Elsevier, 2007.

6 Jasmin Christian Blanchette. Formalizing the Metatheory of Logical Calculi and Automatic Provers in Isabelle/HOL (Invited Talk). In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, page 1–13, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293880.3294087.

7 George Boolos. *The logic of provability.* Cambridge university press, 1995.

8 Alonzo Church. *Introduction to Mathematical Logic.* Princeton: Princeton University Press, 1956.

9 B. Jack Copeland. The genesis of possible worlds semantics. *Journal of Philosophical logic*, 31(2):99–137, 2002.

10 Melvin Fitting and Richard L Mendelsohn. *First-order modal logic*, volume 277. Springer Science & Business Media, 2012.

11 Kurt Gödel. Eine Interpretation des Intuitionistischen Aussagenkalküls. Ergebnisse eines Mathematischen Kolloquiums, 4: 39–40, 1933. english translation, with an introductory note by A.S. Troelstra. *Kurt Gödel, Collected Works*, 1:296–303, 1986.

12 Robert Goldblatt. Mathematical modal logic: A view of its evolution. In *Handbook of the History of Logic*, volume 7, pages 1–98. Elsevier, 2006.

13 Raul Hakli and Sara Negri. Does the deduction theorem fail for modal logic? *Synthese*, 187(3):849–867, 2012.

14 John Harrison. The HOL Light Theorem Prover. Web site: https://github.com/jrh13/hol-light.

15 John Harrison. HOL Light tutorial. http://www.cl.cam.ac.uk/~jrh13/hol-light/tutorial.pdf, 2017.

16 David Hilbert and Paul Bernays. *Grundlagen der Mathematik, Vol. I (1934), Vol II (1939).* Berlin: Springer, 1939.

17 Joachim Lambek and Philip J. Scott. *Introduction to higher-order categorical logic*, volume 7. Cambridge University Press, 1988.

18 Julius Michaelis and Tobias Nipkow. Formalized proof systems for propositional logic. In A. Abel, F. Nordvall Forsberg, and A. Kaposi, editors, *23rd Int. Conf. Types for Proofs and Programs (TYPES 2017)*, volume 104 of *LIPIcs*, pages 6:1–6:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

19 Sally Popkorn. *First steps in modal logic*. Cambridge University Press, 1994.

20 Natarajan Shankar. Towards mechanical metamathematics. *Journal of Automated Reasoning*, 1(4):407–434, 1985.

21 Robert M. Solovay. Provability interpretations of modal logic. *Israel journal of mathematics*, 25(3-4):287–304, 1976.

22 Yiming Xu and Michael Norrish. Mechanised modal model theory. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 518–533, Cham, 2020. Springer International Publishing.

Formal Verification of Termination Criteria for First-Order Recursive Functions

Cesar A. Muñoz
NASA Langley Research Center, Hampton, VA, USA

Mauricio Ayala-Rincón
Departments of Computer Science and Mathematics, Universidade de Brasília, Brazil

Mariano M. Moscato
National Institute of Aerospace, Hampton, VA, USA

Aaron M. Dutle
NASA Langley Research Center, Hampton, VA, USA

Anthony J. Narkawicz
Hampton, VA, USA

Ariane A. Almeida
Department of Computer Science, Universidade de Brasília, Brazil

Andréia B. Avelar
Faculdade de Planaltina, Universidade de Brasília, Brazil

Thiago M. Ferreira Ramos
Department of Computer Science, Universidade de Brasília, Brazil

Abstract

This paper presents a formalization of several termination criteria for first-order recursive functions. The formalization, which is developed in the Prototype Verification System (PVS), includes the specification and proof of equivalence of semantic termination, Turing termination, size change principle, calling context graphs, and matrix-weighted graphs. These termination criteria are defined on a computational model that consists of a basic functional language called PVS0, which is an embedding of recursive first-order functions. Through this embedding, the native mechanism for checking termination of recursive functions in PVS could be soundly extended with semi-automatic termination criteria such as calling contexts graphs.

2012 ACM Subject Classification Theory of computation → Models of computation; Software and its engineering → Software verification; Computing methodologies → Theorem proving algorithms

Keywords and phrases Formal Verification, Termination, Calling Context Graph, PVS

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.27

Supplementary Material *Other (NASA PVS Library)*: `https://github.com/nasa/pvslib`

Funding *Mariano M. Moscato*: corresponding author; research supported by the National Aeronautics and Space Administration under NASA/NIA Cooperative Agreement NNL09AA00A.
Anthony J. Narkawicz: At NASA at time of contribution.

1 Introduction

Advances in theorem proving have enabled the formal verification of algorithms used in safety-critical applications. For instance, the Prototype Verification System (PVS) [11] is extensively used at NASA in the verification of safety-critical algorithms of autonomous unmanned systems.[1] These algorithms are typically specified as recursive functions whose computations are well-behaved, i.e., they terminate for every possible input. In computer science, program termination is the quintessential example of a property that is undecidable. Alan Turing famously proved that it is impossible to construct an algorithm that decides whether or not another algorithm terminates on a given input [13]. Turing's proof applies

[1] For example, see `https://shemesh.larc.nasa.gov/fm`.

to algorithms written as Turing machines, but the proof extends to other formalisms for expressing computations such as λ-calculus, rewriting systems, and programs written in modern programming languages.

As is the case for other undecidable problems, there are syntactic and semantic restrictions, data structures, and heuristics that lead to a solution for subclasses of the problem. In Coq, for example, termination of well-typed functions is guaranteed by the Calculus of Inductive Constructions implemented in its type system [4]. Other theorem provers, such as ACL2, have incorporated syntactic conditions for checking termination of recursive functions [7]. In the Prototype Verification System (PVS), the user needs to provide a measure function over a well-founded relation that strictly decreases at every recursive call [11]. Despite the undecidability result, termination is routine, but is often a tedious and time-consuming stage in a formal verification effort.

This paper reports on the formalization of several termination criteria in PVS. In addition to the proper mechanism implemented in the type checker of PVS to assure termination of recursive definitions, this work also includes the formalization of more general techniques, such as the *size change principle* (SCP) presented by Lee et. al. [9]. The SCP principle states that if every infinite computation would give rise to an infinitely decreasing value sequence, then no infinite computation is possible. Later, Manolios and Vroon introduced a particular concretization of the SCP, namely the Calling Context Graphs (CCG) and demonstrated its practical usefulness in the ACL2 prover [10]. Avelar's PhD dissertation proposes an improvement on the CCG technique, based on a particular algebra on matrices [3]. The formalization reported in this paper includes all these criteria and proofs of equivalence between them. While the formalization itself has been available for some time as part of the NASA PVS Library, the goal of this paper is to report the main results. These results, which have been used in other works such as [2] and [12], have not been properly published before. Furthermore, this paper also presents a practical contribution: a mechanizable technique to automate (some) termination proofs of user-defined recursive functions in PVS.

For readability, this paper uses a stylized PVS notation. The development presented in this paper, including all lemmas and theorems, are formally verified in PVS and are available as part of the NASA PVS Library.

2 PVS & PVS0

PVS is an interactive theorem prover based on classical higher-order logic. The PVS specification language is strongly-typed and supports several typing features including predicate sub-typing, dependent types, inductive data types, and parametric theories. The expressiveness of the PVS type system prevents its type-checking procedure from being decidable. Hence, the type-checker may generate proof obligations to be discharged by the user. These proof obligations are called *Type Correctness Conditions* (TCCs). The PVS system includes several pre-defined proof strategies that automatically discharge most of the TCCs.

In PVS, a recursive function f of type [A→B] is defined by providing a *measure function* M of type [A→T], where T is an arbitrary type, and a *well-founded relation* R over elements in T. The termination TCCs produced by PVS for a recursive function f guarantee that the measure function M strictly decreases with respect to R at every recursive call of f.

▶ **Example 1.**
```
ackermann(m, n: ℕ) : RECURSIVE ℕ =
  IF m = 0 THEN n+1
  ELSIF n = 0 THEN ackermann(m-1,1)
```

```
ackermann_TCC5: OBLIGATION
  ∀(m,n: ℕ): n ≠ 0 ∧ m ≠ 0 ⇒ lex2(m,n-1) < lex2(m,n)

ackermann_TCC6: OBLIGATION
  ∀(m,n: ℕ, f: [{z: [ℕ×ℕ] | lex2(z`1,z`2) < lex2(m,n)} → ℕ]):
   n ≠ 0 ∧ m ≠ 0 ⇒ lex2(m-1, f(m,n-1)) < lex2(m,n)
```

Figure 1 Termination-related TCCs for the Ackermann function in Ex. 1.

```
    ELSE ackermann(m-1,ackermann(m,n-1))
    ENDIF
  MEASURE lex2(m,n) BY <
```

Example 1 provides a definiton of the Ackermann function in PVS. In this example, the type A is the tuple $[\mathbb{N}\times\mathbb{N}]$ and the type B is $\mathbb{N}$. The type T is `ordinal`, the type denoting ordinal numbers in PVS. The measure function **lex2** maps a tuple of natural numbers into an ordinal number. Finally, the well-founded relation R is the order relation "<" on ordinal numbers. The termination-related TCCs generated by the PVS type-checker for the Ackermann function are shown in Figure 1. Since all the TCCs are automatically discharged by a PVS built-in proof strategy, the PVS semantics guarantees that the function `ackermann` is well defined on all inputs.

PVS0 is a basic functional language used in this paper as a computational model for first-order recursive functions in PVS. More precisely, PVS0 is an embedding of univariate first-order recursive functions of type $[\mathcal{V}al\to\mathcal{V}al]$ for an arbitrary generic type $\mathcal{V}al$. The syntactic expressions of PVS0 are defined by the grammar

$$e ::= \texttt{cnst}(v) \mid \texttt{vr} \mid \texttt{op1}(n,e) \mid \texttt{op2}(n,e,e) \mid \texttt{rec}(e) \mid \texttt{ite}(e,e,e),$$

where v is a value of type $\mathcal{V}al$ and n is a natural number. Furthermore, $\texttt{cnst}(v)$ denotes a constant with value v, `vr` denotes a unique variable, `op1` and `op2` denote unary and binary operators respectively, `rec` denotes a recursive call, and `ite` denotes a conditional expression ("if-then-else"). The first parameter of `op1` and `op2` is an index used to identify built-in operators of type $[\mathcal{V}al\to\mathcal{V}al]$ and $[[\mathcal{V}al\times\mathcal{V}al]\to\mathcal{V}al]$, respectively. In the following, the collection of PVS0 expressions is referred to as $\texttt{PVS0Expr}_{\mathcal{V}al}$, where the type parameter for `PVS0Expr` is omitted when possible to lighten the notation. The PVS0 programs with values in $\mathcal{V}al$, denoted by $\texttt{PVS0}_{\mathcal{V}al}$, are 4-tuples of the form $(O_1,O_2,\perp,e)$, such that

- O_1 is a list of unary operators of type $[\mathcal{V}al\to\mathcal{V}al]$, where $O_1(i)$, i.e., the i-th element of the list O_1, interprets the index i as referred by in the application of `op1`,
- O_2 is a list of binary operators of type $[[\mathcal{V}al\times\mathcal{V}al]\to\mathcal{V}al]$, where $O_2(i)$ interprets the index i in applications of `op2`,
- $\perp$ is a constant of type $\mathcal{V}al$ representing the Boolean value *false* in the conditional construction `ite`, and
- e is a expression from `PVS0Expr`: the syntactic representation of the body of the program.

Operators in O_1 and O_2 are PVS pre-defined functions, whose evaluation is considered to be atomic in the proposed computational model. These operators make it easy to modularly embed first-order PVS recursive functions in PVS0, while maintaining non-recursive PVS functions directly available to PVS0 definitions. Henceforth, if $\texttt{p} = (O_1,O_2,\perp,e)$ is a PVS0 program, the symbols $\texttt{p}_{O_1}$, $\texttt{p}_{O_2}$, $\texttt{p}_{\perp}$, and $\texttt{p}_e$ denote, respectively, the first, second, third, and fourth elements of the tuple. Since there is only one variable available to write PVS0 programs, arguments of binary functions such as Ackermann's need to be encoded in $\mathcal{V}al$, for example using tuples as shown in Example 2.

▶ **Example 2.** The Ackermann function of Example 1 can be implemented as the $\mathtt{PVS0}_{[\mathbb{N}\times\mathbb{N}]}$ program `ack` $\equiv (O_1, O_2, \bot, e)$, where the type parameter $\mathcal{V}al$ of PVS0 is instantiated with the type of pair of natural numbers, i.e., $[\mathbb{N}\times\mathbb{N}]$. In this encoding, the first projection of the result of the program represents the output of the function. The components of `ack` are defined below.

- $O_1(0)((m,n)) \equiv \mathtt{IF}\ m = 0\ \mathtt{THEN}\ \top\ \mathtt{ELSE}\ \bot\ \mathtt{ENDIF}$.
- $O_1(1)((m,n)) \equiv \mathtt{IF}\ n = 0\ \mathtt{THEN}\ \top\ \mathtt{ELSE}\ \bot\ \mathtt{ENDIF}$.
- $O_1(2)((m,n)) \equiv (n+1, 0)$.
- $O_1(3)((m,n)) \equiv \mathtt{IF}\ m = 0\ \mathtt{THEN}\ \bot\ \mathtt{ELSE}\ (\max(0, m-1), 1)\ \mathtt{ENDIF}$.
- $O_1(4)((m,n)) \equiv \mathtt{IF}\ n = 0\ \mathtt{THEN}\ \bot\ \mathtt{ELSE}\ (m, \max(0, n-1))\ \mathtt{ENDIF}$.
- $O_2(0)((m,n),(i,j)) \equiv \mathtt{IF}\ m = 0\ \mathtt{THEN}\ \bot\ \mathtt{ELSE}\ (\max(0, m-1), i)\ \mathtt{ENDIF}$.
- $\bot \equiv (0,0)$, and for convenience $\top \equiv (1,0)$.
- e $\equiv$ `ite(op1(0,vr), op1(2,vr),`
  `ite(op1(1,vr), rec(op1(3,vr)), rec(op2(0,vr,rec(op1(4,vr)))))).`

Example 2 illustrates the use of built-in operators in PVS0. Any unary or binary PVS function can be used as an operator in the construction of a PVS0 program. In order to show that `ack` correctly encodes the Ackermann function, it is necessary to define the operational semantics of PVS0.

2.1 Semantic Relation

Given a PVS0 program `p`, the semantic evaluation of a `PVS0Expr` expression e_i is given by the relation ε defined as follows. Intuitively, it holds when given a subexpression e_i of a program `p`, the evaluation of e_i on the input value v_i results in the output value v_o.

▶ **Definition 3** (Semantic Relation). *Let* `p` *be a PVS0 program on a generic type* $\mathcal{V}al$*,* e_i *be an expression in* `PVS0Expr`*, and* v_i, v_o, v, v', v'' *be values from* $\mathcal{V}al$*. The relation* $\varepsilon(\mathtt{p})(e_i, v_i, v_o)$ *holds if and only if*

$$\begin{cases} v_o = v & \textit{if } e_i = \mathtt{cnst}(v) \\ v_o = v_i & \textit{if } e_i = \mathtt{vr} \\ \exists\, v' : \varepsilon(\mathtt{p})(e_1, v_i, v') \wedge v_o = \chi_1(\mathtt{p})(j, v') & \textit{if } e_i = \mathtt{op1}(j, e_1) \\ \exists\, v', v'' : \varepsilon(\mathtt{p})(e_1, v_i, v') \wedge \varepsilon(\mathtt{p})(e_2, v_i, v'') & \\ \qquad\qquad \wedge\, v_o = \chi_2(\mathtt{p})(j, v', v'') & \textit{if } e_i = \mathtt{op2}(j, e_1, e_2) \\ \exists\, v' : \varepsilon(\mathtt{p})(e_1, v_i, v') \wedge \varepsilon(\mathtt{p})(\mathtt{p}_e, v', v_o) & \textit{if } e_i = \mathtt{rec}(e_1) \\ \exists\, v' : \varepsilon(\mathtt{p})(e_1, v_i, v') \wedge (v' \neq \mathtt{p}_\bot \Rightarrow \varepsilon(\mathtt{p})(e_2, v_i, v_o)) & \\ \qquad \wedge\, (v' = \mathtt{p}_\bot \Rightarrow \varepsilon(\mathtt{p})(e_3, v_i, v_o)) & \textit{if } e_i = \mathtt{ite}(e_1, e_2, e_3) \end{cases}$$

where

$$\chi_1(\mathtt{p})(j, v) = \begin{cases} \mathtt{p}_{O_1}(j)(v) & \textit{if } j < |\mathtt{p}_{O_1}| \\ \mathtt{p}_\bot & \textit{otherwise.} \end{cases}$$

$$\chi_2(\mathtt{p})(j, v_1, v_2) = \begin{cases} \mathtt{p}_{O_2}(j)(v_1, v_2) & \textit{if } j < |\mathtt{p}_{O_2}| \\ \mathtt{p}_\bot & \textit{otherwise.} \end{cases}$$

The following lemma states that the `ack` program encodes the function `ackermann`.

▶ **Lemma 4.** *For all $n, m, k \in \mathbb{N}$, $\mathtt{ackermann}(m, n) = k$ if and only if there exists $i \in \mathbb{N}$ such that $\varepsilon(\mathtt{ack})(\mathtt{ack}_e, (m, n), (k, i))$.*

This lemma can be proved by structural induction on the definition of the function `ackermann` and the relation ε. A proof of this kind of statement is usually tedious and long. However, it is fully mechanizable in PVS assuming that the function and the PVS0 program share the same syntactical structure. A proof strategy that automatically discharges equivalences between PVS functions and PVS0 programs was developed. The following theorem shows that the semantic relation ε is deterministic.

▶ **Theorem 5.** *Let $\mathtt{p}$ be a PVS0 program. For any `PVS0Expr` expression e_i and values $v_i, v'_o, v''_o \in \mathcal{V}al$, $\varepsilon(\mathtt{p})(e_i, v_i, v'_o)$ and $\varepsilon(\mathtt{p})(e_i, v_i, v''_o)$ implies $v'_o = v''_o$.*

PVS0 enables the encoding on non-terminating functions. The predicate *ε-determined*, defined below, holds when a PVS0 program encodes a function that returns a value for a given input.

▶ **Definition 6** (ε-determination)**.** *A PVS0 program $\mathtt{p}$ is said to be* ε-determined *for an input value $v_i \in \mathcal{V}al$ (denoted by $D_\varepsilon(\mathtt{p}, v_i)$) when $\exists v_o \in \mathcal{V}al : \varepsilon(\mathtt{p})(\mathtt{p}_e, v_i, v_o)$.*

2.2 Functional Semantics

The operational semantics of PVS0 can be expressed by a function $\chi : [\mathtt{PVS0} \to [\mathtt{PVS0Expr} \times \mathcal{V}al \times \mathbb{N}] \to \mathcal{V}al \uplus \{\diamondsuit\}]$. This function returns either a value of type $\mathcal{V}al$ or a distinguished value $\diamondsuit \notin \mathcal{V}al$. The natural number argument represents an upper bound on the number of nested recursive calls that are to be evaluated. If this bound is reached and no final value has been computed, the function returns $\diamondsuit$.

▶ **Definition 7** (Semantic Function)**.** *Let $\mathtt{p}$ be a PVS0 program, e_i a `PVS0Expr` expression, v_i a value from $\mathcal{V}al$, n a natural number, $v' = \chi(\mathtt{p})(e_1, v_i, n)$, and $v'' = \chi(\mathtt{p})(e_2, v_i, n)$.*

$$\chi(\mathtt{p})(e_i, v_i, n) \equiv \begin{cases} v & \textit{if } n > 0 \textit{ and } e_i = \mathtt{cnst}(v) \\ v_i & \textit{if } n > 0 \textit{ and } e_i = \mathtt{vr} \\ \chi_1(\mathtt{p})(j, v') & \textit{if } n > 0, e_i = \mathtt{op1}(j, e_1), \textit{ and } v' \neq \diamondsuit \\ \chi_2(\mathtt{p})(j, v', v'') & \textit{if } n > 0, e_i = \mathtt{op2}(j, e_1, e_2), \\ & \quad v' \neq \diamondsuit, \textit{ and } v'' \neq \diamondsuit \\ \chi(\mathtt{p})(e, v', n-1) & \textit{if } n > 0, e_i = \mathtt{rec}(e_1), \textit{ and } v' \neq \diamondsuit \\ \chi(\mathtt{p})(e_2, v_i, n) & \textit{if } n > 0, e_i = \mathtt{ite}(e_1, e_2, e_3), v' \neq \diamondsuit, \\ & \quad \textit{and } v' \neq \mathtt{p}_\perp \\ \chi(\mathtt{p})(e_3, v_i, n) & \textit{if } n > 0, e_i = \mathtt{ite}(e_1, e_2, e_3), v' \neq \diamondsuit, \\ & \quad \textit{and } v' = \mathtt{p}_\perp \\ \diamondsuit & \textit{otherwise.} \end{cases}$$

The following theorem states that the semantic relation ε and the semantic function χ are equivalent.

▶ **Theorem 8.** *For any PVS0 program $\mathtt{p}$, $v_i, v_o \in \mathcal{V}al$ and $e_i \in$ `PVS0Expr`, $\varepsilon(\mathtt{p})(e_i, v_i, v_o)$ if and only if $v_o = \chi(\mathtt{p})(e_i, v_i, n)$, for some $n \in \mathbb{N}$.*

A program $\texttt{p}$ is χ-determined for an input v_i, as defined below, if the evaluation of $\texttt{p}(v_i)$ produces a value in a finite number of nested recursive calls.

▶ **Definition 9** (χ-determination). *A PVS0 program $\texttt{p}$ is said to be* χ-determined *for an input value $v_i \in \mathcal{V}al$ (denoted by $D_\chi(\texttt{p}, v_i)$) when there is an $n \in \mathbb{N}$ such that $\chi(\texttt{p})(\texttt{p}_e, v_i, n) \neq \diamondsuit$.*

As a corollary of Theorem 8, the notions of ε-determination and χ-determination coincide.

▶ **Theorem 10.** *For all $\texttt{p} \in PVS0_{\mathcal{V}al}$ and value $v_i : \mathcal{V}al$, $D_\varepsilon(\texttt{p}, v_i)$ if and only if $D_\chi(\texttt{p}, v_i)$.*

In Definition 9, there may be multiple (in fact, infinite) natural numbers n that satisfy $\chi(\texttt{p})(\texttt{p}_e, v_i, n) \neq \diamondsuit$. The following definition distinguishes the minimum of those numbers.

▶ **Definition 11** (μ). *Let $\texttt{p}$ be a PVS0 program and v_i a value in $\mathcal{V}al$ such that $D_\chi(\texttt{p}, v_i)$, the minimum number of recursive calls needed to produce a result (denoted by $\mu(\texttt{p}, v_i)$) is formally defined as* $\min(\{n \in \mathbb{N} \mid \chi(\texttt{p})(\texttt{p}_e, v_i, n) \neq \diamondsuit\})$.

If $\texttt{p}$ is χ-determined for a value v_i, then for any $n \geq \mu(\texttt{p}, v_i)$ the evaluation of $\chi(\texttt{p})(\texttt{p}_e, v_i, n)$ results in a value from $\mathcal{V}al$. This remark is formalized by the following lemma.

▶ **Lemma 12.** *Let $\texttt{p}$ be a PVS0 program and v_i a value from $\mathcal{V}al$ such that $D_\chi(\texttt{p}, v_i)$. For any $n \in \mathbb{N}$ such that $n \geq \mu(\texttt{p}, v_i)$, $\chi(\texttt{p})(\texttt{p}_e, v_i, n) = \chi(\texttt{p})(\texttt{p}_e, v_i, \mu(\texttt{p}, v_i))$.*

2.3 Semantic Termination

The notion of termination for PVS0 programs is defined using the notions of determination from Section 2.2.

▶ **Definition 13** (ε-termination and χ-termination). *A PVS0 program $\texttt{p} \in PVS0_{\mathcal{V}al}$ is said to be* ε-terminating *(noted $T_\varepsilon(\texttt{p})$) when $\forall v_i \in \mathcal{V}al : D_\varepsilon(\texttt{p}, v_i)$. It is said to be* χ-terminating *($T_\chi(\texttt{p})$) when $\forall v_i \in \mathcal{V}al : D_\chi(\texttt{p}, v_i)$.*

As a corollary of Theorem 10, the notions of ε-termination and χ-termination coincide.

▶ **Theorem 14.** *For every PVS0 program $\texttt{p}$, $T_\varepsilon(\texttt{p})$ if and only if $T_\chi(\texttt{p})$.*

Not all PVS0 programs are terminating. For example, consider the PVS0 program $\texttt{p}'$ with body $\texttt{rec}(\texttt{vr})$. It can be proven that $D_\varepsilon(\texttt{p}', v_i)$ does not hold for any $v_i \in \mathcal{V}al$. Hence, $T_\varepsilon(\texttt{p}')$ does not hold and, equivalently, nor does $T_\chi(\texttt{p}')$. Since terminating programs compute a value for every input, the function χ can be refined into an evaluation function for terminating programs that does not depend on the existence of a distinguished value outside $\mathcal{V}al$, such as $\diamondsuit$.

▶ **Definition 15.** *Let $PVS0_{\downarrow_\varepsilon}$ be the collection of PVS0 programs for which T_ε holds, let $\texttt{p} \in PVS0_{\downarrow_\varepsilon}$, and v_i be a value in $\mathcal{V}al$. The semantic function for terminating programs $\epsilon : [PVS0_{\downarrow_\varepsilon} \to \mathcal{V}al \to \mathcal{V}al]$ is defined in the following way.*

$\epsilon(\texttt{p})(v_i) \equiv \epsilon_e(\texttt{p})(\texttt{p}_e, v_i)$, where $v' = \epsilon_e(\texttt{p})(e_1, v_i)$, $v'' = \epsilon_e(\texttt{p})(e_2, v_i)$, and

$$\epsilon_e(\texttt{p})(e_i, v_i) \equiv \begin{cases} v & \text{if } e_i = \texttt{cnst}(v) \\ v_i & \text{if } e_i = \texttt{vr} \\ \chi_1(\texttt{p})(j,\, v') & \text{if } e_i = \texttt{op1}(j, e_1) \\ \chi_2(\texttt{p})(j,\, v',\, v'') & \text{if } e_i = \texttt{op2}(j, e_1, e_2) \\ \epsilon_e(\texttt{p})(e, v') & \text{if } e_i = \texttt{rec}(e_1) \\ \epsilon_e(\texttt{p})(e_2, v_i) & \text{if } e_i = \texttt{ite}(e_1, e_2, e_3) \text{ and } \epsilon_e(\texttt{p})(e_1, v_i) \neq \texttt{p}_\perp \\ \epsilon_e(\texttt{p})(e_3, v_i) & \text{if } e_i = \texttt{ite}(e_1, e_2, e_3) \text{ and } \epsilon_e(\texttt{p})(e_1, v_i) = \texttt{p}_\perp \end{cases}$$

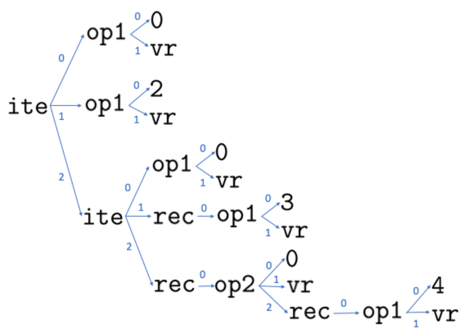


Figure 2 Abstract syntax tree of the Ackermann function from Example 2.

▶ **Theorem 16.** *For all terminating PVS0 program* $\mathtt{p}$*, i.e.,* $T_\varepsilon(\mathtt{p})$ *holds, and values* $v_i, v_o \in \mathcal{V}al$*,* $\varepsilon(\mathtt{p})(\mathtt{p}_e, v_i, v_o)$ *holds if and only if* $\epsilon(\mathtt{p})(v_i) = v_o$*.*

While T_ε and T_χ provide semantic definitions of termination, these definitions are impractical as termination criteria, since they involve an exhaustive examination of the whole universe of values in $\mathcal{V}al$. The rest of this paper formalizes termination criteria that yield mechanical termination analysis techniques.

3 Turing Termination Criterion

In contrast to the purely semantic notions of termination presented in Section 2, the so-called *Turing termination criterion* relies on the syntactic structure of recursive programs. In particular, this termination criterion uses a characterization of the input values that lead to the evaluation of recursive call subexpressions, i.e., $\mathtt{rec}(e)$. In order to define such a characterization, it is necessary to formalize a way to identify univocally a particular subexpression of a given PVS0 program. Furthermore, the subexpression as well as its actual position must be identified. If a given program body contains several repetitions of the same expression, such as op2(0,rec(vr),rec(vr)), which has two occurrences of rec(vr), the criterion needs them to be distinguishable from one another. Such a reference for subexpressions can be formally defined using the abstract syntax tree of the enclosing expression. To illustrate the idea, Figure 2 depicts a graphical representation of the abstract syntax tree of the ack program. A unique identifier for a given subexpression can be constructed by collecting all the numbers labeling the edges from the subexpression to the root of the tree. For example, the sequence of numbers that identify the subexpression rec(op1(4,vr)) is $\langle 2, 0, 2, 2\rangle$. A syntax tree labeled using these sequences is called a *labeled syntax tree.*

▶ **Definition 17** (Valid Path). *Let* $\mathtt{p}$ *be a PVS0 program, a finite sequence of natural numbers* p *is a* Valid Path *of* $\mathtt{p}$ *if* p *determines a path in the labeled syntax tree of* $\mathtt{p}$ *from any node* e *to the root of the tree. In that case,* p *is said to* reach e *in* $\mathtt{p}$*.*

The notion of path is strictly syntactic. Nevertheless, a semantic correlation is also needed to state termination criteria focused on how the inputs change along successive recursive calls, as is the case for Turing termination criterion. A semantic way to identify a subexpression e of a given program p is to recognize all the values that exercise the particular subexpression e when used as inputs for the evaluation of p. It is possible to characterize such values by collecting all the expressions that act as guards for the conditional expressions traversed for a given path reaching e.

Continuing the example based on the `ack` program, for the path $\langle 2,0,2,2\rangle$ reaching `rec(op1(4,vr))`, such expressions would be `op1(0,vr)` and `op1(1,vr)`. For that specific path, the values to be characterized are the ones that falsify both guard expressions, i.e., the values for which both expressions evaluate to $\mathtt{p}_\perp$. Nevertheless, for the path $\langle 1,2\rangle$ reaching `rec(op1(3,vr))`, the collected expressions are the same, but it is necessary for the latter not to evaluate to $\mathtt{p}_\perp$ in order to characterize the input values that would exercise `rec(op1(3,vr))`.

The previous example shows that it is necessary not only to collect the guard expressions, but also to determine whether each one needs to evaluate to $\mathtt{p}_\perp$ or not.

▶ **Definition 18** (Polarized Expression). *Given a* `PVS0Expr` *expression e, the* polarized version *of e is a pair* $[$`PVS0Expr` $\times\ \{0,1\}]$ *such that $(e,0)$, abbreviated as $\neg e$, indicates that e should evaluate to $\mathtt{p}_\perp$ and the pair $(e,1)$, which is abbreviated simply as e, indicates the contrary.*

For a given program $\mathtt{p}$, an input value v_i, and a polarized expression $c = (e,b)$ with $b \in \{0,1\}$, c is said to be valid *when the condition expressed by it holds. The predicate $\varepsilon_\pm$ defined below formalizes this notion.*

$$\varepsilon_\pm(\mathtt{p})(c,v_i) \equiv \begin{cases} \varepsilon(\mathtt{p})(e,v_i,\mathtt{p}_\perp) & \text{if } b = 0,\\ \neg\varepsilon(\mathtt{p})(e,v_i,\mathtt{p}_\perp) & \text{otherwise.}\end{cases}$$

The semantic characterization of a particular subexpression is formalized by the notion of list of path conditions defined below.

▶ **Definition 19** (Path Conditions). *Let p be a valid path of a PVS0 program $\mathtt{p}$ and e the subexpression of $\mathtt{p}_e$ reached by p. The list of polarized guard expressions of $\mathtt{p}$ that are needed to be valid in order for the evaluation of $\mathtt{p}$ to involve the expression e is called the* list of path conditions *of p.*

▶ **Definition 20** (Calling Context). *A* calling context *of a program $\mathtt{p}$ is a tuple $(\mathtt{rec}(e'),p,\mathbf{c})$ containing: a path p, which is valid in $\mathtt{p}$, a recursive-call expression $\mathtt{rec}(e')$ contained in $\mathtt{p}_e$ and reached by p, and the list $\mathbf{c}$ of path conditions of p. The collection of all calling contexts of $\mathtt{p}$ is denoted by $\mathbf{cc}(\mathtt{p})$.*

The notion of calling context captures both the syntactic and the semantic characterizations of the subexpressions of a program that denote recursive calls.

▶ **Example 21.** The calling contexts for the `ack` function from Example 2 are:

- (`rec(op1(3,vr))`, $\langle 1,2\rangle$, $\langle\neg$`op1(0,vr)`, `op1(1,vr)`$\rangle$),
- (`rec(op2(0,vr,rec(op1(4,vr))))`, $\langle 2,2\rangle$, $\langle\neg$`op1(0,vr)`, $\neg$`op1(1,vr)`$\rangle$), and
- (`rec(op1(4,vr))`, $\langle 2,0,2,2\rangle$, $\langle\neg$`op1(0,vr)`, $\neg$`op1(1,vr)`$\rangle$).

An input value v_i is said to *exercise* a calling context $\mathbf{cc} = (e,p,\mathbf{c})$ in a program $\mathtt{p}$ when $\varepsilon_\pm(\mathtt{p})(c,v_i)$ holds. A program $\mathtt{p}$ is TCC-terminating if for each calling context $\mathbf{cc}$ in $\mathtt{p}$ and every input value v_i exercising $\mathbf{cc}$, the value of the expression used as argument by the call in $\mathbf{cc}$ is smaller than v_i. In this context, a value is considered smaller than another one if the former is closer to the bottom induced by a well-founded relation than the latter.

▶ **Definition 22** (TCC-termination). *A PVS0 program $\mathtt{p}$ is said to be* TCC-terminating*, or* Turing-terminating*, on a measuring type M if there exist a function $m : [\mathcal{V}al \to M]$ and a well-founded relation $<_M$ on M such that for all calling context $\mathbf{cc} = (\mathtt{rec}(e),p,\mathbf{c})$ among the calling contexts of $\mathtt{p}$, for all $v_i, v_o \in \mathcal{V}al$, if $\varepsilon_\pm(\mathtt{p})(\mathbf{c},v_i)$ and $\varepsilon(\mathtt{p})(e,v_i,v_o)$ hold, then $m(v_o) <_M m(v_i)$.*

The notion of TCC-termination on a program p is denoted by the predicate $T_T^{[M,<_M,m]}(\mathtt{p})$, which is parametric on the measure type M, the well-founded relation $<_M$, and the measure function m. TCC-termination is equivalent to ε-termination (and, therefore, to χ-termination) as stated by Theorem 25 below. A key construction used in the proof of Theorem 25 is the function Ω, defined as follows.

▶ **Definition 23** (Ω). *Let $<_{\mathtt{p},m}$ be a binary relation on $\mathcal{V}al$ defined as $v_1 <_{\mathtt{p},m} v_2$ if and only if $m(v_1) <_M m(v_2)$ and the evaluation of* p *with v_2 as input reaches a recursive call* $\mathtt{rec}(e)$ *such that $\varepsilon(\mathtt{p})(e, v_2, v_1)$ holds. Then, $\Omega_{\mathtt{p},m}(v) \equiv \min(\{i : \mathbb{N}^+ \mid \forall v' \in \mathcal{V}al : \neg(v' <^i_{\mathtt{p},m} v)\})$ where $v' <^i_{\mathtt{p},m} v$ denotes a chain of $i+1$ values related by $<_{\mathtt{p},m}$ with endpoints in v' and v.*

The following lemma states a relation between μ, the number of nested recursive calls in the evaluation of a particular input v, and $\Omega_{\mathtt{p},m}$ for the same input v.

▶ **Lemma 24.** *Let* p *be a TCC-terminating* PVS0 *program, i.e.,* p *satisfies $T_T^{[M,<_M,m]}(\mathtt{p})$ for a measure type M, a well-founded relation $<_M$ over M, and a measure function m. For any value $v \in \mathcal{V}al$, $\mu(\mathtt{p}, v) \leq \Omega_{\mathtt{p},m}(v)$.*

▶ **Theorem 25.** *Let* p *be a PVS0 program, $T_\varepsilon(\mathtt{p})$ holds if and only if there exist a measure type M, a well-founded relation $<_M$ on M, and a measure function m such that $T_T^{[M,<_M,m]}(\mathtt{p})$ holds as well.*

Proof. Assuming $T_\varepsilon(\mathtt{p})$, it can be proved that $T_T^{[\mathbb{N},<,\mu_{\mathtt{p}}]}(\mathtt{p})$ holds, where $\mu_{\mathtt{p}}(v) = \mu(\mathtt{p}, v)$. The function $\mu_{\mathtt{p}}(v)$ is well defined for every v since $T_\varepsilon(\mathtt{p})$ holds and then, by Theorem 14, $D_\chi(\mathtt{p}, v)$ holds as well. Following the definition of χ and the determinism of ε (Lemma 5), it can be seen that $\mu_{\mathtt{p}}(v_o) < \mu_{\mathtt{p}}(v_i)$ for each pair of values v_i, v_o such that $\varepsilon_\pm(\mathtt{p})(\mathbf{c}, v_i)$ and $\varepsilon(\mathtt{p})(e, v_i, v_o)$ for every calling context $(\mathtt{rec}(e), p, \mathbf{c})$ in p. The opposite implication can be proved stating that if $T_T^{[M,<_M,m]}(\mathtt{p})$ holds, for every $v \in \mathcal{V}al$ and any subexpression e of p, there exists a natural number $n \leq \Omega_{\mathtt{p},m}(v)$ such that $\chi(\mathtt{p})(e, v_i, n) \neq \diamondsuit$, which assures $T_\varepsilon(\mathtt{p})$ by Theorem 14. The proof of such a property proceeds by induction on the lexicographic order given by $(m(v), |e|)$, where $|e|$ denotes the size of the expression e. ◀

Theorem 25 can be used as a practical tool to prove ε-termination of PVS0 programs, as illustrated by the following lemma.

▶ **Lemma 26.** *The* PVS0 *program* ack *from Example 2 is ε-terminating, i.e., $T_\varepsilon(\mathtt{ack})$ holds.*

Proof. In order to use the Theorem 25, it is necessary to prove first that there exist a measure type M, a well-founded relation $<_M$ over M, and a measure function m such that $T_T^{[M,<_M,m]}(\mathtt{ack})$ holds. Let M be the type of pairs of natural numbers $[\mathbb{N} \times \mathbb{N}]$, m the identity function, and $<_M$ the lexicographic order on $[\mathbb{N} \times \mathbb{N}]$, i.e., $(a, b) <_{lex} (c, d) \equiv a < c \vee (a = c \wedge b < d)$ where $<$ is the less-than relation on natural numbers. To prove that $T_T^{[[\mathbb{N}\times\mathbb{N}],<_{lex},id]}(\mathtt{ack})$ holds, it suffices to check that for every input pair v_i, leading to any of the recursive-call subexpressions $\mathtt{rec}(e)$ in ack, v_i is such that for every pair v_o satisfying $\varepsilon(\mathtt{ack})(e, v_i, v_o)$, $v_o <_{lex} v_i$.

There are only three recursive calls in ack (see Example 2), namely: `rec(op1(3,vr))`, `rec(op1(4,vr))`, and `rec(op2(0,vr,rec(op1(4,vr))))`. Each of them determines a case in the proof. For the first subexpression, note that any input value v_i leading to `rec(op1(3,vr))` must be such that $\pi_1(v_i) \neq 0$ and $\pi_2(v_i) = 0$, in order to falsify the guard in the outermost if-then-else and validate the guard in the innermost conditional. Because of the function $O_1(3)$ used to interpret $\mathtt{op1}(3, \cdot)$, for every v_o such that $\varepsilon(\mathtt{ack})(e, v_i, v_o)$ holds, $\pi_1(v_o)$ must be equal to $\pi_1(v_i) - 1$; hence, $v_o <_{lex} v_i$ holds. For the other recursive-call subexpressions in ack, the values v_i that lead to them satisfy $\pi_1(v_i) \neq 0$ and $\pi_2(v_i) \neq 0$. In particular, for the case of `rec(op1(4,vr))`, the function $O_1(4)$ forces any v_o for which $\varepsilon(\texttt{ack})(e, v_i, v_o)$ holds, to be equal to $(\pi_1(v_i), \pi_2(v_i) - 1)$, satisfying $v_o <_{lex} v_i$ as well. Finally, for the values v_i reaching `rec(op2(0,vr,rec(op1(4,vr))))` and because of $O_2(0)$, the first coordinate of v_o must be $\pi_1(v_i) - 1$, which is enough to conclude that $v_o <_{lex} v_i$ holds. Then, $T_T^{[[\mathbb{N}\times\mathbb{N}],<_{lex},id]}(\texttt{ack})$ holds and, by Theorem 25, $T_\varepsilon(\texttt{ack})$ holds as well. ◀

The inequalities of the form $v_o <_{lex} v_i$ that are proved in Lemma 26 correspond to the actual termination correctness conditions generated by the PVS type checker for the function `ackermann` defined in Example 1.

4 Calling Context Graphs

The Size Change Principle (SCP) states that "a program terminates on all inputs if every infinite call sequence (following program control flow) would cause an infinite descent in some data values" [9]. Calling Context Graphs is a technique that implements the SCP [10].

▶ **Definition 27** (Valid Trace). *Given $\texttt{p} \in PVS0$, an infinite sequence $\mathbf{cc} = \langle \texttt{rec}(e_i), p_i, \mathbf{c}_i\rangle_{i\in\mathbb{N}}$ of calling contexts of $\texttt{p}$, and an infinite sequence of values $\mathbf{v}$ from $\mathcal{V}al$, $\mathbf{cc}$ and $\mathbf{v}$ are said to form a* valid trace of calls *if the following predicate τ holds.*[2]

$$\tau_{\texttt{p}}(\mathbf{cc}, \mathbf{v}) \equiv \forall (i : nat) : (\varepsilon_\pm(\texttt{p})(\mathbf{c}_i, \mathbf{v}_i) \wedge \varepsilon(\texttt{p})(e_i, \mathbf{v}_i, \mathbf{v}_{i+1})).$$

▶ **Definition 28** (SCP-Termination). *A PVS0 program $\texttt{p}$ is said to be* SCP-terminating*, denoted by $T_{SCP}(\texttt{p})$, if there are no infinite sequence $\mathbf{cc}$ of calling contexts of $\texttt{p}$ and no infinite sequence $\mathbf{v}$ of values in $\mathcal{V}al$ such that $\tau(\mathbf{cc}, \mathbf{v})$ holds.*

▶ **Theorem 29.** *For all $\texttt{p} \in PVS0$, $T_\varepsilon(\texttt{p})$ if and only if $T_{SCP}(\texttt{p})$.*

Proof. By Theorem 25 it is enough to prove that $T_T(\texttt{p})$ and $T_{SCP}(\texttt{p})$ are equivalent. Proving $T_{SCP}(\texttt{p})$ given $T_T(\texttt{p})$ is straightforward. To prove the other direction, it is necessary to use $\Omega_{\texttt{p},m}$. Since one has $T_{SCP}(\texttt{p})$, it is possible to provide a relation between parameters and arguments of recursive calls and prove that it is well-founded. Similarly to the proof of Theorem 25, the closure of this relation is then used to parametrize the function $\Omega_{\texttt{p},m}$, which provides the height of the tree of evaluation of recursive calls as the needed measure. ◀

▶ **Definition 30.** *Let $<$ be a well-founded relation over $\mathcal{V}al$, $\texttt{SCP}_<(\texttt{p})$ holds if for all infinite sequence $\mathbf{cc}$ of calling contexts of $\texttt{p}$ and for all infinite sequence $\mathbf{v}$ of values in $\mathcal{V}al$ such that $\tau(\mathbf{cc}, \mathbf{v})$ holds, $\mathbf{v}$ is a decreasing sequence on $<$, i.e., for all $i \in \mathbb{N}$, $v_{i+1} < v_i$.*

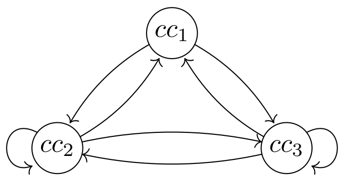
▶ **Theorem 31.** *For all $\texttt{p} \in PVS0_{\mathcal{V}al}$, $T_{SCP}(\texttt{p})$ if and only if $SCP_<(\texttt{p})$ for a well-founded relation $<$ over $\mathcal{V}al$.*

The proof of Theorem 31 uses the fact that every well-founded order provides a non-infinite decreasing sequence of elements.

▶ **Definition 32.** *A* Calling Context Graph *of a PVS0 program $\texttt{p}$ ($\texttt{p} \in PVS0_{\mathcal{V}al}$) is a directed graph $G_{\texttt{p}} = (V, E)$ with a node in V for each calling context in $\texttt{p}$ such that given two calling contexts of $\texttt{p}$ $(\texttt{rec}(e_a), P_a, C_a)$ and $(\texttt{rec}(e_b), P_b, C_b)$, if*

$$\exists (v_a, v_b : \mathcal{V}al) : \varepsilon_\pm(\texttt{p})(C_a, v_a) \wedge \varepsilon(\texttt{p})(e_a, v_a, v_b) \wedge \varepsilon_\pm(\texttt{p})(C_b, v_b),$$

[2] Since $\varepsilon_\pm$ can be straightforwardly extended to lists of polarized expressions, the same symbol is used for both versions along the text.



- $cc_1 = (ack(m-1,1), m \neq 0 \wedge n = 0)$
- $cc_2 = (ack(m-1, ack(m, n-1)), m \neq 0 \wedge n \neq 0)$
- $cc_3 = (ack(m, n-1), m \neq 0 \wedge n \neq 0)$

Figure 3 A possible CCG for the Ackermann function.

then the edge $\langle(\texttt{rec}(e_a), P_a, C_a), (\texttt{rec}(e_b), P_b, C_b)\rangle \in E$.

The condition on the edges admits any fully connected graph of calling contexts to be considered a CCG. For the sake of exemplification, another possible CCG for the Ackermann function as defined in the Example 1 is depicted in the Figure 3, where the calling contexts from Example 21 are abbreviated to improve readability. The lack of the loop on cc_1 does not prevent the graph to be considered a CCG because there exist no tuples $(a,b),(c,d) \in [\mathbb{N} \times \mathbb{N}]$ such that $\varepsilon_\pm(\texttt{ack})(C_{cc_1},(a,b)) \wedge \varepsilon(\texttt{ack})(e_{cc_1},(a,b),(c,d)) \wedge \varepsilon_\pm(\texttt{ack})(C_{cc_2},(c,d))$, since this formula can be expanded to $(a \neq 0 \wedge b = 0) \wedge (c = a-1 \wedge d = 1) \wedge (c \neq 0 \wedge d = 0)$.

The following standard notions from Graph Theory will be used in the definitions below. A *walk* of $G_\texttt{p}$ is a sequence $cc_{i_1}, \ldots, cc_{i_n}$ of calling contexts such that for all $1 \leq j < n$ there is an edge between cc_{i_j} and $cc_{i_{j+1}}$. The collection of all walks of a given graph G is denoted by $\mathbf{Walk}_G$. A *circuit* is a walk $cc_{i_1}, \ldots, cc_{i_n}$, with $n > 1$, where $cc_{i_1} = cc_{i_n}$. A *cycle* is an elementary circuit, i.e., a circuit $cc_{i_1}, \ldots, cc_{i_n}$ where the only repeating nodes are cc_{i_1} and cc_{i_n}. The notation $|\mathbf{w}|$ will be used in the following to denote the length of a walk $\mathbf{w}$ and $|G|$ to denote the size of a graph G. Additionally, if $\mathbf{w} = cc_1, \cdots, cc_n$ the expression $\mathbf{w}[a..b]$ will denote the walk $cc_a, \cdots, cc_b$ when $1 \leq a \leq b \leq n$.

▶ **Definition 33.** *Let $\mathcal{M}$ be a family of N measures $\mu_k : \mathcal{V}al \to M$, with $1 \leq k \leq N$, and $<$ be a well-founded relation over M. A* measure combination *of a sequence of calling contexts $cc_{i_1}, \ldots, cc_{i_n}$ is a sequence of natural numbers $k_1, \ldots, k_n$, with $1 \leq k_j \leq N$ representing measure μ_{k_j}, such that for all $1 \leq j < n$, $v, v' \in \mathcal{V}al$, $\varepsilon_\pm(\texttt{p})(C_j, v) \wedge \varepsilon(\texttt{p})(e_j, v, v')$ implies $\mu_{k_j}(v) \rhd_j \mu_{k_{j+1}}(v')$, where $cc_{i_j} = (\texttt{rec}(e_j), P_j, C_j)$ and $\rhd_j \in \{>, \geq\}$. A measure combination is* descending *if at least one $\rhd_j$ is $>$.*

▶ **Definition 34.** *Let $G_\texttt{p}$ be a CCG of a PVS0 program $\texttt{p} \in \texttt{PVS0}_{\mathcal{V}al}$ and let $\mathcal{M}$ be a family of measures for a well-founded relation $<$ over a type M. The graph $G_\texttt{p}$ is said to be* CCG terminating *(denoted by $T_{CCG}(G_\texttt{p})$) if for all circuits $cc_{i_1}, \ldots, cc_{i_n}$ in $\mathbf{Walk}_{G_\texttt{p}}$ there is a* descending *measure combination $k_1, \ldots, k_n$, with $k_1 = k_n$.*

▶ **Theorem 35.** *For all $\texttt{p} \in \texttt{PVS0}_{\mathcal{V}al}$, $T_{SCP}(\texttt{p})$ if and only if $T_{CCG}(G_\texttt{p})$ for some CCG $G_\texttt{p}$ of $\texttt{p}$ and some family of measures $\mathcal{M}$.*

Since the number of circuits in a CCG is potentially infinite, CCG termination does not directly provide an effective procedure to check termination. Even though the number of cycles in a graph is indeed finite, it is not enough to check for decreasing measure combinations in cycles (see [3] for details).

5 Matrix-Weighted Graphs

Matrix-Weighted Graphs is a technique to check for descending measure combinations in a CCG using an algebra over matrices [3]. Let $\mathcal{M}$ be a family of N measures, every edge in the CCG is labeled with a matrix of dimension $N \times N$ and values in $\{-1, 0, 1\}$. The type of these matrices will be denoted by $\mathbb{M}_\mathbf{3}^N$.

$$\begin{tikzcd}
 & cc_1 \arrow[dl, bend right, "\mathbf{M}_1"'] \arrow[dr, bend left, "\mathbf{M}_1"] & \\
cc_2 \arrow[loop left, "\mathbf{M}_2"] \arrow[ur, "\mathbf{M}_2"'] \arrow[rr, bend left, "\mathbf{M}_2"] & & cc_3 \arrow[loop right, "\mathbf{M}_3"] \arrow[ul, "\mathbf{M}_3"] \arrow[ll, bend left, "\mathbf{M}_3"]
\end{tikzcd}
\qquad
\mathbf{M}_1 = \begin{bmatrix} 1 & 0 \\ -1 & -1 \end{bmatrix} \; \mathbf{M}_2 = \begin{bmatrix} 1 & -1 \\ -1 & -1 \end{bmatrix} \; \mathbf{M}_3 = \begin{bmatrix} 0 & -1 \\ -1 & 1 \end{bmatrix}$$

Figure 4 A MWG for the $\mathtt{p}$ program for the Ackermann function, where the family of measures $\mathcal{M}$ is composed by $\mu_1(m,n) = m$ and $\mu_2(m,n) = n$.

▶ **Definition 36** (Matrix Weighted Graph). *Let $\mathtt{p}$ be a* PVS0 *program in $PVS0_{\mathcal{V}al}$ and $\mathcal{M}$ be a family of N measures $\{\mu_i\}_{i=1}^N$. A* matrix-weighted graph *$W_{\mathtt{p}}^{\mathcal{M}}$ of $\mathtt{p}$ is a CCG $G_{\mathtt{p}} = (V, E)$ of $\mathtt{p}$ whose edges are* correctly *labeled by matrices in $\mathbb{M}_{\mathbf{3}}^N$.*

An edge $(cc_a, cc_b) \in E$ is said to be correctly labeled *by a matrix $\mathbf{M}_{ab}$ when for all $1 \le i, j \le N$,*

- *if $\mathbf{M}_{ab}(i,j) = 1$, for all $v_a, v_b \in \mathcal{V}al$, $\varepsilon_{\pm}(\mathtt{p})(C_a, v_a) \wedge \varepsilon(\mathtt{p})(e_a, v_a, v_b)$ implies $\mu_i(v_a) > \mu_j(v_b)$.*
- *if $\mathbf{M}_{ab}(i,j) = 0$, for all $v_a, v_b \in \mathcal{V}al$, $\varepsilon_{\pm}(\mathtt{p})(C_a, v_a) \wedge \varepsilon(\mathtt{p})(e_a, v_a, v_b)$ implies $\mu_i(v_a) \ge \mu_j(v_b)$.*

An entry $\mathbf{M}_{ab}(i,j) = -1$ provides no information about $v_a, v_b \in \mathcal{V}al$ with respect to μ_i and μ_j.

The Figure 4 depicts a possible MWG for the $\mathtt{p}$ program implementing the Ackermann function.

The algebra of matrices used to define the notion of MWG termination is given by the following operations. Multiplication of matrices with values in $\{-1, 0, 1\}$ is defined as usual, where addition and multiplication of such values is defined below. Let $x, y \in \{-1, 0, 1\}$,

$$x \times y = \begin{cases} -1 & \text{if } \min(x,y) = -1, \\ 1 & \text{if } \min(x,y) \ge 0 \wedge \max(x,y) = 1, \\ 0 & \text{otherwise,} \end{cases} \qquad\qquad x + y = \max(x,y).$$

▶ **Definition 37** (Weight of a Walk). *Let $\mathtt{p}$ be a PVS0 program, $W_{\mathtt{p}}$ a MWG for $\mathtt{p}$, and $\mathbf{w_i} = cc_{i_1}, \ldots, cc_{i_n}$ a walk in such graph, the* weight *of $\mathbf{w_i}$, noted by $w(\mathbf{w_i})$, is defined as $\Pi_{j=1}^{n-1} \mathbf{M}_{i_j i_{j+1}}$. A weight $w(\mathbf{w_i})$ is* positive *if there exists $1 \le i \le N$ such that $w(\mathbf{w_i})(i,i) > 0$.*

▶ **Example 38.** Continuing the example in Figure 4, the weights for walks $\mathbf{w_{1,3}} = cc_1, cc_3$ and $\mathbf{w_{2,3}} = cc_2, cc_3$ are shown below. Both of them are positive.

$$w(\mathbf{w_{1,3}}) = \begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix} \qquad w(\mathbf{w_{2,3}}) = \begin{bmatrix} 1 & -1 \\ -1 & -1 \end{bmatrix}$$

The lemma below states a useful property about walk weights.

▶ **Lemma 39.** *Let $W_{\mathtt{p}}$ be an MWG for a PVS0 program $\mathtt{p}$ and $\mathbf{w} = cc_1, \cdots, cc_n$ be a walk of $W_{\mathtt{p}}$, then $w(\mathbf{w}) = w(cc_1, \cdots, cc_i) \times w(cc_i, \cdots, cc_n)$.*

As in the case of the calling context graphs, a walk in a MWG represents a trace of recursive calls. Hence, circuit denotes a trace ending at the same recursive call where it starts. In line with the notion of CCG termination, a MWG is considered *terminating* when, for every possible circuit, the matrix representing its weight has at least one positive value in its diagonal.

▶ **Definition 40** (Matrix-Weighted Graph Termination). *Let $\mathtt{p}$ a PVS0 program and let $W_{\mathtt{p}}$ be a MWG of $\mathtt{p}$. The graph $W_{\mathtt{p}}$ is said to be* MWG terminating *(denoted by $T_{MWG}(W_{\mathtt{p}})$) when for every circuit $\mathbf{w_i}$ of $W_{\mathtt{p}}$, $w(\mathbf{w_i})$ is positive.*

The equivalence between the notions of termination for CCG and MWG is stated by Theorem 41 below.

▶ **Theorem 41.** *Let $\mathcal{M}$ be a family of N measures for a well-founded relation $<$ over a type M. For all $\mathtt{p} \in \mathtt{PVS0}_{\mathcal{V}al}$, $T_{CCG}(C^{\mathcal{M}}_{\mathtt{p}})$ for some CCG $C^{\mathcal{M}}_{\mathtt{p}}$ if and only if $T_{MWG}(W^{\mathcal{M}}_{\mathtt{p}})$ for some MWG $W^{\mathcal{M}}_{\mathtt{p}}$.*

Proof. This theorem follows from the fact that circuits in $W_{\mathtt{p}}$, built from $G_{\mathtt{p}}$ using the same measures, have positive weights if and only if there exist corresponding descending measure combinations. This property is proved by induction in the length of circuits in $G_{\mathtt{p}}$. ◀

As pointed out in the previous section, a digraph such as any CCG or MWG can have infinitely many circuits. Nevertheless, since the information used to check MWG termination is the weight of the circuits and, for a fixed number N of measures, there are only finitely many possible weights, a bound on the length of the circuits to be considered can be safely imposed as shown in the lemma below.

▶ **Lemma 42.** *Let $\mathtt{p}$ be a PVS0 program and $W_{\mathtt{p}}$ a MWG for it. If for all circuit $\mathbf{w}$ in $W_{\mathtt{p}}$ such that $|\mathbf{w}| \leq |W_{\mathtt{p}}| \cdot 3^{N^2} + 1$, $w(\mathbf{w})$ is positive, then $W_{\mathtt{p}}$ is MWG terminating.*

Proof. In order to prove $T_{MWG}(W_{\mathtt{p}})$, it is necessary to show that every circuit of $W_{\mathtt{p}}$ has positive weight. For every circuit $\mathbf{w} = cc_1, \cdots, cc_n$ of $W_{\mathtt{p}}$, if $n \leq |W_{\mathtt{p}}| \cdot 3^{N^2} + 1$, then $w(\mathbf{w})$ is positive by hypothesis. Otherwise, it can be proved that there exists another circuit $\mathbf{w}'$ such that $w(\mathbf{w}) = w(\mathbf{w}')$ and $|\mathbf{w}'| \leq |W_{\mathtt{p}}| \cdot 3^{N^2} + 1$. Hence, by hypothesis, $w(\mathbf{w})'$ is positive and then $w(\mathbf{w})$ is positive too.

The existence of the circuit $\mathbf{w}'$ can be shown by constructing a sequence of pairs $\langle (cc_i, w(cc_1, \cdots, cc_i)) \rangle_{i=1}^{n}$, where for each $1 \leq i \leq n$, the vertex cc_i is the i^{th} vertex in $\mathbf{w}$ and it is paired with the weight of the prefix of $\mathbf{w}$ of length i. By a simple counting argument, it can be seen that there cannot exist more than $|W_{\mathtt{p}}| \cdot 3^{N^2}$ of these pairs. Since $n > |W_{\mathtt{p}}| \cdot 3^{N^2} + 1$, there are two indices i, j such that $(cc_i, w(cc_1, \cdots, cc_i)) = (cc_j, w(cc_1, \cdots, cc_j))$ and $i \neq j$. Without loss of generality, it can be assumed that $i < j$. Then, the walk $\mathbf{w}'' = cc_1, \cdots, cc_{i-1}, cc_j, cc_{j+1}, \cdots, cc_n$ is a circuit, since $cc_i = cc_j$ and $cc_1 = cc_n$, and it is shorter than $\mathbf{w}$. To calculate the length of $\mathbf{w}''$, first it should be noted that, by Lemma 39, $w(cc_1, \cdots, cc_i, cc_{j+1}, \cdots, cc_n) = w(cc_1, \cdots, cc_{i-1}, cc_j) \times w(cc_j, cc_{j+1}, \cdots, cc_n)$. Since $cc_i = cc_j$ and $w(cc_1, \cdots, cc_i) = w(cc_1, \cdots, cc_j)$, $w(\mathbf{w}'') = w(cc_1, \cdots, cc_j) \times w(cc_j, cc_{j+1}, \cdots, cc_n)$, which by Lemma 39 again is equal to $w(\mathbf{w})$.

If the length of $\mathbf{w}''$ is at most $|W_{\mathtt{p}}| \cdot 3^{N^2} + 1$, it can be taken to be $\mathbf{w}'$. Otherwise, the same procedure can be repeated to shorten the circuit even further. Since this procedure removes at least one vertex each time, eventually a circuit shorter than $|W_{\mathtt{p}}| \cdot 3^{N^2} + 1$ and with the same weight than $\mathbf{w}$ will be obtained. ◀

Lemma 42 allows for the definition of a procedure to check termination on a matrix-weighted graph. This procedure is referred to as Dutle's procedure. Given a MWG $W^{\mathcal{M}}_{\mathtt{p}} = (V, E)$ on a family of N measures $\mathcal{M}$ for a PVS0 program $\mathtt{p}$, the general idea of this procedure is to build sequentially a family of functions $f_i : V \to \mathbf{list}[\mathbb{M}^N_{\mathbf{3}}]$ with $1 \leq i \leq |W_{\mathtt{p}}| \cdot 3^{N^2} + 1$. These functions are such that for each vertex $cc \in V$ and every circuit $\mathbf{w}$ in $W^{\mathcal{M}}_{\mathtt{p}}$ starting at cc and $|\mathbf{w}| <= i$, there is a weight $\mathbf{M} \in f_i(cc)$ for which $\mathbf{M} \leq w(\mathbf{w})$. If for some i there

```
terminating?($W_p$: MWG): bool =
  LET $f_1 \leftarrow$ expandWeightLists($W_p, \lambda(v : V_{W_p}) :$ null)
  IN terminatingAt?($W_p$, 1, $f_1$)

terminatingAt?($W_p$: MWG, $i$ : $\mathbb{N}$, $f_i$ : [$V_{W_p} \to$ list[$\mathbb{M}_3^N$]]): bool =
 $i \geq |W_p| \cdot 3^{N^2} + 1$ OR
 LET $f_{i+1} \leftarrow$ expandWeightLists($W_p$, $f_i$) IN
 IF $\exists$ ($cc \in V_{W_p}, \mathbf{M} \in f_{i+1}(cc)) : \neg$ positive?($\mathbf{M}$) THEN FALSE
 ELSE $f_i$ = $f_{i+1}$ OR terminatingAt?($W_p$, $i+1$, $f_{i+1}$) ENDIF

expandWeightLists($W_p$: MWG, $f_i$ : [$V_{W_p} \to$ list[$\mathbb{M}_3^N$]]): [$V_{W_p} \to$ list[$\mathbb{M}_3^N$]] =
 $\lambda(v : V_{W_p})$: map(expandPartialWeight($f_i$), allCyclesAt($W_p$,$v$))

expandPartialWeight($f_i$ : [$V_{W_p} \to$ list[$\mathbb{M}_3^N$]]): [Walk$_{W_p} \to$ list[$\mathbb{M}_3^N$]] =
 $\lambda$(w : Walk$_{W_p}$):
   LET $l$ $\leftarrow$ cons(id$_\times$, $f_i$(w[0]))
   IN IF |w| = 1 THEN $l$
      ELSE LET $l_1 \leftarrow$ map($\lambda$ ($\mathbf{M}$ : $\mathbb{M}_3^N$) : $\mathbf{M}$ * $w$(w[0..1]))($l$),
               $l_2 \leftarrow$ expandPartialWeight(w[1 .. |w| − 1], $f_i$)
         IN pairwiseMultiplication($l_1$,$l_2$) ENDIF
```

Figure 5 Dutle's procedure to check termination on matrix-weighted graphs.

is vertex cc and a weight $\mathbf{M}$ such that $\mathbf{M} \in f_i(cc)$ and $\mathbf{M}$ is not positive, then it can be concluded that $W_p^{\mathcal{M}}$ is not terminating, since there is a circuit whose weight is not positive. On the contrary, if the algorithm reaches the point where $i = |W_p| \cdot 3^{N^2} + 1$ with positive matrices in the range of $f_i(cc)$ for each i, $W_p^{\mathcal{M}}$ can be safely declared as terminating thanks to Lemma 42.

Figure 5 depicts a pseudocode for Dutle's procedure. The function **terminatingAt**? implements the rough idea described in the previous paragraph. The auxiliary function **expandWeightLists** computes f_{i+1} given its predecessor f_i. Hence, for instance, f_1 contains lower bounds for the weight of each cycle in the graph W_p. Starting from there, in every recursive call to **terminatingAt**?, for each vertex cc in W_p, $f_{i+1}(cc)$ grows with respect to $f_i(cc)$ by incorporating lower bounds for the circuits passing through cc that are longer that the ones considered in $f_i(cc)$ by a complete cycle each. Then, f_i provides information about a lower bound on each walk of length at most i as previously stated, but it also contains information about longer circuits. Hence, a guard that checks saturation of such functions ($f_{i+1} = f_i$) is also included to prematurely end the recursion if possible.

In the pseudocode, **cons**(x, l) denotes the list constructed from the element x and the list l, **null** denotes the empty list, and **map**(f, l) is used to denote the list formed by the application of the function f to each element in l. Furthermore, **positive**?($\mathbf{M}$) checks if a matrix $\mathbf{M}$ is positive in the sense of Definition 37, **allCyclesAt**(G, v) returns the list of all the cycles in the graph G passing through node v (if any), **id**$_\times$ denotes the matrix weight that acts as multiplicative identity, and **pairwiseMultiplication**(l_1, l_2) is the funtion that given two lists l_1, l_2 of matrices in $\mathbb{M}_3^N$ returns the list resulting from the pairwise multiplication of the elements in those lists.

Dutle's Procedure is a sound and complete procedure to decide positive weight of all circuits in a matrix-weighted graph and hence to check termination on MWG. This procedure has been formally verified in PVS as part of this work. The performance of the procedure can be improved in both execution time and used storage space. For example, the function **expandWeightLists** keeps enlarging the lists on the range of each f_{i+1} (with respect to its predecessor f_i), while it is enough to keep such lists minimal, for instance by adding a new weight $\mathbf{M}$ to a list l only if there are no $\mathbf{M}'$ in l already such that $\mathbf{M}' \leq \mathbf{M}$.

The notion of Matrix Weighted Termination can be used to define a procedure to automatically prove termination of certain recursive functions in PVS. Such a procedure consist of the steps described below.

1. Extract the calling contexts from the PVS program definition. The set of calling contexts is finite and can be extracted from the program by syntactic analysis.
2. Generate a sound CCG for the program.
   - A fully connected CCG is *sound* (the more edges the more inefficient the method).
   - The theorem prover itself can be used to *soundly* remove edges from the graph, i.e., an edge cc_a, cc_b can be removed if $\vdash \forall (v_a, v_b : \mathcal{V}al) : \varepsilon_{\pm}(\mathtt{p})(C_a, v_a) \ \wedge \ \varepsilon(\mathtt{p})(e_a, v_a, v_b) \ \Rightarrow \neg\, \varepsilon_{\pm}(\mathtt{p})(C_b, v_b)$ can be discharged.
   - In order to select measures to form the family $\mathcal{M}$, the following heuristics can be used.
     - The order relation $<$ over natural numbers is usually a good starting point.
     - Since *CCG* allows for a family of measures, it is *sound* to add as many measures as possible (of course the more measures the more inefficient the method).
     - Predefined functions can be used, e.g., parameter projections (in the case of natural numbers), natural size of parameters (in the case of data types), maximum/minimum of parameters, etc. More complex recursions may need heuristics based on static analysis.
3. Construct a MWG for the program based on the CCG defined in the previous step in the following way: all edges starting in a given calling context cc_a can be labeled with the same matrix $\mathbf{M}_a$. It is *sound* to set all its entries to -1. The theorem prover can then be used to *soundly* set the entries in $\mathbf{M}_a(i, j)$ to either 0 or 1 as follows,
   - If $\vdash \ \forall (v_a, v_b : \mathcal{V}al) : \varepsilon_{\pm}(\mathtt{p})(C_a, v_a) \ \wedge \varepsilon(\mathtt{p})(e_a, v_a, v_b) \ \Rightarrow \ \mu_i(v_a) \ > \ \mu_j(v_b)$ can be proved, set $M_a(i, j)$ to 1.
   - If $\vdash \forall (v_a, v_b : \mathcal{V}al) : \varepsilon_{\pm}(\mathtt{p})(C_a, v_a) \ \wedge \varepsilon(\mathtt{p})(e_a, v_a, v_b) \ \Rightarrow \ \mu_i(v_a) \ \geq \ \mu_j(v_b)$ can be proved, set $M_a(i, j)$ to 0.
4. Use Dutle's procedure to check termination on the MWG.

6 Conclusion, Related and Future Work

The termination of programs expressed in a language such as `PVS0` can be guaranteed by providing a measure on a well-founded relation that strictly decreases at every recursive call. This criterion can be traced back to Turing [14]. A related practical approach was further proposed by Floyd [6]. The inputs and outputs of program instructions are enriched with assertions (Floyd-Hoare first-order well-known pre- and post-conditions) so that if the pre-condition holds and the instruction is executed the post-condition must hold. To verify termination, these assertions are enriched with decreasing assertions that are built using a well-founded ordering according to some *measure* function on the inputs and outputs of the program. This approach can also be used in recursive functions as shown by Boyer and Moore [5]. In this case, a measure is provided over the arguments of the function. The measure must strictly decrease at every possible recursive call. The conditions to effectively check if a recursive call is possible or not are statically given by the guards of branching instructions that lead to the function call. In the case of PVS, as in many other proof assistants, the user provides a measure function and a well-founded relation for each recursive function. The necessary conditions that guarantee termination are built during type checking. In this paper, these conditions are referred to as *termination TCCs* and the process that generates termination TCCs for PVS0 is formally verified against other termination criteria.

The functional language Agda tries to automatically check termination of programs by finding a lexicographic order on the parameters of the functions participating in the recursive-call chain [1]. This technique operates on multi-graphs whose edges are labeled with matrices, but they differ from the graphs and matrices used in this paper in several aspects. In that paper, each node represents a function instead of a calling context, each edge represents a call, and the matrices labeling the edges relate the arguments used in each call under the same order relation, instead of different measures as in the technique presented in this paper. Closer to the work in this paper, Krauss formalizes the size-change termination principle in Isabelle/HOL [8]. He also developed a technology based on this principle and the dependency pair criterion to verify the termination of a class of recursive functions specified in Isabelle/HOL. CCGs are implemented in ACL2s by Manolios and Vroon, where they report that "[CCG] was able to automatically prove termination for over *98%* of the more than 10,000 functions in the regression suite [of ACL2s]" [10]. In his PhD thesis, Vroon provides a pencil and paper proof of the correctness of his method based on CCGs [15].

The formalization presented in this paper includes proofs of equivalence among several termination criteria. Other related formalizations that use or connect to the one presented in this paper have been previously presented. For example, Alves Almeida and Ayala-Rincón formalized a notion of termination for term rewriting systems based on dependency pairs and showed how it can be related to the notions explained in this paper [2]. Also, Ferreira Ramos et. al. have presented a proof of termination undecidability constructed on the model language `PVS0` [12]. The Matrix Weighted Graphs algebraic approach, which is an implementation of the CCG technique, was first presented in Avelar's PhD along with its formalization in PVS [3]. That formalization does not include Dutle's procedure. The authors are currently working on the implementation of proof strategies, based on computational reflection, that use the CCG/MWG technique to automate termination proofs of PVS recursive functions.

7 Bibliography

References

1 Andreas Abel. foetus–termination checker for simple functional programs. Programming Lab Report 474, LMU München, 1998. URL: `http://www.cse.chalmers.se/~abela/foetus/`.

2 Ariane Alves Almeida and Mauricio Ayala-Rincón. Formalizing the dependency pair criterion for innermost termination. *Sci. Comput. Program.*, 195:102474, 2020. `doi:10.1016/j.scico.2020.102474`.

3 Andréia B. Avelar. *Formalização da automação da terminação através de grafos com matrizes de medida*. PhD thesis, Universidade de Brasília, Departamento de Matemática, Brasília, Distrito Federal, Brasil, 2015. In Portuguese.

4 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. Springer-Verlag Berlin Heidelberg, 2004. `doi:10.1007/978-3-662-07964-5`.

5 Robert S. Boyer and J Strother Moore. *A Computational Logic*. Academic Press, 1979.

6 Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

7 Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

8 Alexander Krauss. Certified size-change termination. In Frank Pfenning, editor, *Automated Deduction – CADE-21*, pages 460–475, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

9 Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 81–92, 2001. `doi:10.1145/360204.360210`.

10 Panagiotis Manolios and Daron Vroon. Termination analysis with calling context graphs. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification*, pages 401–414, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

11 Sam Owre, John Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *Proceedings of CADE 1992*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer, 1992.

12 Thiago Mendonça Ferreira Ramos, César Muñoz, Mauricio Ayala-Rincón, Mariano Moscato, Aaron Dutle, and Anthony Narkawicz. Formalization of the undecidability of the halting problem for a functional language. In Lawrence S. Moss, Ruy de Queiroz, and Maricarmen Martinez, editors, *Logic, Language, Information, and Computation*, pages 196–209, Berlin, Heidelberg, 2018. Springer Berlin Heidelberg.

13 Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. of the London Mathematical Society*, 42(1):230–265, 1937.

14 Alan Turing. Checking a large routine. In *Report of a Conference High Speed Automatic Calculating-Machines*, pages 67–69. University Mathematical Laboratory, 1949.

15 Daron Vroon. *Automatically Proving the Termination of Functional Programs*. PhD thesis, Georgia Institute of Technology, 2007.

Verified Double Sided Auctions for Financial Markets

Raja Natarajan
Tata Institute of Fundamental Research, Mumbai, India

Suneel Sarswat
Tata Institute of Fundamental Research, Mumbai, India

Abhishek Kr Singh
Birla Institute of Technology and Science Pilani, Goa, India

Abstract

Double sided auctions are widely used in financial markets to match demand and supply. Prior works on double sided auctions have focused primarily on single quantity trade requests. We extend various notions of double sided auctions to incorporate multiple quantity trade requests and provide fully formalized matching algorithms for double sided auctions with their correctness proofs. We establish new uniqueness theorems that enable automatic detection of violations in an exchange program by comparing its output with that of a verified program. All proofs are formalized in the Coq proof assistant without adding any axiom to the system. We extract verified OCaml and Haskell programs that can be used by the exchanges and the regulators of the financial markets. We demonstrate the practical applicability of our work by running the verified program on real market data from an exchange to automatically check for violations in the exchange algorithm.

2012 ACM Subject Classification Information systems → Online auctions; Software and its engineering → Formal software verification; Theory of computation → Algorithmic mechanism design; Theory of computation → Computational pricing and auctions; Theory of computation → Program verification; Theory of computation → Automated reasoning

Keywords and phrases Double Sided Auction, Formal Verification, Financial Markets, Proof Assistant

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.28

Related Version *Full Version*: `https://arxiv.org/abs/2104.08437` [5]

Supplementary Material *Software*: `https://github.com/suneel-sarswat/dsam`
archived at `swh:1:dir:94b9da19e3cfdb243bd47f1619a2901758fb1243`

Acknowledgements We wish to thank Mohit Garg for his generous contribution to this work.

1 Introduction

Computer algorithms are routinely deployed nowadays by all big stock exchanges to match buy and sell requests. These algorithms are required to abide by various regulatory guidelines. For example, market regulators make it mandatory for trades resulting from double sided auctions at exchanges to be fair, uniform and individual-rational.

In this paper, we introduce a formal framework for analyzing trades resulting from double sided auctions used in the financial markets. To verify the essential properties required by market regulators, we formally define these notions in a theorem prover and then develop important results about matching demand and supply. Finally, we use this framework to verify properties of two important classes of double sided auction mechanisms.

One of the resulting advantages of our work for an exchange or a regulator is that they can check the algorithms deployed for any violations from required properties automatically. This is enabled by the new uniqueness results that we establish in this work. All the definitions and results presented in this paper are completely formalized in the Coq proof assistant without adding any additional axioms to it. The complete formalization in Coq facilitates automatic program extraction in OCaml and Haskell, with the guarantee that extracted programs satisfy the requirements specified by the market regulator. Consequently, the extracted program could also be deployed directly at an exchange, apart from checking for violations in existing programs. We demonstrate the practical applicability of our work by running the verified program on real market data from an exchange to automatically check for violations in the exchange algorithm.

The rest of this paper is organized as follows: Section 2 provides a brief background and overview of trading at exchanges which is needed to describe our contributions; In Section 3, we briefly state our contributions; Section 4 provides basic definitions and establishes certain combinatorial results; Section 5 describes a fairness procedure; Section 6 describes the uniform matching mechanism used in the financial markets; Section 7 describes the maximum matching mechanism; Section 8 establishes uniqueness results that enables automatic checking for violations in an exchange matching algorithm; Section 9 describes the practical utility of our work through running our verified program on real market data from an exchange; Section 10 concludes the paper with related work and future directions. Parts of some sections which could not be fully accommodated due to space constraints have been moved to the appendix of the full version of the paper [5].

2 Background

Financial trades occur at various types of exchanges. For example, there are exchanges for stocks, commodities and currencies. At any exchange, multiple buyers and sellers participate to trade certain products. Mostly exchanges employ double sided auction mechanisms to match the buyers and sellers. Some exchanges, apart from using double sided auctions, also use an online continuous algorithm for executing trades during certain time intervals, especially for highly traded products.

For conducting trades of a certain product using a double sided auction mechanism, the exchange collects buy and sell requests from the traders for a fixed time period. At the end of this time period, the exchange matches some of the trade requests and outputs trades, all at a single price. This price is sometimes referred to as the equilibrium price and the process as price discovery. A buyer places a buy request, also known as a *bid*, which consists of a quantity indicating the maximum number of units he is interested in buying and a common maximum price (bid's limit price) for each of the units. Similarly, a seller's sell request, an *ask*, consists of a quantity and a minimum price (ask's limit price). Each trade (*transaction*) consists of a bid, an ask, traded quantity, and a trade price. Naturally, the traded quantity should be at most the minimum of the bid and the ask quantities and the trade price should be compatible with the bid and the ask.

Apart from the single price property and compatibility constraint mentioned above, there are other desired properties that the trades (*matching*) should have. The properties that capture these constraints are: *uniform*, *individual-rational*, *fair* and *maximum*. We briefly describe these matching properties:

- **Uniform**: A matching is uniform if all the trades happen at the same price.
- **Individual-rational**: A matching is individual-rational if for each matched bid-ask pair the trade price is between the bid and ask limit prices. In the context of financial markets, the trade price should always be between the limit prices of the matched bid-ask pair.
- **Fair**: A bid b_1 is more competitive than a bid b_2 if b_1 has a higher limit price than b_2 or if their limit prices are the same and b_1 arrives earlier than b_2. Similarly, we can define competitiveness between two asks. A matching is *unfair* if a less competitive bid gets

matched but a more competitive bid is not fully matched. Similarly, it could be unfair if a more competitive ask is not fully matched. If a matching is not unfair, then it is fair.
- **Maximum**: A matching is maximum if it has the maximum possible total traded quantity among all possible matchings.
- **Optimal individual-rational-uniform**: An individual-rational and uniform matching is called optimal individual-rational-uniform if it has the largest total trade volume among all matchings that are individual-rational and uniform.

No single algorithm can possess all the above first four properties simultaneously [12, 4]. In the context of financial markets, regulators insist on the matching being fair and optimal individual-rational-uniform, thus compromising on the maximum property. In other contexts where the matching being maximum is important along with individual rational and fair, uniformity is lost. This gives rise to two different classes of double sided auction mechanisms, each with a different objective. In our work, we consider both these classes of mechanisms.

3 Our Contributions

In this work, we formalize the notion of double sided auctions where trade requests can be of multiple quantities. Prior to our work, similar notions were explicitly defined only for single unit trade requests [8, 6, 13]. In going from formalizing the theory for single unit to the theory of multiple units, the mechanisms and their correctness proofs changed substantially. Due to the possibility of partial trades, the formal analysis of multiple unit trades becomes significantly more involved than in [8]. In this work, we show how to efficiently handle this extra complexity by making the functions and their properties sensitive to the partial trade quantities. This helps us to develop formal proofs of correctness of the recursive mechanisms for double sided auctions.

In addition, we provide new uniqueness results that guarantee that the matching algorithm for the double sided auctions used in the financial markets outputs a unique volume of trades per order if the algorithm is fair and optimal individual-rational-uniform; thus enabling automatic checking of violations in the exchange algorithm by comparing its output with that of a verified program. We demonstrate this by running the extracted OCaml code of our certified mechanism on real data from an exchange and comparing the outputs. Following is a brief description of the key results formalized in this work.

- **Combinatorial result**: We show that the modeling and the libraries we created to obtain our results are also useful in proving other important results on double sided auctions. For example, in Theorem 7, we show that for any p, no matching can achieve a trade volume higher than the sum of the total demand and the total supply in the market at price p.
- **Fairness**: We show that any matching can be converted into a fair matching without compromising on the total traded volume. For this, we design an algorithm, the Fair procedure, which takes a matching M as input, and outputs a matching M'. In Theorem 15, we show that the total traded quantities of M and M' are the same and M' is a fair matching.
- **Uniform mechanism**: We design an algorithm, the UM procedure, that takes as input the bids and the asks and outputs a fair, individual-rational and uniform matching. Furthermore, in Theorem 20, we show that the output matching has the largest total trade volume among all the matchings that are uniform and individual-rational and thus is optimal individual-rational-uniform. This algorithm is used in the exchanges that output trades using double sided auctions.

- **Maximum mechanism**: We design an algorithm, the MM procedure, that takes as input the bids and the asks and outputs an individual-rational, fair and maximum matching. In Theorem 21, we show that the output matching has the largest total trade volume among all the matchings that are individual-rational.
- **Uniqueness theorems**: For any two fair and optimal individual-rational-uniform matchings, Theorem 23 implies that their total trade volume for each order is the same. Thus, if we compare the trade volumes between an exchange's matching output with our verified program's output and for some order they do not match, then the exchange's matching is not fair and optimal individual-rational-uniform. On the contrary, if for each order, the trade volumes match, then Theorem 24 implies that the exchange's matching is also fair and optimal individual-rational-uniform (given that it already is individual-rational and uniform, which can be easily verified by checking the trade prices). Making use of these results, in Section 9, we check violations automatically in real data from an exchange.

The Coq code together with the extracted OCaml and Haskell programs for all the above results is available at [10]. Our Coq formalization consists of approximately 50 new definitions, 750 lemmas and theorems and 12000 lines of code. In the following sections, we provide definitions, procedures and proof sketches that closely follow our actual formalization.

4 Modeling Double Sided Auctions

In a double sided auction multiple buyers and sellers place their orders to buy or sell an underlying product. The auctioneer matches these buy-sell requests based on their *limit prices*, *arrival time*, and the maximum specified *trade quantities*. Note that the limit prices are natural numbers when expressed in the monetary unit of the lowest denomination (like cents in USA). In our presentation, we will be working with lists (of bids, asks and transactions); For ease of readability, we will often use set-theoretic notations like $\in$, $\subseteq$, $\supseteq$, $\emptyset$ on lists whose meanings are easy to guess from the context.

▶ **Definition 1** (Bid). *A bid $b = (id_b, \tau_b, q_b, p_b)$ represents a buy request having four components. Here, the first two components id_b and τ_b are the unique identifier and the timestamp assigned to the buy request b, respectively, whereas the third component q_b represents the quota of b, the maximum quantity of the item the buyer is willing to buy. The last component p_b is the limit price of the buy request, which is the price above which the buyer does not want to buy the item.*

▶ **Definition 2** (Ask). *An ask $a = (id_a, \tau_a, q_a, p_a)$ represents a sell request having four components. Here, the first two components id_a and τ_a are the unique identifier and the timestamp assigned to the sell request a, respectively, whereas the third component q_a represents the quota of a, the maximum quantity of the item the seller is willing to sell. The last component p_a is the limit price of the sell request, which is the price below which the seller does not want to sell the item.*

We say that a bid $b \in B$ is matchable with an ask $a \in A$ if $p_a \leq p_b$.

In a double sided auction, the auctioneer is presented with duplicate-free[1] lists of buy and sell requests (lists B and A, respectively). The auctioneer can match a bid $b \in B$ with an ask

[1] A list of bids or asks is duplicate-free if all the participating orders have distinct ids.

$a \in A$ only if $p_b \geq p_a$. Furthermore, the auctioneer assigns a trade price and a trade quantity to each matched bid-ask pair, which finally results in a *transaction* m. Therefore, we can represent a matching of demand and supply by using a list whose entries are *transactions*.

▶ **Definition 3** (Transaction). *A transaction* $m = (b_m, a_m, q_m, p_m)$ *describes a trade between the bid* b_m *and the ask* a_m. *The next two components* q_m *and* p_m *are the traded quantity and the trade price, respectively. For ease of readability, we use the terms* $p(b_m)$, $p(a_m)$, $q(b_m)$, *and* $q(a_m)$ *for* p_{b_m}, p_{a_m}, q_{b_m} *and* q_{a_m}, *respectively.*

▶ **Definition 4** (Matching M B A). *A list of transactions* M *is a matching between the duplicate-free lists of bids* B *and asks* A *if*

1. *For each transaction* $m \in M$, *the bid of* m *is matchable with the ask of* m *(i.e.,* $p(a_m) \leq p(b_m)$*).*
2. *The list of bids present in* M, *denoted by* B_M, *is a subset of* B *(i.e.,* $B_M \subseteq B$*).*
3. *The list of asks present in* M, *denoted by* A_M, *is a subset of* A *(i.e.,* $A_M \subseteq A$*).*
4. *For each bid* $b \in B$, *the total traded volume of bid* b *in the matching* M, *denoted by* $Q(b, M)$, *is not more than its maximum quantity (i.e., for all* $b \in B, Q(b, M) \leq q_b$*).*
5. *For each ask* $a \in A$, *the total traded volume of ask* a *in the matching* M, *denoted by* $Q(a, M)$, *is not more than its maximum quantity (i.e. for all* $a \in A, Q(a, M) \leq q_a$*).*

Description. Note that there might be some bids in B which are not matched to any asks in M and some asks in A which are not matched to any bids in M.

▶ Note 5. For simplicity, with slight abuse of notation, we use Q to denote total quantity of various objects which will be clear from the context. So, $Q(b, M)$ and $Q(a, M)$ represent the total quantities of the bid b and the ask a traded in the matching M, respectively. Similarly, the terms $Q(B)$ and $Q(A)$ denote the sum of the quantities of all the bids in B and the sum of the quantities of all the asks in A, respectively. And also, for the total traded quantity in a matching M, we use the term $Q(M)$. However, in the Coq implementation, each of these terms are represented by different names: QMb, QMa, QB, QA and QM.

Formalization notes: We have defined Bid, Ask and Transaction as record types in Coq. We define the proposition *matching_in* B A M to be true if and only if M is a matching between the list of bids B and the list of asks A.

4.1 Matching Demand and Supply

Let $B_{\geq p}$ represents the list of bids in B whose limit prices are at least a given number p. Similarly, $A_{\leq p}$ represents the list of asks in A whose limit prices are at most p. Therefore, the quantities $Q(B_{\geq p})$ and $Q(A_{\leq p})$ represents the total demand and the total supply of the product at the price p in the market, respectively. Although, in general we cannot say much about the relationship between the total demand (i.e. $Q(B_{\geq p})$) and supply (i.e. $Q(A_{\leq p})$) at an arbitrary price p, we can prove the following important results about the traded quantities of the matched bid-ask pairs.

▶ **Lemma 6.** *If* M *is a matching between the list of bids* B *and the list of asks* A, *then*

$$Q(M) = \sum_{b \in B} Q(b, M) \leq \sum_{b \in B} q_b = Q(B) \text{ and } Q(M) = \sum_{a \in A} Q(a, M) \leq \sum_{a \in A} q_a = Q(A)$$

▶ **Theorem 7.** *If* M *is a matching between the list of bids* B *and the list of asks* A, *then for all natural numbers* p, *we have* $Q(M) \leq Q(B_{\geq p}) + Q(A_{\leq p})$

Theorem 7 states that no matching M can achieve a trade volume higher than the sum of the total demand and supply in the market at any given price.

Proof Idea. We first partition the matching M into two lists: $M_1 = \{m \in M \mid p(b_m) \geq p\}$ and $M_2 = \{m \in M \mid p(b_m) < p\}$. Thus, $Q(M) = Q(M_1) + Q(M_2)$.

It is easy to see that M_1 is a matching between $B_{\geq p}$ and A, and hence from Lemma 6, $Q(M_1) \leq Q(B_{\geq p})$.

Next, we prove that M_2 is a matching between B and $A_{\leq p}$. Consider a transaction m from M_2. Since $m \in M$, $p(b_m) \geq p(a_m)$, and from the definition of M_2, we have $p(b_m) < p$. This implies $p(a_m) < p$, i.e., asks of M_2 come from $A_{\leq p}$. Hence, M_2 is a matching between B and $A_{\leq p}$, and applying Lemma 6, we have $Q(M_2) \leq Q(A_{\leq p})$.

Combining, we have $Q(M) = Q(M_1) + Q(M_2) \leq Q(B_{\geq p}) + Q(A_{\leq p})$, which completes the proof of Theorem 7. ☐

Formalization notes: The formal proof of Theorem 7 is completed by first proving the Lemmas *Mbgep_bound* ($Q(M_1) \leq Q(B_{\geq p})$) and *Mbltp_bound* ($Q(M_2) \leq Q(A_{\leq p})$) and then combining them in theorem *bound_on_M*. These results can be found in the file "Bound.v".

4.2 Individual-Rational Trades

An auctioneer assigns a trade price to each matched bid-ask pair. In any matching it is desired that the trade price of a bid-ask pair lies between their limit prices. A matching which has this property is called an *individual-rational (IR)* matching.

▶ **Definition 8** (Individual rational). Is_IR(M) := *for all* $m \in M,\ p(b_m) \geq p_m \geq p(a_m)$.

Note that any matching can be converted to individual-rational by changing the price of each transaction to lie between the limit prices of its bid and ask (See Fig 1).

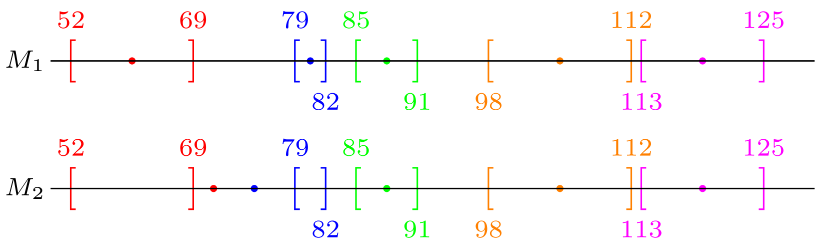


Figure 1 The colored dots represent trade prices for matched bid-ask pairs. Matching M_2 is not IR but M_1 is IR, even though both the matchings contain exactly the same bid-ask pairs.

5 Fairness in Competitive Markets

A double sided auction is a competitive event, where the priority among participating traders is determined by various attributes of the orders. A bid with higher limit price is considered more *competitive* compared to bids with lower limit prices. Similarly, an ask with lower limit price is considered more competitive compared to asks with higher limit prices. Ties are broken in favor of the requests that have an earlier arrival time. A matching which prioritizes more competitive traders is called a *fair* matching.

▶ **Definition 9** (Arrow notation). $\overset{\mathbb{B}}{\uparrow} L$ *denotes that the list L is sorted as per the competitiveness of the bids in L, with the most competitive bid being on top. Similarly,* $\overset{\mathbb{A}}{\uparrow} L$ *denotes that the* *list L is sorted as per the competitiveness of the asks in L, with the most competitive ask being on top. Similarly we can define $\overset{\mathbb{A}}{\downarrow}$ and $\overset{\mathbb{B}}{\downarrow}$ for sorting lists where the most competitive orders lie at the bottom.*

In this section, we show that there exists a procedure Fair that takes a matching M between bids B and asks A as input and outputs a fair matching $M' = \mathsf{Fair}(M, B, A)$ with the same trade volume as that of M. To describe the Fair procedure, we will need the following definitions.

▶ **Definition 10.** *Let M be a matching between bids B and asks A.*

- *M is fair on bids if for all pairs of bids $b_1, b_2 \in B$ such that b_1 is more competitive than b_2 and b_2 participates in the matching M, then b_1 is fully traded in M (i.e., $Q(b_1, M) = q(b_1)$).*
- *Similarly, M is fair on asks if for all pairs of asks $a_1, a_2 \in A$ such that a_1 is more competitive than a_2 and a_2 participates in the matching M, then a_1 is fully traded in M (i.e., $Q(a_1, M) = q(a_1)$).*
- *M is fair if it is both fair on bids and asks.*

The Fair procedure works in two steps: first, it sorts the matching M and the asks A based on the competitiveness of the asks and then runs on them a procedure "fair on asks" FOA that outputs a matching M' that is of the same volume as that of M and is fair on the asks. In the second step, it sorts the resulting matching M' and the bids B based on the competitiveness of the bids and then runs on them a procedure fair on bids FOB that outputs a matching M'' that is of the same volume as that of M' and is fair on the bids. The Fair procedure returns M'' as its output. The procedures FOB and FOA, along with their correctness proofs, mirror each other and we just describe FOB below and show that $\mathsf{FOB}(\overset{\mathbb{B}}{\uparrow} M', \overset{\mathbb{B}}{\uparrow} B)$ outputs a fair on bids matching and has the same trade volume as that of M'. Furthermore, we will show that if M' is fair on asks, then $\mathsf{FOB}(\overset{\mathbb{B}}{\uparrow} M', \overset{\mathbb{B}}{\uparrow} B)$ is fair on asks. This will immediately imply that the procedure $\mathsf{Fair}(M, B, A)$ outputs a fair matching with the same total trade volume as that of M.

5.1 Fair on Bids

To describe the fair on bids FOB procedure, we first need the following notation.

▶ **Definition 11.** *Given a list L and and an element a, $a :: L$ denotes the list whose top element (head) is a and the following elements (tail) are the elements of L (in the same order as they appear in L).*

The FOB procedure takes sorted (based on the bids' competitiveness) lists of transactions M and bids B. Intuitively, when all the bids are of unit quantity, we want to scan the list of transactions in M from top to bottom replacing the bids therein with the bids of B from top to bottom. So, in effect, in the FOB procedure we will implement this intuition apart from taking care of multiple quantity bids; and also make the procedure recursive so that we can provide a formalization friendly inductive proof of correctness. Let $B = b :: B'$ and $M = m :: M'$. In our procedure, we first pick the top bid b of B and the top transaction m of M, and compare q_b with q_m. Now we have three cases. In each of the three cases, the procedure first outputs a transaction between the bid b and the ask of m of quantity $\min\{q_m, q_b\}$. Case I: If $q_b = q_m$, we remove b and m from their respective lists and recursively solve the problem on B' and M'. Case II: If $q_b < q_m$, we remove b from the list B and update

q_m to $q_m - q_b$ and recursively solve the problem on B' and M. Case III: If $q_b > q_m$, we remove m from the list M and set a parameter t to q_m that we will send to the recursive call along with M' and B. The parameter t informs our recursive procedure that the top element b of B has effectively quantity $q_b - t$. Thus, our procedure will take three parameters: the list of transactions, the list of bids and the parameter t (Note that unlike Case II ($q_b < q_m$) where the top transaction is updated, the top bid is not updated in Case III ($q_b > q_m$). This is done for technical reasons: Later we need to prove that the set of bids of FOB is a subset of B, and at the same time we have to ensure that the total traded quantity of the bid b in the matching outputted by FOB remains below its maximum quantity q_b, as required by the matching property. This would not be possible to do if we updated B and hence we take this approach of keeping the total traded quantity of the top bid b in a separate argument: t of f.). Keeping this description in mind, we now formally define the procedure FOB.

▶ **Definition 12** (Fair On Bid (FOB)).

$$\mathsf{FOB}(M,B) = f(M,B,0)$$

where $f(M,B,t) =$

$$\begin{cases} nil & \text{if } M = nil \text{ or } B = nil \\ (b, a_m, q_m, p_m) :: f(M', B', 0) & \text{if } q_m = q_b - t \\ (b, a_m, q_m, p_m) :: f(M', b :: B', t + q_m) & \text{if } q_m < q_b - t \\ (b, a_m, q_b - t, p_m) :: f((b_m, a_m, q_m - (q_b - t), p_m) :: M', B', 0) & \text{if } q_m > q_b - t \end{cases}$$

where $M = m :: M'$ *when* $M \neq nil$ *and* $B = b :: B'$ *when* $B \neq nil$.

▶ **Theorem 13.** *Let M be a matching between bids B and asks A where the lists M and B are sorted in the descending order of the competitiveness of their bids (i.e., the most competitive bid and the transaction with the most competitive bid are on top of their respective lists). Let $M_\beta = \mathsf{FOB}(M, B)$, then*

(a) *M_β is a matching between bids B and asks A.*
(b) *For each ask $a \in A$, the total traded quantity of a in M is same as the total traded quantity of a in M_β (i.e., $Q(a, M) = Q(a, M_\beta)$). As a corollary, we get that if M is fair on asks, then M_β is also fair on asks.*
(c) *The total traded quantity of M is equal to the total traded quantity of M_β (i.e., $Q(M) = Q(M_\beta)$).*
(d) *The matching M_β is fair on bids.*

Proof Outline. Here, we briefly describe certain aspects of the proof; more details can be found in the full version of the paper [5] and for the complete formalization see [10]. Note that in each of the recursive calls in f, either the size of the first argument $|M|$ decreases or the size of the second argument $|B|$ decreases. Therefore, we prove the above statements using (well founded) induction on the sum ($|M| + |B|$). Proof of (a) and (b) is done using induction and case analysis. The proof of (c) follows by combining Lemma 6 with (b). We focus on the proof of (d) below.

Let bid b be the top element of the bids B and $B = b :: B'$. First, we prove two general results:

$$\text{For all } t, \text{ if } Q(M) \geq q_b - t, \text{ then } Q(f(M, b :: B', t), b) = q_b - t,$$

which states that if the total trade volume of the matching M is at least $q_b - t$, then in the matching $f(M, b :: B', t)$ the top bid b has trade quantity $q_b - t$. The proof of this can be done using induction on the size of M. Intuitively, f tries to match as much quantity of the top bid b with the top transaction in M. When the call $f(M, b :: B', t)$ is made, the top bid b already has t traded quantity and $q_b - t$ of its quantity remains untraded. If the quantity of the top transaction m of M is at least $q_b - t$, then we are done. Otherwise, f matches q_m quantities of b and recursively calls f on a smaller list and then we will be done by applying the induction hypothesis.

Now, we state the second general result.

For all t, if distinct bids b, b' belong to the bids of $f(M, b :: B', t)$, then $Q(M) \geq q_b - t$.

This result can be proved, like the previous result, using induction on the sum $(|M| + |B|)$; see [10] for details. Intuitively, since b' is matched by $f(M, b :: B', t)$ (in particular $b' \in B'$), then $f(M, b :: B', t)$ will completely match b (which has at least $q_b - t$ quantity remaining untraded) before it matches even a single quantity of b'.

Now using the above general results, we prove (d). We need to show the following: for all $b_1, b_2 \in B$, if b_1 is more competitive than b_2 and $Q(\mathsf{FOB}(M, B), b_2) \geq 1$, then $Q(\mathsf{FOB}(M, B), b_1) = q_{b_1}$, i.e., if the bid b_2 participates in the matching M then the bid b_1 is fully traded in M. Fix $b_1, b_2 \in B$ such that b_1 is more competitive than b_2 and $Q(\mathsf{FOB}(M, B), b_2) \geq 1$. Note that the bid b_2 cannot be equal to the bid b since bids B are sorted. Now we analyze three possible cases: $b_1 \neq b$, $b_1 = b$ and $Q(M) \geq q_b$, and $b_1 = b$ and $Q(M) < q_b$.

- In the case when $b_1 \neq b$, we consider the recursive call where b_1 is the top bid in the argument for the first time. In this recursive call the list of bids is smaller than B since the bid b must be fully traded before. Then, we are immediately done by applying the induction hypothesis.
- In the case $b_1 = b$ and $Q(M) \geq q_b$, in the matching $\mathsf{FOB}(M, b :: B') = f(M, b :: B', 0)$ the top bid b has total trade volume $q_b - 0 = q_b$ from the first general result invoked with $t = 0$, and hence $b_1 = b$ is fully traded.
- In the case $b_1 = b$ and $Q(M) < q_b$, we arrive at the contradiction $Q(M) \geq q_b$ by invoking the second general result with $t = 0$, $b = b_1$, $b' = b_2$ and $\mathsf{FOB}(M, b :: B') = f(M, b :: B', 0)$. ◻

Similar to the procedure FOB, we have a procedure FOA, that produces a fair matching on asks (see [10]). Combining the FOA and FOB procedures, we have the following definition of the Fair procedure.

▶ **Definition 14.** $\mathsf{Fair}(M, B, A) = \mathsf{FOB}(\overset{\mathbb{B}}{\uparrow} \mathsf{FOA}(\overset{\mathbb{A}}{\uparrow} M, \overset{\mathbb{A}}{\uparrow} A), \overset{\mathbb{B}}{\uparrow} B)$.

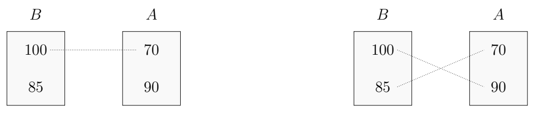
We conclude this section by formally summarizing the main fairness result.

▶ **Theorem 15.** *If M is a matching on the list of bids B and the list of asks A, then the matching $M' = \mathsf{Fair}(M, B, A)$ on B and A is a fair matching such that $Q(M) = Q(M')$.*

Formalization notes: The procedure FOB and FOA are implemented in Coq using the Equations plugin which is helpful to write functions involving well-founded recursion [9]. The proof of Theorem 15 is quite extensive and done in several parts. First we prove all the parts of Theorem 13 in the file "mFair_Bid.v". We prove similar theorems for the procedure FOA in "mFair_Ask.v" file. Later all the results are combined in the file "MQFair.v" and the above theorem is proved as *exists_fair_matching*.

6 Uniform Price Matchings in Financial Markets

Liquidity in a market is a measure of how quickly one can trade in that market and maximizing the total trade volume helps increase liquidity. However, to maximize the total trade volume sometimes we have to accept different trade prices to the matched bid-ask pairs (Fig 2).



(a) UM (b) MM

Figure 2 Both the bids and the asks have quantity one. The only individually rational matching of size two is not uniform.

Assigning different trade prices for the same product in the same market simultaneously, might lead to dissatisfaction among some traders. As stated in the introduction, in the financial markets, the matching should be fair and optimal individual-rational-uniform. In this section, we describe the UM process that takes as input a list of bids and a list of asks and produces a fair and optimal individual-rational-uniform matching that can be directly applied in the financial markets for conducting double sided auctions. We present a novel proof of optimality of the UM process.

Before we describe the UM process, we first give some intuition. Observe that in any individual-rational and uniform matching M all the buyers are matched at a single price p and the price p lies between the limit prices of all the matched bid-ask pairs. This means all the matched bids' limit prices are at least p and all the matched asks' limit prices are at most p. In the special case when all the orders are of unit quantity, the matching can be visualized as a fully nested balanced parenthesis (for example, [[[[ ]]]]) where each bid is represented by a closed parenthesis "]" and each ask as an open parenthesis "[" (See Figure 1).

Now, we describe the UM process. We recursively pair the most competitive available bid with the most competitive available ask, if they are matchable. The trade quantity for each matched bid-ask pair is the minimum of the remaining quantities of the respective bid and the ask. The trade price assigned to each pair is the price of the ask in that pair[2]. We terminate the process once there are no more matchable bid-ask pairs remaining. At the end of the process, to produce a uniform matching we have to assign a single trade price to all the matched bid-ask pairs which we choose to be the trade price of the last matched bid-ask pair (which also keeps the individual-rational property intact).

Keeping this description in mind, we now formally define the UM process using recursion.

▶ **Definition 16** (Uniform Matching (UM)). .

$$\mathsf{UM}(B,A) = \mathsf{Replace\_prices}(f_u(\overset{\mathbb{B}}{\uparrow} B, \overset{\mathbb{A}}{\uparrow} A, 0, 0), \mathsf{Last\_trade\_price}(f_u(\overset{\mathbb{B}}{\uparrow} B, \overset{\mathbb{A}}{\uparrow} A, 0, 0)))$$

$$\textit{where } f_u(B, A, t_b, t_a) =$$

$$\begin{cases} nil & \textit{if } B = nil \textit{ or } A = nil \textit{ or } p_b < p_a \\ (b, a, q_b - t_b, p_a) :: f_u(B', A', 0, 0) & \textit{if } q_a - t_a = q_b - t_b \textit{ and } p_b \geq p_a \\ (b, a, q_b - t_b, p_a) :: f_u(B', a :: A', 0, t_a + q_b - t_b) & \textit{if } q_a - t_a > q_b - t_b \textit{ and } p_b \geq p_a \\ (b, a, q_a - t_a, p_a) :: f_u(b :: B', A', t_b + q_a - t_a, 0) & \textit{if } q_a - t_a < q_b - t_b \textit{ and } p_b \geq p_a \end{cases}$$

where $B = b :: B'$ *when* $B \neq nil$ *and* $A = a :: A'$ *when* $A \neq nil$.

[2] Observe that any value in the interval of the limit prices of the matched bid-ask pair can be assigned as the trade price and it will not affect any analysis done in this work.

Description. Observe that, similar to the parameter t in the FOB process, we have two parameters t_b and t_a that inform the recursive procedure f_u that the top bid b and the top ask a have effective quantities $q_b - t_b$ and $q_a - t_a$, respectively. In each recursive call, the process f_u outputs a transaction (top bid b, top ask a, quantity $\min\{q_b - t_b, q_a - t_a\}$, price p_a). The process f_u terminates when the top bid is not matchable with the top ask.

Remark 1. It is easy to see that UM outputs a uniform matching: Once the f_u process terminates, Last_trade_price computes the trade price of the last transaction in the output of f_u and Replace_prices replaces the trade prices of each transaction of the output of f_u with the trade price of the last transaction of the output, thus ensuring UM produces a uniform matching. Also, notice that the process Replace_prices does not alter any other information of the output of f_u apart from the trade prices (we will later use this fact in the proof of optimality of UM).

Remark 2. It is easy to see that UM outputs an individual-rational matching: the trade price of a transaction m outputted by a recursive call of f_u is between the limit prices of the bid and the ask of m. Later these prices are altered by Replace_prices, but the individual-rational property is not lost; the trade price of m is also between the limit prices of the transactions of all the previous calls as the bids and the asks are sorted by their competitiveness, and Replace_prices replaces all the trade prices with the trade price of the last transaction.

Now, we discuss the optimality result of the UM process. Throughout this discussion, WLOG, all lists of bids and asks will be sorted by their competitiveness. We make use of the following notation.

▶ **Definition 17.** *Given a matching M, a bid b and an ask a, we use $Q(a \leftrightarrow b, M)$ to denote the total traded quantity between the bid b and the ask a in the matching M.*

Next, we state the main result of this section.

▶ **Theorem 18.** *Given a list of bids B and a list of asks A, let M_U = UM(B, A) and let M be an arbitrary individual-rational and uniform matching between B and A. Then, $Q(M_U) \geq Q(M)$. In other words,* UM *outputs an optimal individual-rational-uniform matching.*

To prove the above theorem, we need the following lemma.

▶ **Lemma 19.** *If M is an individual-rational and uniform matching between the lists of bids $B = b :: B'$ and asks $A = a :: A'$ such that $Q(M) \geq \min\{q_b, q_a\}$, then there exists another individual-rational and uniform matching M' between the same lists of bids B and asks A such that $Q(M) = Q(M')$ and $Q(a \leftrightarrow b, M') = \min\{q_b, q_a\}$.*

Assuming this lemma, we will first prove Theorem 18 and then later prove the lemma.

Proof of Theorem 18. Note that $f_u(B, A, 0, 0)$ is a specific instance of $f_u(B, A, t_b, t_a)$. So in order to apply the induction hypothesis, we sensitize the theorem statement to incorporate arbitrary values of t_a and t_b. Also, as indicated earlier, the Replace_prices function does not alter the total trade quantity of the output of the f_u, thus $Q(f_u(B, A, 0, 0)) = Q(\text{UM}(B, A))$. Consequently, showing the following suffices.

(∗) Fix an arbitrary list of bids $B = b :: B'$ and an arbitrary list of asks $A = a :: A'$. Fix arbitrarily $t_b < q_b$ and $t_a < q_a$. Let b' be the bid obtained from the bid b by reducing its quantity to $q_b - t_b$. Similarly, let a' be the ask obtained from the ask a by reducing its quantity to $q_a - t_a$. We will show: for all individual-rational and uniform matchings M between $(b' :: B')$ and $(a' :: A')$, $Q(f_u(B, A, t_b, t_a)) \geq Q(M)$.

Clearly, setting $t_b = t_a = 0$ in the above statement $(*)$ gives us Theorem 18.

We prove the above statement using induction on the sum $(|B| + |A|)$. We consider two cases: $p(b') < p(a')$ and $p(b') \geq p(a')$.

In the first case, when $p(b') < p(a')$, since the most competitive bid in $b' :: B'$ is not matchable with the most competitive ask in $a' :: A'$, any matching between $b' :: B'$ and $a' :: A'$ is empty. Thus, $Q(M) = 0$, and we are done.

In the second case, when $p(b') \geq p(a')$, if the total trade quantity of M is less than the quantity of the transaction created by f_u in the first recursive call (i.e., $Q(M) < \min\{q_b - t_b, q_a - t_a\} \leq Q(f_u(B, A, t_b, t_a))$, then we are done. In the case when the total traded quantity of M is more than the quantity of the transaction created by f_u in the first recursive call (i.e., $Q(M) \geq \min\{q_b - t_b, q_a - t_a\}$), we apply Lemma 19 and get another individual-rational and uniform matching M' such that the total volume of M' is equal to the total volume of M and the total traded quantity between the bid b' and the ask a' in M' is equal to $\min\{q_{b'} = q_b - t_b, q_{a'} = q_a - t_a\}$. Now since we have M' such that $Q(M) = Q(M')$, proving the following suffices.

$$Q(f_u(B, A, t_b, t_a)) \geq Q(M') \qquad\qquad (**),$$

where M' is an individual-rational and uniform matching between the list of bids $b' :: B'$ and $a' :: A'$ such that $Q(a' \leftrightarrow b', M') = \min\{q_b - t_b, q_a - t_a\}$. We define the matching $M_0 \subseteq M$ as follows: we remove all transactions between b' and a' (of total quantity $Q(a' \leftrightarrow b', M')$) from M' to get M_0. We have (†): $Q(M') = \min\{q_a - t_a, q_b - t_b\} + Q(M_0)$. Also, note that M_0 is individual-rational and uniform (since $M' \supseteq M_0$ is individual-rational and uniform).

Now we argue the proof of $(**)$ in each of the three recursive branches of the function f_u corresponding to $p(b) \geq p(a)$.

- Case: $q_a - t_a = q_b - t_b$. In this case M_0 is a matching between B' and A'. Since $(|B| + |A|) > (|B'| + |A'|)$, we can apply the induction hypothesis to get $Q(f_u(B', A', 0, 0)) \geq Q(M_0)$. Now, applying the definition of f_u we get,

$$\begin{aligned} Q(f_u(B, A, t_b, t_a)) &= \min\{q_a - t_a, q_b - t_b\} + Q(f_u(B', A', 0, 0)) \\ &\overset{\text{I.H.}}{\geq} \min\{q_a - t_a, q_b - t_b\} + Q(M_0) \overset{(\dagger)}{=} Q(M'). \end{aligned}$$

- Case: $q_a - t_a > q_b - t_b$. In this case M_0 is a matching between B' and $\hat{a} :: A'$ (where $q_{\hat{a}} = q_a - t_a - (q_b - t_b) \leq q_a$). Since $(|B| + |A|) > (|B'| + |\hat{a} :: A'|)$, we can apply the induction hypothesis when $B' \neq \emptyset$ to get $Q(f_u(B', A, 0, t_a + (q_b - t_b))) \geq Q(M_0)$. When $B' = \emptyset$, then $Q(f_u(B', A, 0, t_a + (q_b - t_b))) \geq Q(M_0)$ holds trivially as both the sides of the inequality are zeros. Now, applying the definition of f_u we get,

$$\begin{aligned} Q(f_u(B, A, t_b, t_a)) &= (q_b - t_b) + Q(f_u(B', A, 0, t_a + (q_b - t_b))) \\ &\overset{\text{I.H.}}{\geq} (q_b - t_b) + Q(M_0) \overset{(\dagger)}{=} Q(M'). \end{aligned}$$

- Case: $q_a - t_a < q_b - t_b$. This is symmetric to the previous case and the proof follows similarly. ☐

Having finished the proof of the main result, we now discuss the proof of the lemma that we assumed.

Main proof idea of Lemma 19. Given an individual-rational and uniform matching M with $Q(M) \geq \min\{q_b, q_a\}$ between the list of bids $B = b :: B'$ and the list of asks $A = a :: A'$, we need to show existence of an individual-rational and uniform matching M' such that $Q(M') = Q(M)$ and the total trade quantity between the bid b and ask a in M' is $\min\{q_b, q_a\}$. We do the following surgery on M in two steps to obtain the desired M'.

Step 1: We first modify M to ensure that bid b and ask a each has at least $\min\{q_b, q_a\}$ total trades in M (not necessarily between each other). This is accomplished by running the Fair procedure on M that outputs a matching which prefers the most competitive orders (b and a) over any other orders. Since $Q(M) \geq \min\{q_b, q_a\}$, we get that $\mathsf{Fair}(M, B, A)$ has at least $\min\{q_b, q_a\}$ trades for each of b and a. Note that Fair does not change the total trade quantity or affect the individual-rational and uniform properties of M. Set $M \leftarrow \mathsf{Fair}(M, B, A)$.

Step 2: In this step, we modify M to ensure that the bid b and ask a have $\min\{q_b, q_a\}$ quantity trade between them. Note that in M individually both b and a have at least $\min\{q_b, q_a\}$ total trade quantity. We will inductively transfer trades of b and a that are not between them to the transaction between b and a, a unit quantity at a time, till they have $\min\{q_b, q_a\}$ quantity trade between them. To better understand this, consider the case when b and a have zero trade quantity between them. Let us say there is a transaction between b and a_1 of quantity q_1 and a transaction between a and b_1 of quantity q_2. We remove these two transactions and replace it with the following four transactions (see Figure 3) that keeps the matching trade volume intact: (1) transaction between b and a_1 of quantity $q_1 - 1$, (2) transaction between a and b_1 of quantity $q_2 - 1$, (3) transaction between b_1 and a_1 of quantity one and (4) transaction between b and a of quantity one. Recall, in a individual-rational and uniform matching with price p, the limit price of each bid is at least p and the limit price of each ask is at most p, implying any bid and ask participating in the matching are matchable. Thus, doing such a replacement surgery is legal and does not affect the individual-rational and uniform properties, and we obtain the desired M' by repeatedly doing this surgery.

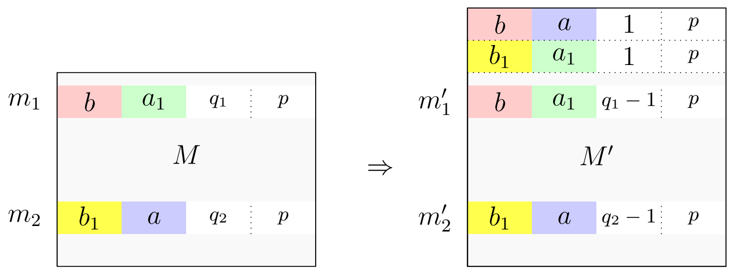


Figure 3 In the above figure the matching M' is obtained from the matching M. Each bid or ask has the same trade quantity in both M and M'. Furthermore, the trade quantity between a and b in M' is one more than that in M.

The proof that UM produces a fair matching follows from inducting on the sum $(|A| + |B|)$ and the fact that B and A are sorted by competitiveness of the participating bids and asks. The argument is similar to the correctness proof of Fair that we saw before. From the discussion above, the next theorem follows immediately.

▶ **Theorem 20.** *For a given list of bids B and the list of asks A, $M = \mathsf{UM}(B, A)$ is a fair and optimal individual-rational-uniform matching on B and A.*

Formalization notes: The formalized proof of the above theorem is done by first proving Lemma 19 (*exists_opt_k*) using induction on the gap $k = \min\{q_b, q_a\} - Q(a \leftrightarrow b, M)$. From this lemma, we get another matching M' such that $Q(a \leftrightarrow b, M) = \min\{q_b, q_a\}$. The matching M' is altered to M_0 (as described in the proof of Theorem 18 above) by removing all the transactions between the bid b and the ask a. We prove that the altered list M_0 is is a matching between the reduced lists of bids and asks. All the results related to M_0 are in the file "MachingAlter.v". Finally, combining all these we prove the main Theorem 20 as *"UM_main"*.

7 A Maximum Matching Mechanism

In the previous section, we indicated that to achieve maximum trade volume matching we sometimes have to assign different trade prices to the matched bid-ask pairs. An individual rational matching with maximum trade volume is called a maximum matching. In this section we describe a process MM, that takes a list of bids and a list of asks and outputs a fair, individual-rational and maximum matching.

The MM procedure roughly works as follows. In step one, the MM procedure repeatedly pairs the most competitive bid b with the least competitive matchable ask a and outputs a transaction $(b, a, \min\{q_b, q_a\}, p_a)$ and decreases the quantities of b and a by $\min\{q_b, q_a\}$. In step two, the MM procedure applies the Fair procedure on the output of step one.

The detailed MM procedure and the proof of its correctness are similar to that of the UM procedure in spirit. In the proof of optimality, we need to prove a lemma similar to Lemma 19 which states that a given arbitrary individual-rational matching M of sufficiently large trade volume can be altered to obtain a matching M' of the same total trade volume such that the total trade quantity between the most competitive bid and the corresponding least competitive matchable ask in M' is the minimum of their respective quantities. The proof of this requires more surgeries as compared to that in the proof of Lemma 19. Besides this deviation all other arguments of the proof of optimality of MM are similar to that of UM with minor variations.

The proof of MM producing an individual-rational matching is trivial and the proof that it produces a fair matching follows from the fact that MM applies the Fair procedure before it outputs a final matching. We now state the main theorem of this section.

▶ **Theorem 21.** *For a given list of bids B and a list of asks A,* MM(B, A) *is a fair, individual-rational and maximum trade volume matching between B and A.*

The proof of the above theorem and discussion around the MM procedure can be found in the full version of the paper [5].

Formalization notes: All the formalization details can be found in [10].

8 Uniqueness Theorem

In this section, we establish certain theorems that enable us to automatically check for violations in an exchange matching algorithm by comparing its output with the output of our certified program. Detailed proofs are available in the Coq formalization [10].

Ideally, we would have wanted a theorem that the properties (fair and optimal individual-rational-uniform) imply a unique matching. Such a theorem would enable us to automatically compare a matching produced by an exchange with a matching produced by our certified program to find violations of these properties in the matching produced by the exchange. Unfortunately, such a theorem is not possible; there exists two different matchings M_1 and M_2 on the same list of bids B and asks A, where both are fair and optimal individual-rational-uniform: $M_1 = \{(b_1, a_1, 1, p), (b_2, a_2, 2, p)\}$ and $M_2 = \{(b_1, a_2, 1, p), (b_2, a_2, 1, p), (b_2, a_1, 1, p)\}$

on bids $B = \{b_1 = (*, *, 1, p), b_2 = (*, *, 2, p)\}$ and asks $A = \{a_1 = (*, *, 1, p), a_2 = (*, *, 2, p)\}$ for some arbitrary price p, timestamps and ids. Note that fairness does not require the most competitive bid to be paired with the most competitive ask. For example, assuming a_1 has a lower timestamp than a_2 and b_1 has a lower timestamp than b_2 in the above example, a_1 and b_1 are not matched in the matching M_2, which is a fair matching. Nonetheless, we can show that given a list of bids B and a list of asks A, all matchings that are fair and individual-rational-uniform, must have the same trade volume for each trader. This still allows us to automatically check for violations of the properties in an exchange, by comparing the trades of each trader produced by the exchange against that produced by our certified program.

We have the following lemma which formulates this uniqueness relation on the matchings.

▶ **Theorem 22.** *Let M_1 and M_2 be two fair matchings on the list of bids B and the list of asks A such that $Q(M_1) = Q(M_2)$, then for each order ω, the total traded quantity of ω in M_1 is equal to the total traded quantity of ω in M_2.*

Proof Idea. We now prove the above theorem by using Lemma 6 and deriving a contradiction. Let M_1 and M_2 be fair matchings such that $Q(M_1) = Q(M_2)$. Let b be a buyer whose total trade quantity in M_1 is different (WLOG, more) from his total trade quantity in M_2. It is easy to show that there exists another buyer b' such that her total traded quantity in M_1 is less than her total traded quantity in M_2, i.e., $Q(M_2, b') > Q(M_1, b')$ (since the sum of the total traded quantities of all the bids of B in M_1 is equal to the sum of the total traded quantities of all the bids of B in M_2 from Lemma 6).

Now, there can be two cases: (i) b is more competitive than b' or (ii) b' is more competitive than b, as per price-time priority. In the first case, since $Q(M_1, b) > Q(M_2, b)$, it follows that $Q(M_2, b) < Q(M_1, b) \leq q_b$. This contradicts the fact that M_2 is fair on the bids; this is because a less competitive bid b' is being traded in M_2 (since $Q(M_2, b') > Q(M_1, b') \geq 0$ as noted above), while a more competitive bid b is not fully traded. Similarly, in the second case, we show a contradiction to the fact that M_1 is fair on the bids. ☐

From the above theorem, we have the following corollary.

▶ **Theorem 23.** *For any two fair and optimal individual-rational-uniform matchings M_1 and M_2 on the list of bids B and the list of asks A, for each order ω, the total traded quantity of ω in M_1 is equal to the total traded quantity of ω in M_2.*

For each trader, we can compare the total traded quantities of the trader in the matching M_1 produced by an exchange with the total traded quantities of the trader in the matching $M_2 = \mathsf{UM}(B, A)$ produced by our certified program. If for some trader, the traded quantities do not match, then from Theorem 20 and Theorem 23 we know that M_1 does not have the desired properties as required by the regulators. On the other hand, if they do match for all traders, then the following theorem states that M_1 is fair (Note that uniform and individual-rational properties can be verified directly from the trade prices and clearly the total trade volume of M_1 and M_2 are the same if the traded quantities are same for each trader).

▶ **Theorem 24.** *Given a list of bids B and a list of asks A, if M_1 is a fair matching and M_2 is an arbitrary matching such that for each order ω, the total traded quantity of ω in M_1 is equal to the total traded quantity of ω in M_2, then M_2 is fair.*

The proof follows immediately from the definition of fairness.

Formalization notes: All the theorems in this section are formalized in the file "Uniqueness.v" using the above proof ideas.

9 Demonstration: Automatic Detection of Violations in Real Data

Please see Appendix A for details on our demonstration, where we automate the process of checking violations in trades using verified programs extracted from our formalization. We then use this to find violations in trades of 100 stocks traded on a real exchange on a particular day. Below, we describe our findings.

Out of the 100 stocks we checked, for three stocks our program outputted "Violation detected!". When we closely examined these stocks, we realized that in all of these stocks, a market ask order (with limit price = 0), was not matched by the exchange in its trading output (and these were the only market ask orders in the entire order-book). On the contrary, market bid orders were matched by them. With further investigation, we observed that corresponding to each of these three violations, in the raw data there was an entry of update request in the order-book with a limit price and timestamp identical to the uniform price and the auction time, respectively. It seems highly unlikely that these three update requests were placed by the traders themselves (to match the microsecond time and also the trade price seems very improbable); we suspect this is an exchange's system generated entry in the order-book. We hope that the exchange is aware of this and doing this consciously. When we delete the market asks in the preprocessing stage, no violations are detected. Even if it is not a violation (but a result of the exchange implementing some unnatural rule that we are not aware of), it is fascinating to see that with the help of verified programs we can identify such minute and interesting anomalies which can be helpful for regulating and improving the exchange's matching algorithm.

10 Related Works and Future Direction

In an earlier work [8], Sarswat and Singh dealt primarily with single unit trade requests and thus provided a proof of concept for obtaining verified programs for financial markets. In the current work, we extend their work to multiple units that results in verified programs which we run on real market data and establish new uniqueness theorems that enable automatic detection of violation in exchanges as demonstrated in this work.

Passmore and Ignatovich in [7] highlight the significance, opportunities and challenges involved in formalizing financial markets. They describe the whole spectrum of financial algorithms that need to be verified for ensuring safe and fair markets. Iliano *et al.* [1] use concurrent linear logic (CLF) to outline two important properties of a continuous trading system. There are also some works formalizing various concepts from auction theory [2, 3, 11], particularly focusing on the Vickrey auction mechanism.

In our opinion, future works should focus on developing a theory for continuous double auctions for financial markets. Currently the specifications for continuous double auctions are vague and this is an obstacle for obtaining verified programs.

References

1 Iliano Cervesato, Sharjeel Khan, Giselle Reis, and Dragisa Zunic. Formalization of automated trading systems in a concurrent linear framework. In *Linearity-TLLA@FLoC*, volume 292 of *EPTCS*, pages 1–14, 2018. URL: `http://arxiv.org/abs/1904.06159`.

2 Cezary Kaliszyk and Julian Parsert. Formal microeconomic foundations and the first welfare theorem. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 91–101. ACM, 2018.

3 Stéphane Le Roux. Acyclic preferences and existence of sequential nash equilibria: a formal and constructive equivalence. In *International Conference on Theorem Proving in Higher Order Logics*, pages 293–309. Springer, 2009.
4 R Preston McAfee. A dominant strategy double auction. *Journal of economic Theory*, 56(2):434–450, 1992.
5 Raja Natarajan, Suneel Sarswat, and Abhishek Kr Singh. Verified double sided auctions for financial markets, 2021. `arXiv:2104.08437`.
6 Jinzhong Niu and Simon Parsons. Maximizing matching in double-sided auctions. In *International conference on Autonomous Agents and Multi-Agent Systems, AAMAS '13, Saint Paul, MN, USA, May 6-10, 2013*, pages 1283–1284, 2013.
7 Grant Olney Passmore and Denis Ignatovich. Formal verification of financial algorithms. In *26th International Conference on Automated Deduction, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 26–41. Springer, 2017.
8 Suneel Sarswat and Abhishek Kr Singh. Formally verified trades in financial markets. In Shang-Wei Lin, Zhe Hou, and Brendan Mahoney, editors, *Formal Methods and Software Engineering - 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1-3, 2021, Proceedings*, volume 12531 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 2020. `doi:10.1007/978-3-030-63406-3_13`.
9 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.
10 Suneel Sarswat. Coq formalization of mdsa. `https://github.com/suneel-sarswat/dsam`.
11 Emmanuel M. Tadjouddine, Frank Guerin, and Wamberto Weber Vasconcelos. Abstracting and verifying strategy-proofness for auction mechanisms. In *DALT*, volume 5397 of *Lecture Notes in Computer Science*, pages 197–214. Springer, 2008.
12 Peter R. Wurman, William E. Walsh, and Michael P. Wellman. Flexible double auctions for electronic commerce: theory and implementation. *Decision Support Systems*, 24(1):17–27, 1998.
13 Dengji Zhao, Dongmo Zhang, Md Khan, and Laurent Perrussel. Maximal matching for double auction. In *Australasian Conference on Artificial Intelligence*, volume 6464 of *Lecture Notes in Computer Science*, pages 516–525. Springer, 2010.

A Demonstration on real data

In this section, we demonstrate the practical applicability of our work. For this, we procured real data from a prominent stock exchange. This data consists of order-book and trade-book of everyday trading for a certain number of days. For our demonstration, we considered trades for the top 100 stocks (as per their market capitalizations) of a particular day. For privacy reasons, we conceal the real identity of the traders, stocks and the exchange by masking the stock names (to s1 to s100) and the traders' identities. We also converted the timestamps appropriately into natural numbers (which keeps the time in microseconds, as in the original data). Furthermore, the original data has multiple requests with the same order id; this is because some traders update or delete an existing order placed by them before the double sided auction is conducted. In our preprocessing, we just keep the final lists of bids and asks in the order-book that participate in the auction. Furthermore, there are certain market orders, i.e., orders that are ready to be traded at any available price, which effectively means a limit price of zero for an ask and a limit price of infinity for a bid; in the preprocessing we set these limit prices to zero and the largest OCaml integer, respectively.

We then extracted the verified OCaml programs and ran them on the processed market data. The output trades of the verified code were then compared with the actual trades in the trade-book from the exchange. From the uniqueness theorems in the Section 8, we know that if the total trade quantity of each order in these two matchings are equal, then the matching produced by the exchange has the desired properties (if it is uniform and IR which can be checked trivially by looking at the prices in the trade-book). We also know that if they are not equal for some trader, then the matching algorithm of the exchange does not have the requisite desired properties (or there is some error in storing or reporting the order-book or the trade-book accurately).

The processed data and the relevant programs for this demonstration are available at [10]. The extracted OCaml programs of the functions required for this demonstration are stored in a separate file named “certified.ml”. The input bids, asks and trades of each stock are in “s.bid”, “s.ask” and “s.trade” files, where “s” is the masked id for that stock. For example, file “s1.bid” contains all the bids for the stock “s1”. To feed the inputs to the verified program and to print the output of the certified program, we have written two OCaml scripts: create.ml and compare.ml. The create.ml script feeds inputs (lists of bids and asks) to the UM process, and then prints its output matching M. The compare.ml script compares the matching produced by the UM process M with the actual trades M_{EX} in the exchange trade-book. If the total trade quantity for all the traders in M matches with that of the total trade quantity in M_{EX}, then the compare.ml script outputs “Matching does not violate the guidelines”. If for some bid (or ask) the total trade quantity of M and M_{EX} does not match, then the program outputs “Violation detected!”.

Reaching for the Star: Tale of a Monad in Coq

Pierre Nigron ✉
Sorbonne Université, CNRS, Inria, LIP6, Paris, France

Pierre-Évariste Dagand ✉
Sorbonne Université, CNRS, Inria, LIP6, Paris, France

Abstract

Monadic programming is an essential component in the toolbox of functional programmers. For the pure and total programmers, who sometimes navigate the waters of certified programming in type theory, it is the only means to concisely implement the imperative traits of certain algorithms. Monads open up a portal to the imperative world, all that from the comfort of the functional world. The trend towards certified programming within type theory begs the question of *reasoning* about such programs. Effectful programs being encoded as pure programs in the host type theory, we can readily manipulate these objects through their encoding. In this article, we pursue the idea, popularized by Maillard [21], that every monad deserves a dedicated program logic and that, consequently, a proof over a monadic program ought to take place within a Floyd-Hoare logic built for the occasion. We illustrate this vision through a case study on the `SimplExpr` module of CompCert [18], using a separation logic tailored to reason about the freshness of a monadic gensym.

2012 ACM Subject Classification Software and its engineering → Software notations and tools

Keywords and phrases monads, hoare logic, separation logic, Coq

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.29

Supplementary Material *Software*: `https://github.com/Artalik/CompCert/tree/ITP`
archived at `swh:1:dir:bdd72cb914b59758fa869b3346e7674d1f8be2e4`

Funding *Pierre Nigron*: "Ministère des armées – Agence de l'Innovation de Défense" grant.
Pierre-Évariste Dagand: Emergence(s) – Ville de Paris grant.

1 Introduction

This article dwells on the challenges of verifying imperative algorithms implemented in a proof assistant. As certified programming becomes more commonplace, proof assistants are indeed being used as the ultimate integrated development environment [5, 10]. The question of specifying and proving the correctness of such programs is part of a long tradition, starting from various generalizations of monads [11, 33, 4] accounting for dependent types and YNot [24], an axiomatic extension of type theory featuring imperative traits, as well as the family of Dijsktra monads [3, 21, 22, 32] in F⋆ and their intuitionistic counterparts in Agda [35], including the recent activity around algebraic presentations of effects and their embedding in Coq and Agda [6, 7, 37, 20, 19]. This article reports on an experiment in revisiting a proof of Leroy [18] with the help of Hoare [14] and Reynolds [29], under the direction set by Plotkin and Power [28].

Before reaching for the top on the shoulders of these giants, let us warm up with a classical monadic verification problem due to Hutton and Fulger [15] involving labelled binary trees

```
Inductive Tree (X: Type) :=
| Leaf: X → Tree X
| Node: Tree X → Tree X → Tree X.
```

The challenge consists in implementing a function `label: Tree X → Tree nat` that labels every leaf with a fresh symbol, here a natural number. In order to implement this relabeling procedure in Coq, we are naturally led to define the following variant of the state monad [26]:

```
Definition Fresh X := nat → X * nat.

Definition ret (x: X): Fresh X :=
  fun n ⇒ (x, n).
Definition bind (m: Fresh X)(f: X → Fresh Y): Fresh Y :=
  fun n ⇒ let (x, n') := m n in f x n'.
Definition gensym (tt: unit): Fresh nat :=
  fun n ⇒ (n, 1+n).

Notation "'do' x '←' e1 ';' e2" := (bind e1 (fun x ⇒ e2)).
```

Tree relabeling is then the straightforward imperative program one would have written in any ML-like language:

```
Fixpoint label {X} (t: Tree X): Fresh (Tree nat) :=
  match t with
  | Leaf _ ⇒
    do n ← gensym tt;
    ret (Leaf n)
  | Node l r ⇒
    do l ← label l;
    do r ← label r;
    ret (Node l r)
  end.
```

The function `label` is correct if the structure of the tree is preserved and each leaf stores a unique number. Setting aside the question of preserving the tree structure, Hutton and Fulger [15] offered the following formal specification for the latter property:

```
Lemma label_spec : ∀ t n ft n',
    label t n = (ft, n') → n < n' ∧ flatten ft = interval n (n'-1).
```

where `flatten` accumulates each leaf value during a left-to-right traversal and `interval a b` computes the list of integers in the interval $[a, b]$. Note that this specification is extremely prescriptive as it requires that `label` consecutively numbers the leaves of the tree from the initial state n of the fresh name generator to its final state n' in a left-to-right fashion.

It is easy to deduce the absence of duplicates, captured by the `NoDup` predicate in Coq standard library:

```
Definition relabel (t: Tree X): Tree nat := fst (label t 0).

Lemma relabel_spec : ∀ t ft, relabel t = ft → NoDup (flatten ft).
```

which makes for a reasonable public API to expose, unlike the property established by `label_spec`. The correctness of relabeling rests on our ability to prove `label_spec`. To do so, it is obviously possible to treat `label` as a pure function (since it is one, after all) and therefore directly manipulate the functional encoding of our variant of the state monad. For example, to reason about a sequence of operations, we would use the inversion lemma

```
Remark bind_inversion: ∀ m f y n1 n3,
    (do x ← m; f x) n1 = (y, n3) →
    ∃ v n2, m n1 = (v, n2) ∧ f v n2 = (y, n3).
```

that reifies, through an existential, the intermediate state that occurs between the first and second operation, thus allowing us to reason piece-wise about the overall program.

Here, the proof proceeds by induction over the tree `t`. For instance, in the `Node` case, we are given the hypothesis

```
(do l ← label t1;
 do r ← label t2;
 ret (Node l r)) n = (t', n')
```

which we invert twice using `bind_inversion` so as to reveal the intermediate states `n2`, `n3` and intermediate results `t1'`, `t2'`:

- `label t1 n = (t1', n2)`
- `label t2 n2 = (t2', n3)`
- `Node t1' t2' = t'`
- `n3 = n'`

We can then proceed by induction over the first two hypothesis in order to deduce `flatten t1 = interval n (n2-1)` (with `n < n2`) on the one hand and `flatten t2 = interval n2 (n3-1)` (with `n2 < n3`) on the other hand. Properties of intervals allow us to deduce that `flatten (Node t1 t2) = interval n n'`, which establishes the desired invariant. The resulting proof is thus a back-and-forth between reasoning steps related to the monadic structure of the program (for example, `bind_inversion` above) and reasoning steps related to the invariants preserved by the program (for example, concatenating intervals above).

In order to decouple the monadic structure (whose role is to sequentialize effects) from specific interpretations of this structure (which defines its admissible semantics), one can follow the mantra of the algebraic presentations of effects [28]: start with syntax (by means of signatures) and obtain monads. In Coq, we can easily give the term algebra corresponding to the `Fresh` monad using the folklore free monad construction [19]:

```
Inductive FreeFresh X :=
| ret : X → FreeFresh X
| gensymOp : unit → (nat → FreeFresh X) → FreeFresh X.

Fixpoint bind (m: FreeFresh X)(f: X → FreeFresh Y): FreeFresh Y :=
  match m with
  | ret v ⇒ f v
  | gensymOp _ k ⇒ gensymOp tt (fun n ⇒ bind (k n) f)
  end.

Definition gensym (tt: unit): FreeFresh nat := gensymOp tt (@ret nat).
```

In effect, we are defining a *syntax* for an embedded imperative language (sequenced through the `bind` construct) featuring all Coq values (through the `ret` constructor) as well as a `gensym` operator. To give a semantics to this language, an avid Coq programmer would claim that an interpreter is as good a denotational semantics as anything else:

```
Fixpoint eval (m: FreeFresh X): nat → X * nat :=
  match m with
  | ret v ⇒ fun n ⇒ (v, n)
  | gensymOp _ k ⇒ fun n ⇒ eval (k n) (1 + n)
  end.
```

Alternatively, a zealous disciple of Dijsktra (who may well be his grand nephew [35]) would perhaps give a semantics based on predicate transformers, using for example a weakest-precondition calculus:

```
Fixpoint wp (m: FreeFresh X)(Q: X → nat → Prop): nat → Prop :=
  match m with
  | ret v ⇒ fun n ⇒ Q v n
  | gensymOp _ k ⇒ fun n ⇒ wp (k n) Q (1+n)
  end.
```

To get them to come to an agreement, we would prove the adequacy of both semantics:

```
Lemma adequacy: ∀ m Q n n' v,
    wp m Q n → eval m n = (v, n') → Q v n'.
```

Whilst we have argued against reasoning directly about the semantics of monadic programs (which amounts to `eval m` here), the adequacy lemma gives us an opportunity to switch to a more predicative reasoning style. In particular, Hoare triples [14], dear to the heart of imperative programmers, can be obtained through a simple notational trick

```
Notation "{{ P }} m {{ Q }}" := (∀ n, P n → wp m Q n)
```

from which we can readily prove the usual rules of Hoare logic [27]

```
Lemma rule_value: ∀ Q v,
    (*-----------------------*)
    {{ Q v }} ret v {{ Q }}.

Lemma rule_composition: ∀ m f P Q R,
    {{ P }} m {{ Q }} →
    (∀ v, {{ Q v }} f v  {{ R }}) →
    (*-------------------------------*)
    {{ P }} do x ← m; f x {{ R }}.

Lemma rule_gensym: ∀ k,
    (*-------------------------------------------------------*)
    {{ fun n ⇒ n = k }} gensym tt {{fun v n' ⇒ v = k ∧ n' = 1+k}}.

Lemma rule_consequence: ∀ P P' Q Q' m,
    {{ P' }} m {{ Q' }} →
    (∀ n, P n → P' n) →
    (∀ x n, Q' x n → Q x n) →
    (*-----------------------*)
    {{ P }} m {{ Q }}.
```

or, put otherwise, we obtain a shallow embedding of Hoare logic within the logic of Coq.

While, syntactically, the code of `label` is unchanged, it is now a mere abstract syntax tree. Accordingly, the correctness lemma is naturally expressed as a Hoare triple:

```
Lemma label_spec: ∀ t k,
    {{ fun n ⇒ n = k }}
      label t
    {{ fun ft n' ⇒ k < n' ∧ flatten ft = interval k (n'-1) }}.
```

This specification remains unsatisfactory: we have still over-specified the behavior of a counter whereas, *in fine*, we are only ever interested in the property `NoDup (flatten t)`. To prove it, we only need the assurance that every call to `gensym tt` produce a number distinct from any previous call (which is indeed verified by an implementation that produces consecutive numbers but this is an implementation detail).

In the remaining of this article, we argue that separation logic [29] is the perfect vehicle for this kind of specification. Our plan is to unleash the power of the wonderful ecosystem created by the MoSel [17] (and, by extension, Iris [16]) – initially introduced to model and reason about fine-grained models of concurrent systems and languages– to bear on the verification of our monadic programs. Our contributions are the following:

- We instantiate the MoSel framework (Section 2) with a custom logic to reason (exclusively) about freshness over monadic programs. The result is a tailor-made program logic embedded within Coq supporting modular reasoning about freshness, MoSel offering a wonderful environment to harness this flexibility;
- We resume our formalization of `relabel` in this framework (Section 3) and highlight the key point of the methodology;
- We offer a larger case study (Section 4) by porting the `SimplExpr` module of CompCert [18] to our framework. This module extensively relies on a monad offering a fresh name generator together with non catchable exceptions. Crucially, we show that separation logic can be used locally while the resulting theorems can be integrated in a larger (pre-existing) development standing solely in `Prop`.

Our Coq development is available online[1]. The symbol [🐓] in the electronic version of the paper will lead the reader to the corresponding source code.

2 Supporting Modular Specifications [🐓]

Separation logic [29] prominently features a *frame* rule that enables modular reasoning about properties supporting a notion of *disjointedness*. This is particularly relevant for freshness: we naturally expect to be able to reason separately about two programs producing fresh identifiers, without interference. We now formalize this intuition by instantiating the MoSel [17] framework with a minimalist separation logic to reason about generated symbols.

The type of assertions `hprop` corresponds to predicates over finite sets[2] of identifiers:

```
Definition hprop := gset ident → Prop.
```

Through this definition, `hprop` inherits the logical apparatus of `Prop` (through pointwise lifting): existential quantification, universal quantification, conjunction, *etc.* This also includes any Coq propositions `P`, called *pure* propositions and written `⌜ P ⌝`

```
Definition hpure (P : Prop) : hprop := fun _ ⇒ P.
```

The defining feature of a separation logic is the presence of a *separating conjunction*

```
Definition hstar (P1 P2 : hprop) : hprop := fun idents ⇒
  ∃ ids1 ids2, P1 ids1 ∧ P2 ids2 ∧ ids1 ## ids2 ∧ idents = ids1 ∪ ids2.
```

[1] `https://github.com/Artalik/CompCert/tree/ITP`
[2] Implemented by the `gset` type in the `Coq-std++` library [23]

that splits a given set of identifiers `idents` in a two sets `ids1` and `ids2` that are distinct (`ids1 ## ids2`), form a partition of `idents` (`idents ≡ ids1 ∪ ids2`), each satisfying its respective predicate. Unlike standard conjunction (where both propositions must hold for the *whole* set of identifiers), the separating conjunction translates the independence of both predicates by extracting two independent subsets of identifiers. Dually, the *separating implication*, written `P1 -∗ P2`, amounts to the predicate

```
fun ids1 ⇒ ∀ ids2, ids1 ## ids2 ∧ P1 ids2 → P2 (ids1 ∪ ids2).
```

and consists, intuitively, in offering `P2` provided that one can extend the existing set of identifiers so as to satisfy `P1`.

The assertion `emp = fun idents ⇒ idents = ∅` states that no identifier has been generated. We can also assert the freshness of an identifier `ident` (written `& ident`) by stating that it is the sole identifier in the supporting set

```
Definition hsingle ident : hprop := fun idents ⇒ idents = {[ ident ]}.
```

and, more generally, the operator `&& h` states that the set of identifiers amounts precisely to the identifiers in `h`. The interplay between the separating connectives and this characterization of freshness allows us to prove the absence of duplicates, such as the following instrumental lemma[3]:

```
Lemma singleton_neq : ∀ l l', ⊢ & l -∗ & l' -∗ ⌜l ≠ l'⌝.
```

From such an algebra of logical connectives, we instantiate the MoSel [17] framework. As a result, we obtain a full-featured interactive environment for reasoning about and manipulating statements in the corresponding separation logic. MoSel introduces the type `iProp` of (suitably-encoded) separation logic assertions, which subsumes `hprop` and its connectives. The relationship between the separation logic and `Prop` is preserved through a (somewhat more noisy) characterization

```
Lemma equivalence (P: iProp) idents: P () idents ↔ (⊢ && idents -∗ P).
```

3 Monadic Proof in Separation Logic [🐓]

Equipped with a separation logic, we can redefine our weakest precondition calculus to take advantage of the added structure

```
Fixpoint wp (m: FreeFresh X)(Q: X → iProp): iProp :=
  match m with
  | ret v ⇒ Q v
  | gensymOp _ k ⇒ ∀ (v: ident), & v -∗ wp (k v) Q
  end.
```

from which we naturally derive Hoare triples and their associated logic [9] as a shallow embedding

```
Notation "{{ P }} m {{ v ; Q }}" := (P -∗ wp m (fun v ⇒ Q))
```

```
Lemma rule_gensym : ⊢ {{ emp }} gensym tt {{ ident; & ident }}.
```

[3] The infix operator ⊢ embeds assertions expressed in the internal separation logic into the ambiant logic of Coq Propositions.

```
Lemma rule_consequence: ∀ P P' Q Q' m,

  (⊢{{ P' }} m {{ v; Q' v }}) →
  (P ⊢ P') →
  (∀ v, Q' v ⊢ Q v) →
 (*------------------------*)
  ⊢{{ P }} m {{ v; Q v }}.

Lemma frame: ∀ P Q P' m,

  (⊢{{ P }} m {{ v; Q v }}) →
  (*-----------------------------*)
  ⊢{{ P ∗ P' }} m {{ v; Q v ∗ P' }}.
```

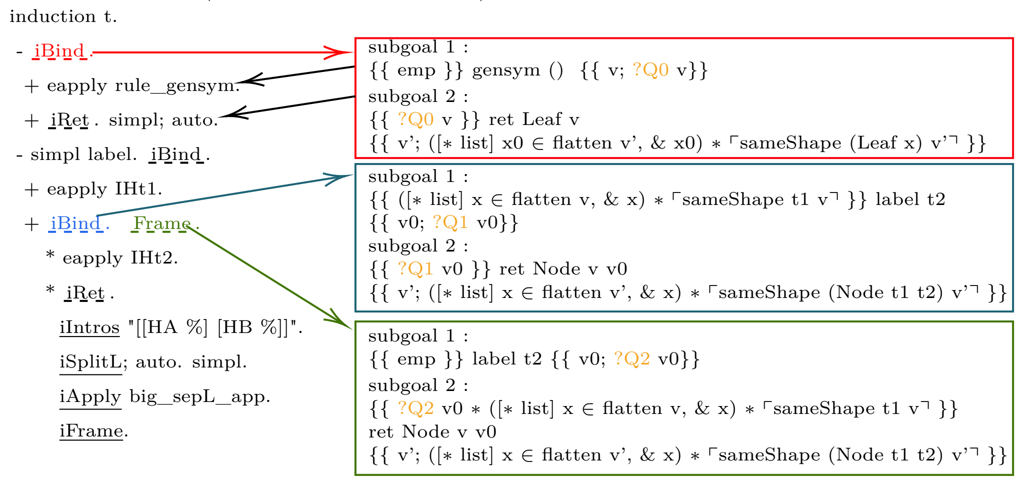
while the statement of the earlier lemmas `rule_value` and `rule_composition` remains essentially unchanged (but their signification did change!).

We are now able to specify `label` by actively exploiting the separating conjunction[4]:

```
Lemma label_spec_aux : ∀ t,
    ⊢ {{ emp }}
        label t
      {{ ft; ([∗ list] x ∈ (flatten ft), & x) ∗ ⌜sameShape t ft⌝ }}.
```

Through this move to separation logic, we have discharged the handling of freshness down to the logic, which conveniently provides us with the frame rule (`rule_frame`) to abstract over disjoint sets of identifiers. The proof of `label_spec_aux` is thus significantly simpler and consists only in *local* invariants. This is in stark contrast with our earlier proof in Section 1, where we had to maintain a global invariant across the whole execution of the program.

Thanks to MoSel, the proof script now sums up to the following instructions, which are almost intelligible. The MoSel framework provides the underlined tactics, which we extended with custom tactics (underlined with dashes) specifically manipulating the Hoare triples:



```
induction t.
 - iBind.
   + eapply rule_gensym.
   + iRet . simpl; auto.
 - simpl label.  iBind .
   + eapply IHt1.
   + iBind .   Frame .
      * eapply IHt2.
      * iRet .
        iIntros "[[HA %] [HB %]]".
        iSplitL; auto. simpl.
        iApply big_sepL_app.
        iFrame.
```

```
subgoal 1 :
{{ emp }} gensym ()  {{ v; ?Q0 v}}
subgoal 2 :
{{ ?Q0 v }} ret Leaf v
{{ v'; ([∗ list] x0 ∈ flatten v', & x0) ∗ ⌜sameShape (Leaf x) v'⌝ }}
```

```
subgoal 1 :
{{ ([∗ list] x ∈ flatten v, & x) ∗ ⌜sameShape t1 v⌝ }} label t2
{{ v0; ?Q1 v0}}
subgoal 2 :
{{ ?Q1 v0 }} ret Node v v0
{{ v'; ([∗ list] x ∈ flatten v', & x) ∗ ⌜sameShape (Node t1 t2) v'⌝ }}
```

```
subgoal 1 :
{{ emp }} label t2 {{ v0; ?Q2 v0}}
subgoal 2 :
{{ ?Q2 v0 ∗ ([∗ list] x ∈ flatten v, & x) ∗ ⌜sameShape t1 v⌝ }}
ret Node v v0
{{ v'; ([∗ list] x ∈ flatten v', & x) ∗ ⌜sameShape (Node t1 t2) v'⌝ }}
```

[4] The notation `([* list] x in l, P x)` asserts that every element `x` of the list `l` satisfies the predicate `P`. In the present case, we state that all the elements in the flattened tree are fresh.

In the leaf case, the proof essentially boils down to applying `rule_gensym`. The power of the approach strikes in the node case, where we gain access to the recursive cases through the composition rule, at which point the proof is over: the frame rule allows us to automatically combine the results of both sub-calls.

However, at this stage, we only have a proof in `iProp` while our users are expecting a pure Coq proposition, living in `Prop`. We can first narrow the gap between the two worlds by showing that the non-pure post-condition of `label_spec_aux` amounts to a pure one

```
∀ idents, ⊢ ([∗ list] i ∈ idents, & (i: ident)) -∗ ⌜NoDup idents⌝.
```

and, consequently, we obtain a specification with a pure post-condition

```
Lemma label_spec: ∀ t,
   ⊢ {{ emp }} label t {{ ft; ⌜NoDup (flatten ft) ∧ sameShape t ft⌝ }}.
```

The gap is finally bridged through an adequacy lemma, relating the execution of monadic programs with the generator set to 0

```
Definition run (m: FreeFresh X): X := fst (eval m 0).
```

with pure post-conditions obtained in the separation logic

```
Lemma adequacy : ∀ {X} {m: FreeFresh X} {Q},
    (⊢ {{ emp }} m {{ v; ⌜Q v⌝ }}) →
    Q (run m).
```

As a corollary, we obtain a publicly-usable relabeling function together with a specification expressed at a suitable level of detail:

```
Definition relabel (t: Tree X): Tree nat := run (label t).
```

```
Lemma relabel_spec : ∀ t ft,
    relabel t = ft →  NoDup (flatten ft) ∧ sameShape t ft.
```

4 Case study: `SimplExpr`

To evaluate our approach, we tackle a pre-existing certified program, namely the `SimplExpr` module of the CompCert certified compiler. This module implements a simplification phase over C expressions, pulling side-effects out of expressions and fixing an evaluation order. In the following, we offer a side-by-side comparison of the original specification with ours, exploiting separation logic (Section 2) to reason about freshness. We first materialize the underlying monad in Section 4.1 together with its dynamic and predicate transformer semantics. We then delve into the benefits of having a rich logic of assertions (Section 4.2) to carry the proofs. We finally demonstrate how these properties can then be translated to and interact with pure Coq propositions (Section 4.3), so as to be usable in the correctness proof of the whole compiler.

4.1 The monad

As for our introductory example, we crucially rely on a syntactic description of the monad `mon` used by the `SimplExpr` module. This monad, which has received some attention in the literature [34], exposes two operations: an `error e` operator, to report a run-time error `e`; a `gensym ty` operator, to generate a fresh symbol associated with a type `ty`, and a `trail` operator, to get the association list of identifiers to types constructed thus far.

Following the usual free monad construction, we reify this interface through a datatype:

```
Inductive mon (X : Type) : Type :=
| ret : X → mon X
| errorOp : Errors.errmsg → mon X
| gensymOp : type → (ident → mon X) → mon X
| trailOp : unit → (list (ident * type) → mon X) → mon X.

Definition error {X} (e : Errors.errmsg) : mon X := errorOp e.
Definition gensym (t : type) : mon ident := gensymOp t ret.
Definition trail (_ : unit): mon (list (ident * type)) := trailOp tt ret.
```

The definition of the monadic bind follows naturally. As before, we will use the user-friendly notation `do _ ← _ ; _` in code.

Note that an `error` does not require a continuation: at run-time, it corresponds to an uncatchable exception. It is used by the compiler to abort when some input program falls outside the semantic domain of C (delineated by the mechanized semantics given by CompCert).

The dynamic semantics of `mon` is slightly richer than the one of `FreeFresh` (Section 1). First, we must handle the addition of an uncatchable error during execution. We piggy-back on CompCert's implementation of the error monad

```
Inductive res (A: Type) : Type :=
| OK: A → res A
| Error: errmsg → res A.
```

and, essentially, inline the usual error monad transformer over the state monad necessary to maintain the internal state of the `gensym` operator. However, unlike earlier, `gensym` now associates fresh identifiers with their provided type. This is reflected in the semantics, which maintains an association list of ident and types together with the next fresh ident:

```
Record generator : Type := mkgenerator { gen_next : ident;
                                         gen_trail: list (ident * type) }.
```

The dynamic semantics amounts to the usual interpretation of errors in `res` and stateful operations in `generator → M (generator * X)`:

```
Fixpoint eval {X} (m : mon X) : generator → res (generator * X) :=
  match m with
  | ret v ⇒ fun s ⇒ OK (s, v)
  | errorOp e ⇒ fun s ⇒ Error e
  | gensymOp ty f ⇒
    fun s ⇒
      let h := gen_trail s in
      let n := gen_next s in
      eval (f n) (mkgenerator (n+1) ((n,ty) :: h))
  | trailOp _ f ⇒
    fun s ⇒
      let h := gen_trail s in
      eval (f h) s
  end.
```

The compiler pass is ran with an `initial_generator` that is provided from the OCaml driver, remaining opaque to Coq until after extraction:

```
Definition run {X} (m: mon X): res X :=
  match eval m (initial_generator tt) with
  | OK (_, v) ⇒ OK v
  | Error e ⇒ Error e
  end.
```

The predicate transformer semantics is given by a straightforward weakest-precondition calculus:

```
Fixpoint wp {X} (e1 : mon X) (Q : X → iProp) : iProp :=
  match e1 with
  | ret v ⇒ Q v
  | errorOp e ⇒ True
  | gensymOp _ f ⇒ ∀ l, & l -∗ wp (f l) Q
  | trailOp _ f ⇒ ∀ l, wp (f l) Q
  end.
```

where the semantics of `gensym` follows exactly our earlier definition. The semantics of `error` does not require any precondition (but, as we shall see in the adequacy lemma, this also means that our post-conditions are only true *if* the compiler did not raise an error). The specification of `trail` is purposefully non-committal: CompCert does not make any assumption about the output of `trail` (in a rather elegant twist, the fact that the identifiers produced by `trail` are all distinct is a decidable property that is checked at run-time in a later compilation pass: `trail` is indeed free to return any list of identifiers but CompCert will simply refuse to compile a piece of code triggering an invalid output.)

As in Section 1, we derive Floyd-Hoare triples `{{ P }} m {{ v; Q }}` from our weakest-precondition calculus, together with the usual structural rules. The monad-specific operators are specified as follows:

```
Lemma rule_gensym ty : ⊢{{ emp }} gensym ty {{ ident; & ident }}.
```

```
Lemma rule_error Q e: ⊢{{ True }} error e {{ v; Q v }}.
```

```
Lemma rule_trail : ⊢{{ emp }} trail tt {{ _; emp  }}.
```

In particular, the operator `error` amounts to a "get out of proof free card", allowing us to discharge any post-condition by refusing to do any work. We relate the dynamic and predicate transformer semantics through an adequacy lemma

```
Lemma adequacy: ∀ m Q v,
   (⊢ {{ emp }} m {{ v; ⌜ Q v ⌝}}) →
   run m = OK v →  Q v.
```

that only proves the post-condition when the evaluation succeeds in producing a value.

4.2 Proofs and Programs [🐓]

The expression simplification pass is part of the CompCert front-end. It consists of 3 files: `cfrontend/SimplExpr.v` (which contains the monadic programs), `cfrontend/SimplExprspec.v` (which contains a Prolog-like specification of the monadic programs through inductive relations, as well as the proof relating the monadic programs to their

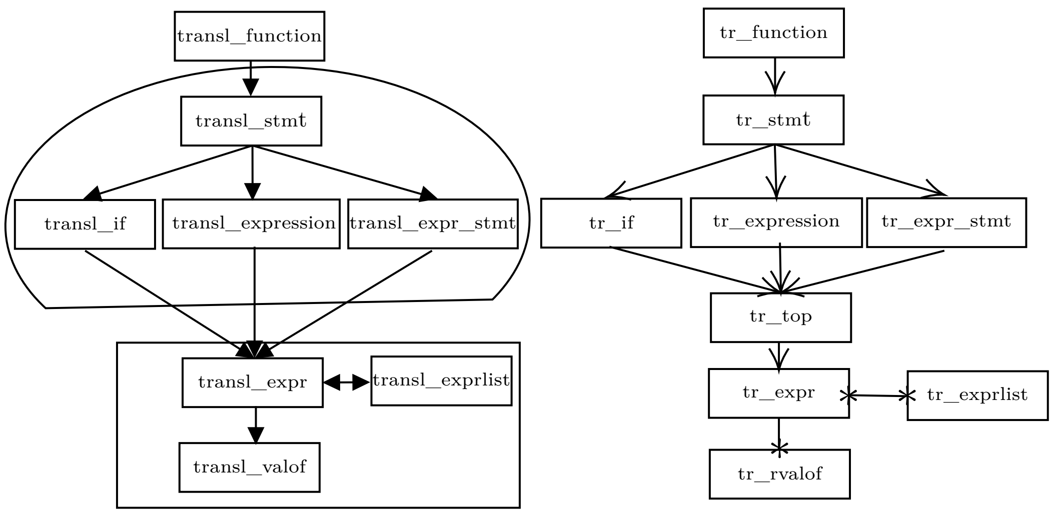


Figure 1 `SimplExpr` call graph (left) and the corresponding specifications (right).

specification) and `cfronted/SimplExprproof.v` (which contains the proof of correctness of the compilation pass, exploiting the relational specifications). Syntactically, `cfrontend/SimplExpr.v` is left unchanged when we swap in our monad: we were careful to implement the same interface as the previous one. However the semantics is very different: whereas the previous monad was building an actual computation, ours is just building an abstract syntax tree. We therefore need to add suitable call to `run` to turn this syntax into an actual computation.

We give an overview of the `SimplExpr` module through its call graph (Figure 1). The raison d'etre of this module is to define `transl_function: Csyntax.function` → `res function` that performs the simplification over functions. This is the (only) entry-point into the error monad `res`. It hosts the call `run`. `transl_function` recursively depends on a host of helpers operating in the `error` and `trail` fragment of the monad, grouped in the circular frame (Figure 1). Crucially, none of the functions invoke a fresh symbol generator themselves. A third group of functions, all dispatched from `transl_expr` and collected in the rectangular frame (Figure 1), consists of those functions that actually generate fresh symbols and must therefore belong to the full-fledged monad `mon`.

In the following, we present several programs extracted or modified from CompCert, together with their specifications. In those, aspects related to the freshness of names is a means toward an overall correctness result. Consequently, programs and specifications involve a backbone of operations and properties dealing with freshness, fleshed out with further transformations and properties implementing the desired compilation pass. In order to see the forest (of freshness) for the trees, we adapt a typographical legerdemain: we typeset in a tiny font size the parts of the program and proof that do not involve freshness. As part of our work, we were led to replace definitions from the original CompCert with new ones: when recalling the original, we display it on a gray background to set it apart.

Let us begin our exploration of the `SimplExpr` module through `transl_expr`, which involves both fresh name generation and errors

```
Fixpoint transl_expr (dst: destination) (a: Csyntax.expr) : mon (list statement * expr)
```

Its argument `dst` may wrap, in the `For_set` case, an identifier within a value of type `set_destination`

```
Inductive set_destination : Type :=
  | SDbase (tycast ty: type) (tmp: ident)
  | SDcons (tycast ty: type) (tmp: ident) (sd: set_destination).

Inductive destination : Type :=
  | For_val
  | For_effects
  | For_set (sd: set_destination).
```

The type `destination` specifies how to pass along the result of a given expression, *i.e.* whether the contribution of an expression lies in its returned value, or solely in its side effects, or in a temporary variable in which its denotation has been saved.

For correctness of this optimization pass, it is crucial that this identifier is fresh with respect to any identifier that `transl_expr` may produce. The function `transl_expr` itself is defined by pattern-matching over the source AST, we focus here on the assignment case:

```
| Csyntax.Eassign l1 r2 ty ⇒
  do (sl1, a1) ← transl_expr For_val l1;
  do (sl2, a2) ← transl_expr For_val r2;
  let ty1 := Csyntax.typeof l1 in
 let ty2 := Csyntax.typeof r2 in
  match dst with
  | For_val | For_set _ ⇒
    do t ← gensym ty1;
    ret (finish dst
             (sl1 ++ sl2 ++ Sset t (Ecast a2 ty1) ::
                make_assign a1 (Etempvar t ty1) :: nil)
             (Etempvar t ty1))
  | For_effects ⇒
    ret (sl1 ++ sl2 ++ make_assign a1 a2 :: nil,
         dummy_expr)
  end
```

It performs two recursive calls with destinations that do not involve fresh identifiers (`For_val`). However, when its own destination is a value (`For_val`) or an assignment (`For_set`), it also performs a call to `gensym`. The specification needs to reflect the fact that the identifiers generated by the recursive calls are distinct between each other and distinct from the identifier potentially generated in the assignment case. In CompCert, this is achieved by explicitly threading the lists (in this case, `tmp`, `tmp1` and `tmp2`) of identifiers generated and asserting their disjointness:

```
Inductive tr_expr: temp_env → destination → Csyntax.expr → list statement → expr →
                   list ident → Prop :=

  | tr_assign_val: ∀ le dst e1 e2 ty sl1 a1 tmp1 sl2 a2 tmp2 t tmp ty1 ty2,
      tr_expr le For_val e1 sl1 a1 tmp1 →
      tr_expr le For_val e2 sl2 a2 tmp2 →
      incl tmp1 tmp → incl tmp2 tmp →
      list_disjoint tmp1 tmp2 →
```

```
    In t tmp → ~In t tmp1 → ~In t tmp2 →
    ty1 = Csyntax.typeof e1 →
    ty2 = Csyntax.typeof e2 →
    tr_expr le dst (Csyntax.Eassign e1 e2 ty)
           (sl1 ++ sl2 ++
            Sset t (Ecast a2 ty1) ::
            make_assign a1 (Etempvar t ty1) ::
            final dst (Etempvar t ty1))
           (Etempvar t ty1) tmp
```

In order to express the precondition on `transl_expr`, stating that any potential identifier in `dst` is fresh, CompCert introduces the following predicate

```
Definition sd_temp (sd: set_destination) :=
  match sd with SDbase _ _ tmp ⇒ tmp | SDcons _ _ tmp _ ⇒ tmp end.

Definition dest_below (dst: destination) (g: generator) : Prop :=
  match dst with
  | For_set sd ⇒ Plt (sd_temp sd) g.(gen_next)
  | _ ⇒ True
  end.
```

that, in a very operational manner, asserts that the identifiers stored in `dst` occurred earlier in the execution of the fresh name generator and are therefore distinct from any future identifier (since they are produced as consecutive numbers).

Having access to a notion of freshness in our language of assertions, we can prevent these operational details from leaking out and simply assert that such an identifier must be fresh:

```
Definition dest_below (dst: destination) : iProp :=
  match dst with
  | For_set sd ⇒ & (sd_temp sd)
  | _ ⇒ emp
  end.
```

The implementation of `transl_expr` is then abstracted away thanks to the relational specification given by `tr_expr` as follows

```
Lemma transl_meets_spec:
   (∀ r dst g sl a g' I,
    transl_expr dst r g = Res (sl, a) g' I →
    dest_below dst g →
    ∃ tmps, (∀ le, tr_expr le dst r sl a (add_dest dst tmps)) ∧
            contained tmps g g')
```

where `g` and `g'` represent the state of the fresh name generator at the beginning and, respectively, the end of the transformation. These are necessary to assert that any ident in `dst` is indeed fresh (through `dest_below`) and that the temporaries produced by `transl_expr` will not conflict with any earlier or later use of the generator (through `contained tmps g g'`, which guarantees that all the identifiers in `tmps` were produced between `g` and `g'`).

In our setting, the freshness of the identifiers produced in the subcalls and of the locally generated identifier is captured with separating conjunctions:

```
Fixpoint tr_expr (le : temp_env) (dst : destination) (e : Csyntax.expr)
          (sl : list statement ) (a : expr)  : iProp :=

  | Csyntax.Eassign e1 e2 ty ⇒
    match dst with
    | For_val | For_set _ ⇒
      ∃ sl2 a2 sl3 a3 t,
       tr_expr le For_val e1 sl2 a2 ∗
       tr_expr le For_val e2 sl3 a3 ∗
       & t ∗
       dest_below dst ∗
       ⌜ sl = sl2 ++ sl3 ++ Sset t (Ecast a3 (Csyntax.typeof e1)) ::
            make_assign a2 (Etempvar t (Csyntax.typeof e1)) ::
            final dst (Etempvar t (Csyntax.typeof e1)) ∧
    a = Etempvar t (Csyntax.typeof e1)⌝
```

Similarly, the relationship between `transl_expr` and `tr_expr` is now straightforward, the constraint that `dst` is fresh with respect to the identifiers produced by `transl_expr` being naturally expressed through a separating implication

```
Lemma transl_meets_spec :
  (∀ r dst,
     ⊢ {{ emp }} transl_expr dst r
       {{ res; dest_below dst −∗ ∀ le, tr_expr  le  dst  r res.1 res.2  }})
```

Through this process, we have entirely removed the painstaking need to track the operational state of the name generators and maintain global invariants about the relative freshness of program fragments. Doing so, we have elevated our specification and successfully decoupled it for the operational aspects of generating fresh identifiers. As an added bonus, we can now rely on MoSel to prove that our implementation meets its specification. In practice, we observe that the length of the proof scripts is divided by two when moving to MoSel but we shall resist from the temptation of drawing any conclusion from such an unreliable metric.

4.3 Leaving `iProp` [🐓]

Reasoning about freshness occurs only in the group of functions below `transl_expr` in the call graph. For the functions (and their respective specifications) above `transl_expr`, the set of fresh identifiers ranged over by the specification is always existentially quantified. Since, by construction, `iProp` is isomorphic to `gset ident → Prop` (Section 2), we have integrated this discipline in a wrapper-specification

```
Inductive tr_top: destination → Csyntax.expr → list statement → expr →  Prop :=

| tr_top_base: ∀ dst r sl a tmp,
    tr_expr le dst r sl a () tmp →
    tr_top dst r sl a.
```

As a consequence, functions above `transl_expr` do not need to propagate freshness invariants. As a result, `Prop` is a sufficient vehicle to write their specifications. However, to show that these functions satisfy their specifications, we took on ourselves to port the proofs to MoSel as well. For example, the function `transl_stmt`, which translates statements, is specified as follow in our setting

```
Lemma transl_stmt_meets_spec : ∀ s,
    ⊢ {{ emp }} transl_stmt s {{ res; ⌜ tr_stmt s res ⌝}}
```

which is merely a iota away from the original

```
Lemma transl_stmt_meets_spec:
  ∀ s g ts g’ I, transl_stmt s g = Res ts g’ I → tr_stmt s ts
```

While a purely cosmetic change, this has allowed us to streamline the proofs, which were designed around inversion lemmas over the monadic structure (themselves wrapped in tactics). Note that this effort was not strictly necessary: we could have kept the pre-existing definitions and their proofs.

To restore the overall compiler correctness proof [🐓], we must re-establish a simulation lemma relating source and target programs. This work is carried solely over the specifications of the various functions (right-hand side of Figure 1). Above `tr_top` (included), the specific-ations lives in `Prop` so the proofs remain unchanged. For `tr_expr`, where the specification lives in `iProp`, we resort to reasoning in separation logic: we have therefore updated the original predicates so as to fully exploit the separating connectives to handle freshness. We carry this part of the simulation proof in MoSel. To bridge the gap between `iProp` and `Prop`, which occurs when we go through `tr_top`, we resort to lemmas such as `singleton_neq` (Section 2) that translates freshness assertions into propositional facts.

5 Related Work

Early on, dependent type theory was used to develop various models of Hoare logic [25, 30], including several ones based on separation logic [24, 8, 16]. However, these formalisms were introduced to reason about *models* of imperative or concurrent programs: type theory was not yet recognized as a vehicle for writing effectful programs. CompCert was instrumental in showing that non-trivial effectful programs could be written within a proof assistant. This inspired the work of Swiertra [34], aiming at rationalizing and generalizing the indexed state monad construction introduced by Leroy specifically in `SimplExpr`.

The Dijkstra Monad [13, 31, 3, 2, 32] research program, spearheaded by Swamy and collaborators, has demonstrated that effectful programming has its place in the context of certified programming in F*. On their journey, the designer of F* have shown the benefits of a modular approach to effects (polymonads), each equipped with a suitable program logic (Dijkstra monad) which – in some instances – could be automatically derived from the underlying monad (using an interpretation in the continuation monad). However, this line of work actively exploits the refinement-based approach to typing of F* (relying extensively on an SMT solver to decide the conversion of indices). As-is, this would be ill-fitted for a proof assistant based on dependent type theory, where conversion is not as rich and relying on functional values at the type level would make for a painful experience. Our approach is rooted in the pragmatics of indexed programming in dependent type theory and of Coq in particular. In that respect, MoSel offers the ultimate development environment for reasoning – in a natural manner – about effectful programs in Coq.

Before us, this approach has been pursued in the context of the FreeSpec project [19] in Coq. While its scope was limited to modeling and reasoning about (hardware) interfaces, FreeSpec has shown the benefits of a syntactic treatment of monads (through the free monad construction) and how to construct domain-specific logics for those through pre/post pairs. The key contribution of FreeSpec is a generic treatment of effects, which we could easily borrow to factor out our monadic constructions.

In Agda, Swierstra and Baanen [35] have shown how the FreeSpec approach (based on free monads) and the Dijsktra Monads (deriving program logics from monads) could be fruitfully combined. This results in a library of predicate transformers, operating over the syntactic model of the monad. We followed this approach to the letter, specializing our definition to the monads at hand for pedagogical purposes. Being in Coq, we also benefit from the impredicativeness of `Prop` and, by extension, `iProp`, which saves us from tiptoeing around universe stratification when defining the predicate transformer semantics.

While many of the work above is focused on emulating some form of Hoare logics in type theory, there is also a parallel and rich line of work betting on the power of equational reasoning for effectful programs. Gibbons and Hinze [12] were instrumental in illustrating – on paper – how to use algebraic presentations of monads to prove the correctness of programs implemented in those. In particular, they revisited the relabel program from Hutton and Fulger and gave a purely equational proof of correctness. Affeldt *et al.* [1] realized this vision in the Coq theorem prover, extensively relying on SSReflect [36] to enable a compositional treatment of monads and to effectively reason about monadic programs by rewriting.

Interaction Trees [37] are a middle ground between the purely equational treatment of Affeldt *et al.* and the syntactic treatment of FreeSpec. Much like FreeSpec, interaction trees are constructed from a signature of possible operations. However, the authors dispense with the free monad construction altogether and directly manipulate the free completely iterative monad, *i.e.* infinite unfoldings of the signature's control-flow graph. Program equivalence is thus proved by establishing a bisimilarity between two unfoldings: in practice, this is achieved through equational reasoning; substituting equivalent program fragments for each others. The treatment of diverging computations is worthwile and would deserve further attention in our setting.

6 Conclusion

This paper reports on an experiment: use one of the most advanced piece of technology for reasoning about imperative features – separation logic, embodied by the MoSel framework – to reason about certified monadic programs in Coq. To exercise this approach, we ported the `SimplExpr` module of CompCert to use a separation logic for reasoning about fresh names. Our version of `SimplExpr` is feature-complete and integrated in the rest of compiler pipeline. The definition of the monad and its separation logic introduce an additional 750 lines of code [🐓] (ignoring the 30 000 lines of code of Iris/MoSel). Conversely, the specifications and their proofs go from 1100 lines of code originally down to 650 lines of code [🐓]. The correctness proof stands at around a thousand lines of code [🐓].

We should be careful when interpreting these numbers, as code size is but a poor metric to judge the quality of a development. It is however clear that, while certainly encouraging, this experiment points towards developing an integrated library of monads and their operational semantics (à la FreeSpec [19] and interaction trees [37]) as well as their predicate transformer semantics (à la Dijkstra monad [3]). This effort should also be aimed at providing a library of ready-made separation logics for reasoning about common effects, which would allow us to amortize some of those 750 additional lines of code.

As far as proof engineering goes, it would be interesting to study how our proofs fare compared to the original ones when the underlying code evolves. We believe that the abstract reasoning style enabled by separation logic provides more opportunities for automation, which should smooth out the proof update process. Further experiment is required to confirm or refute this hypothesis.

References

1 Reynald Affeldt, David Nowak, and Takafumi Saikawa. A hierarchy of monadic effects for program verification using equational reasoning. In Graham Hutton, editor, *Mathematics of Program Construction - 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings*, volume 11825 of *Lecture Notes in Computer Science*, pages 226–254. Springer, 2019. `doi:10.1007/978-3-030-33636-3_9`.

2 Danel Ahman, Cédric Fournet, Catalin Hritcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. Recalling a witness: foundations and applications of monotonic state. *Proc. ACM Program. Lang.*, 2(POPL):65:1–65:30, 2018. `doi:10.1145/3158153`.

3 Danel Ahman, Catalin Hritcu, Kenji Maillard, Guido Martínez, Gordon D. Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 515–529. ACM, 2017. `doi:10.1145/3093333.3009878`.

4 Robert Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19(3-4):335–376, 2009. `doi:10.1017/S095679680900728X`.

5 Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. Strongly typed term representations in coq. *J. Autom. Reason.*, 49(2):141–159, 2012. `doi:10.1007/s10817-011-9219-0`.

6 Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 133–144, 2013. `doi:10.1145/2500365.2500581`.

7 Edwin Brady. Resource-dependent algebraic effects. In *Trends in Functional Programming - 15th International Symposium, TFP 2014, Soesterberg, The Netherlands, May 26-28, 2014. Revised Selected Papers*, pages 18–33, 2014. `doi:10.1007/978-3-319-14675-1_2`.

8 Arthur Charguéraud. Program verification through characteristic formulae. In Paul Hudak and Stephanie Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 321–332. ACM, 2010. `doi:10.1145/1863543.1863590`.

9 Arthur Charguéraud. Separation logic for sequential programs (functional pearl). *Proc. ACM Program. Lang.*, 4(ICFP):116:1–116:34, 2020. `doi:10.1145/3408998`.

10 Adam Chlipala, Benjamin Delaware, Samuel Duchovni, Jason Gross, Clément Pit-Claudel, Sorawit Suriyakarn, Peng Wang, and Katherine Ye. The end of history? using a proof assistant to replace language design with library design. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*, volume 71 of *LIPIcs*, pages 3:1–3:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPIcs.SNAPL.2017.3`.

11 Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *J. Funct. Program.*, 13(4):709–745, 2003. `doi:10.1017/S095679680200446X`.

12 Jeremy Gibbons and Ralf Hinze. Just do it: simple monadic equational reasoning. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 2–14, 2011. `doi:10.1145/2034773.2034777`.

13 Michael Hicks, Gavin M. Bierman, Nataliya Guts, Daan Leijen, and Nikhil Swamy. Polymonadic programming. In Paul Levy and Neel Krishnaswami, editors, *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014*, volume 153 of *EPTCS*, pages 79–99, 2014. `doi:10.4204/EPTCS.153.7`.

14 C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. `doi:10.1145/363235.363259`.

15 Graham Hutton and Diana Fulger. Reasoning About Effects: Seeing the Wood Through the Trees. In *Proceedings of the Symposium on Trends in Functional Programming*, Nijmegen, The Netherlands, May 2008.

16 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`.

17 Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. Mosel: a general, extensible modal framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.*, 2(ICFP):77:1–77:30, 2018. `doi:10.1145/3236772`.

18 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. `doi:10.1145/1538788.1538814`.

19 Thomas Letan and Yann Régis-Gianas. Freespec: specifying, verifying, and executing impure computations in Coq. 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 32–46. ACM, 2020. `doi:10.1145/3372885.3373812`.

20 Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. Modular verification of programs with effects and effect handlers in Coq. In *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*, pages 338–354, 2018. `doi:10.1007/978-3-319-95582-7_20`.

21 Kenji Maillard. *Principles of Program Verification for Arbitrary Monadic Effects. (Principes de la Vérification de Programmes à Effets Monadiques Arbitraires)*. PhD thesis, École Normale Supérieure, Paris, France, 2019. URL: `https://tel.archives-ouvertes.fr/tel-02416788`.

22 Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Catalin Hritcu, Exequiel Rivas, and Éric Tanter. Dijkstra monads for all. *Proc. ACM Program. Lang.*, 3(ICFP):104:1–104:29, 2019. `doi:10.1145/3341708`.

23 Iris development team. coq-std++. URL: `https://plv.mpi-sws.org/coqdoc/stdpp`.

24 Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: dependent types for imperative programs. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 229–240. ACM, 2008. `doi:10.1145/1411204.1411237`.

25 Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6):865–911, 2008. `doi:10.1017/S0956796808006953`.

26 Simon Peyton Jones. *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*, pages 47–96. IOS Press, January 2001.

27 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, 2018.

28 Gordon D. Plotkin and John Power. Adequacy for algebraic effects. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2030 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2001. `doi:10.1007/3-540-45315-6_1`.

29 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002. `doi:10.1109/LICS.2002.1029817`.

30 Christoph Sprenger and David A. Basin. A monad-based modeling and verification toolbox with application to security protocols. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern,*

Germany, September 10-13, 2007, Proceedings, volume 4732 of *Lecture Notes in Computer Science*, pages 302–318. Springer, 2007. doi:10.1007/978-3-540-74591-4_23.

31 Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 256–270. ACM, 2016. doi:10.1145/2837614.2837655.

32 Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. Steelcore: an extensible concurrent separation logic for effectful dependently typed programs. *Proc. ACM Program. Lang.*, 4(ICFP):121:1–121:30, 2020. doi:10.1145/3409003.

33 Wouter Swierstra. A hoare logic for the state monad. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 440–451. Springer, 2009. doi:10.1007/978-3-642-03359-9_30.

34 Wouter Swierstra. The hoare state monad (proof pearl), 2009.

35 Wouter Swierstra and Tim Baanen. A predicate transformer semantics for effects (functional pearl). *Proc. ACM Program. Lang.*, 3(ICFP):103:1–103:26, 2019. doi:10.1145/3341707.

36 Iain Whiteside, David Aspinall, and Gudmund Grov. An essence of ssreflect. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors, *Intelligent Computer Mathematics - 11th International Conference, AISC 2012, 19th Symposium, Calculemus 2012, 5th International Workshop, DML 2012, 11th International Conference, MKM 2012, Systems and Projects, Held as Part of CICM 2012, Bremen, Germany, July 8-13, 2012. Proceedings*, volume 7362 of *Lecture Notes in Computer Science*, pages 186–201. Springer, 2012. doi:10.1007/978-3-642-31374-5_13.

37 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.

Specifying Message Formats with Contiguity Types

Konrad Slind
Trusted Systems Group, Collins Aerospace, Minneapolis, USA

Abstract

We introduce *Contiguity Types*, a formalism for network message formats, aimed especially at self-describing formats. Contiguity types provide an intermediate layer between programming language data structures and messages, offering a helpful setting from which to automatically generate decoders, filters, and message generators. The syntax and semantics of contiguity types are defined and used to prove the correctness of a matching algorithm which has the flavour of a parser generator. The matcher has been used to enforce semantic well-formedness conditions on complex message formats for an autonomous unmanned avionics system.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Logic, verification, formal language theory, message format languages

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.30

Supplementary Material HOL4 formalization may be found at:
Model: `https://github.com/loonwerks/splat/blob/monitors/contig/abscontigScript.sml`
archived at `swh:1:cnt:62ff99d4ed11f6d7f8452467e27b8013398d3577`

Acknowledgements This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Discussions with Robby, Luke Ryon, Dave Greve, David Hardin, Magnus Myreen, and Michael Norrish have greatly improved this work.

1 Introduction

Serialized data, for example network messages, is an important component in many computer systems.[1] As a result, innumerable libraries and tools have been created that use high level specifications as a basis for automating the creation, validation, and decoding of such data. Usually, these high level specifications describe the format of a message in terms of how the elements (fields) of the message are packed side-by-side to make the full message. When the size of each field is known in advance, there are really no conceptual difficulties. However, messages can be more complicated than that.

The main source of difficulty is *self-describing* messages: those where information embedded in fields of the message determines the final structure of the message. Two of the main culprits are variable-length arrays and unions. A *variable-length array* is a field where the number of elements in the field depends on the value of some already-seen field (or, more generally, as the result of a computation involving previously-seen information in the message). The length is therefore a value determined at runtime. A *union* is deployed when some information held in a message is used to determine the structure of later portions of the message. For example, unions can be used to support versioning where version i has n fields, and version $i+1$ has $n+1$. In settings where both versions need to be supported in a single format, it can make sense to encode the version handling inside the message, and unions are how this can be specified.

[1] DISTRIBUTION STATEMENT A. Approved for public release.

$$
\begin{array}{rcl}
\mathit{base} & = & \mathsf{bool} \mid \mathsf{char} \mid \mathsf{u8} \mid \mathsf{u16} \mid \mathsf{u32} \mid \mathsf{u64} \mid \mathsf{i16} \mid \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{float} \mid \mathsf{double} \\
\tau & = & \mathit{base} \\
 & \mid & \mathsf{Recd}\ (f_1 : \tau_1) \ldots (f_n : \tau_n) \\
 & \mid & \mathsf{Array}\ \tau\ \mathit{exp} \\
 & \mid & \mathsf{Alt}\ \mathit{bexp}\ \tau\ \tau
\end{array}
$$

Figure 1 Contiguity types.

We believe that tools and techniques from formal language theory such as regular expressions, automata, grammars, parser generators, etc. can provide an effective way to tackle message formats, and have been using the acronym *SPLAT* (Semantic Properties for Language and Automata Theory) to refer to this approach. For example, we have used regular expressions as a specification language for message formats having simple interval constraints on the values allowed in fields. Generation of the corresponding DFA results in an efficient table-driven automaton implementing the specified constraints, with a solid proof certificate connecting the original constraints with the DFA behavior [5].

However, self-describing data formats fall outside the realm of common formal language techniques; e.g., variable-length fields are clearly not able to be described by regular or context-free languages. (These language classes encompass repetitions of a fixed or unbounded size, but not repetitions of a size determined by parts of the input string.) It seems that context-sensitive grammars can, in principle, specify such information, but there are few tools supporting context sensitive languages. Knuth introduced attribute grammars [8] for dealing with context-sensitive aspects of parsing, and those techniques address similar problems to ours. Another possibility would be to use *parser combinators* in order to quickly stitch together a parser; it seems likely that the combinators can be instrumented to gather and propagate contextual information. However, we are seeking a high level of formal specification and automation, while still being rooted in formal languages, with their emphasis on sets of strings as the basic notion.

2 Contiguity Types

The characteristic property of a message is *contiguity*: all the elements of the message are laid out side-by-side in a byte array (or string). Our assumption is that a message is the *result* of a map from structured data and we will rely on a basic collection of programming language types to capture that structure. Contiguity types (Figure 1) start with common base types (booleans, characters, signed and unsigned integers, etc.) and are closed under the construction of records, arrays, and unions. [2]

Notice that τ is defined in terms of a type of arithmetic expressions *exp* and also *bexp*, boolean expressions built from *exp*. Now consider

$$\mathsf{Array}\ \tau\ \mathit{exp}\ .$$

For this to specify a varying length array dependent on other fields of the message, its dimension *exp* should be able to refer to the *values* of those fields. The challenge is just how to express the concept of "other fields", i.e., we need a notation to describe the *location* in

[2] We will use the terms "*contiguity type*, contig, and τ interchangeably.

the message buffer where the value of a field can be accessed. Our core insight is that this is similar to a problem that programming language designers had in the 60s and 70s, resolved by the notions of *L-value* and *R-value*. The idea is originally due to Christopher Strachey in CPL [13] and developed subsequently, for example by Dennis Ritchie in C [12].

Before getting into formal details, we discuss a few examples. We will use familiar notation: records are lists of *filedname* : τ elements enclosed by braces; an array field Array c dim is written $c\,[dim]$; and Alt b τ_1 τ_2 is written '`if` b `then` τ_1 `else` τ_2'. "Cascaded" Alts may be written in Lisp "cond" style, i.e., as

$$\begin{array}{ll}\mathsf{Alt} & b_1 \longrightarrow \tau_1 \ldots \\ & b_n \longrightarrow \tau_n \\ & \text{otherwise} \longrightarrow \tau_{n+1}\end{array}$$

1. The following is a record with no self-describing aspects: each field is of a statically known size.

```
{A : u8
 B : {name : char [13]
      cell : i32}
 C : bool
}
```

   The `A` field is specified to be an unsigned int of width 8 bits, the `B` field is a record, the first element of which is a character array of size 13, and the second element of which is a 32 bit integer; the last field is specified to be a boolean.
2. Variable-sized strings are a classic self-describing aspect. In this example the contents of the `len` field determines the number of elements in the `elts` field.

```
{ len : u16
  elts : char [len]
}
```

3. The following example shows the Alt construct being used to support multiple versions in a single format. Messages with the value of field `versionID` being less than 14 have three fields in the message, while all others have two.

```
{versionID : u8
 versions : if versionID < 14 then
              { A : i32, B : u16})
            else { Vec : char [13]}
 }
```

4. The following is a contrived example showing the need for resolution of multiple similarly named fields; it also shows how the information needed to determine the message structure may be deeply buried in some fields.

```
{len : u16
 A : {len : u16
      elts : u16[len]
     }
 B : char [A.len - 1 * len]
 C : i32 [A.elts[0]]
}
```

$$\begin{array}{rcl} \mathit{lval} & = & \mathit{varname} \mid \mathit{lval}\,[\mathit{exp}] \mid \mathit{lval}.\mathit{fieldname} \\ \mathit{exp} & = & \mathsf{Loc}\ \mathit{lval} \mid \mathsf{nLit}\ \mathsf{nat} \mid \mathit{constname} \mid \mathit{exp} + \mathit{exp} \mid \mathit{exp} * \mathit{exp} \\ \mathit{bexp} & = & \mathsf{bLoc}\ \mathit{lval} \mid \mathsf{bLit}\ \mathsf{bool} \mid \neg \mathit{bexp} \mid \mathit{bexp} \wedge \mathit{bexp} \mid \mathit{exp} = \mathit{exp} \mid \mathit{exp} < \mathit{exp} \end{array}$$

Figure 2 L-values, expressions, and boolean expressions.

2.1 Expressions, L-values, and R-values

In programming languages, an *L-value* is an expression that can occur on the left-hand side of an assignment statement. Similarly, an *R-value* can occur on the right-hand side of assignments. Following are a few examples:

```
x := x + 1
A[x] := B.y + 42
A[x].lens.fst[7] := MAX_LEN * 1024 + B.y
```

Figure 2 presents the formal syntax for L-values, R-values, and the boolean expressions we will use. An L-value can be a variable, an array index, or a record field access. R-values are arithmetic expressions that can contain L-values (we will use *exp* interchangeably with R-value).

An L-value denotes an *offset* from the beginning of a data structure, plus a *width*. In an R-value, an occurrence of an L-value is mapped to the value of the patch of memory between *offset* and *offset* + *width*. For the purpose of specifying message formats, it may not be immediately obvious that a notation supporting assignment in imperative languages can help, but there is indeed a form of assignment lurking.

The above explanation of L-values centers on indices into a byte buffer; in the following we will give a mild variant of this: instead of indices into the buffer, we lift out the designated slices. Thus, given environments $\theta : \mathit{lval} \mapsto \mathsf{string}$ (binding L-values to strings), and functions $\mathsf{toN} : \mathsf{string} \to \mathbb{N}$ and $\mathsf{toB} : \mathsf{string} \to \mathsf{bool}$ (which interpret byte sequences to numbers and booleans, respectively), expression evaluation and boolean expression evaluation have conventional definitions:

$$\mathsf{evalExp}\ \theta\ e = \mathsf{case}\ e \begin{cases} \mathsf{Loc}\ \mathit{lval} & \Rightarrow\ \mathsf{toN}(\theta(\mathit{lval})) \\ \mathsf{nLit}\ n & \Rightarrow\ n \\ e_1 + e_2 & \Rightarrow\ \mathsf{evalExp}\ \theta\ e_1 + \mathsf{evalExp}\ \theta\ e_2 \\ e_1 * e_2 & \Rightarrow\ \mathsf{evalExp}\ \theta\ e_1 * \mathsf{evalExp}\ \theta\ e_2 \end{cases}$$

$$\mathsf{evalBexp}\ \theta\ b = \mathsf{case}\ b \begin{cases} \mathsf{bLoc}\ \mathit{lval} & \Rightarrow\ \mathsf{toB}(\theta(\mathit{lval})) \\ \mathsf{bLit}\ b & \Rightarrow\ b \\ \neg b & \Rightarrow\ \neg(\mathsf{evalBexp}\ \theta\ b) \\ b_1 \vee b_2 & \Rightarrow\ \mathsf{evalBexp}\ \theta\ b_1 \vee \mathsf{evalBexp}\ \theta\ b_2 \\ b_1 \wedge b_2 & \Rightarrow\ \mathsf{evalBexp}\ \theta\ b_1 \wedge \mathsf{evalBexp}\ \theta\ b_2 \\ e_1 = e_2 & \Rightarrow\ \mathsf{evalExp}\ \theta\ e_1 = \mathsf{evalExp}\ \theta\ e_2 \\ e_1 < e_2 & \Rightarrow\ \mathsf{evalExp}\ \theta\ e_1 < \mathsf{evalExp}\ \theta\ e_2 \end{cases}$$

▶ Remark 1 (Partiality). Expression evaluation is partial because there is no guarantee that $\theta(\mathit{lval})$ is defined: an *lval* being looked-up may not be in the map θ. Failure in evaluation is modelled by the option type, and must be handled in the semantics and the matching algorithm. However error handling is omitted in the presentation since it hampers readability. See the HOL4 formalization for full details.

2.2 Semantics

We now confess to misleading the reader: in spite of the notational similarity, a contiguity type is *not* a type: it is a formal language. A type is usually understood to represent a set, or domain, of values, e.g., the type int32 represents a set of integers. In contrast, the contiguity type i32 represents the set of strings of width 32 bits. An element of a contiguity type can be turned into an element of a type by providing interpretations for all the strings at the leaves and interpreting the Recd and Array constructors into the corresponding type constructs. (A base contiguity type therefore serves mainly as a *tag* to be interpreted as a width and also as an intended target type.) Thus, contiguity types sit – conveniently – between the types in a programming language and the strings used to make messages.

The semantics definition depends on a few basic notions familiar from language theory: language concatenation, and iterated language concatenation.

$$\begin{aligned} L_1 \cdot L_2 &= \{w_1 w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\} \\ L^0 &= \varepsilon \\ L^{n+1} &= L \cdot L^n \end{aligned}$$

▶ **Definition 2** (Semantics of contiguity types). *In the following, we assume given an assignment θ for evaluating expressions. If an expression evaluation fails, the language being constructed will be $\emptyset$.*

$$\mathcal{L}_\theta(\tau) = \mathtt{case}\ \tau \left\{ \begin{array}{l} \mathit{base} \Rightarrow \{s \mid \mathsf{len}(s) = \mathsf{width}(\mathit{base})\} \\ \mathsf{Recd}\ (f_1 : \tau_1) \ldots (f_n : \tau_n) \Rightarrow \mathcal{L}_\theta(\tau_1) \cdot \ldots \cdot \mathcal{L}_\theta(\tau_n) \\ \mathsf{Array}\ \tau\ \mathit{exp} \Rightarrow \\ \quad \left\{ \begin{array}{ll} \mathcal{L}_\theta(\tau)^{\mathsf{evalExp}\ \theta\ \mathit{exp}} & \text{if } \mathsf{evalExp}\ \theta\ \mathit{exp} \text{ succeeds} \\ \emptyset & \textit{if evaluation fails} \end{array} \right. \\ \mathsf{Alt}\ \mathit{bexp}\ \tau_1\ \tau_2 \Rightarrow \\ \quad \left\{ \begin{array}{ll} \mathcal{L}_\theta(\tau_1) & \text{if } \mathsf{evalBexp}\ \theta\ \mathit{bexp} = \mathsf{true} \\ \mathcal{L}_\theta(\tau_2) & \text{if } \mathsf{evalBexp}\ \theta\ \mathit{bexp} = \mathsf{false} \\ \emptyset & \textit{if evaluation fails} \end{array} \right. \end{array} \right.$$

▶ **Example 3.** Consider the following schema for an *option* contiguity type. The empty record {} associated with boolean expression b has no fields.

$\mathtt{if}\ b\ \mathtt{then}\ \{\,\}\ \mathtt{else}\ c$

In case b evaluates to true, no portion of the string is consumed; otherwise, c specifies the remainder of the processing. It may be instructive to consider how this type works with arrays. For example, a string meeting the following contig specification

```
(if b then {} else i32) [3]
```

is either zero or twelve bytes in length (assuming that `i32` is four bytes wide).

3 Algorithms

The following are classical topics in formal language theory and practice, and they are worth investigating in the context of contiguity types. At present we have been working on decoding, filtering, and test generation.

Decoding A decoder breaks a sequence of bytes up and puts the pieces into a useful data structure, typically a parse tree. We will discuss this in more detail in Section 3.1.

Filtering A filter computes an answer to the question: "does a sequence of bytes meet the specification of a given contiguity type". This is an instance of the language recognition problem. More powerful filters enforce that certain fields of a message, when interpreted, meet specific semantic properties. We will discuss this further in Section 4.1.

Serialization Given a contiguity type, synthesize a function that writes a compact binary version of a data structure to a message.

Test generation Given a contiguity type, generate byte sequences that do (or do not) meet its specification and feed the sequences to implementations in order to observe their behaviour.

Learning Given training sets of messages that are accepted/rejected by an implementation, attempt to discover a contiguity type for the entire set of messages.

3.1 Decoding

Above we mentioned that decoding can result in parse trees; however, self-describing messages allow a different conceptual framework to be brought to bear. There is an important distinction between parsing, which *creates* structure (parse trees), and matching, which is given structure and calculates assignments (substitutions).[3] Giving some evocative types helps make the difference clear:

$$\begin{aligned} \mathsf{parse} &: \mathit{grammar} \to \mathsf{string} \to \mathit{parsetree} \\ \mathsf{match} &: \mathit{pattern} \to \mathsf{string} \to \mathit{assignments} \end{aligned}$$

For our purposes, namely decoding datastructures in binary format, the central decoding algorithm is a *matcher*: given a contiguity type τ and a string s, the matcher will either fail, or succeed with an assignment $\theta : \mathit{lval} \mapsto \mathsf{string}$ mapping each L-value in τ to its corresponding slice of s. The assignment θ can be post-processed to yield a standard parse tree, but its novelty and strength is that θ can be dynamically consulted to access the values needed to guide the processing of self-describing messages.

▶ **Definition 4** (Matching algorithm). *The matcher operates over a triple $(\mathit{worklist}, \mathit{str}, \theta)$ where worklist is a stack used to linearize the input contiguity type τ, str represents the remainder of the input string, and θ is the assignment being built up. Each element of the worklist is a (τ, lval) pair, where τ is a* contig*, and lval is the path growing down from the root to τ. The notation $(\mathit{lval} \mapsto \mathit{slice}) \bullet \theta$ denotes the addition of binding $\mathit{lval} \mapsto \mathit{slice}$ to θ. We examine the cases in turn:*

1. *The worklist is empty; the match succeeds.*

$$([], \mathit{str}, \theta) \Rightarrow \mathsf{SOME}(\mathit{str}, \theta)$$

2. *The first element of the worklist is a base type. The prescribed number of bytes are broken off the front of the string, giving $\mathit{str} = (\mathit{slice}, \mathit{rst})$; then the binding is added to θ before recursing. If the string is shorter than the requested number of bytes, fail.*

$$((\mathsf{Basic}\ a, \mathit{lval}) :: t, \mathit{str}, \theta) \Rightarrow (t, \mathit{rst}, (\mathit{lval} \mapsto \mathit{slice}) \bullet \theta)$$

[3] Thus the notion of matching discussed here is in the tradition of term rewriting [1], the main difference being that our substitutions are applied to *lval*s rather than variables.

3. *The first element of the worklist is recd* $= \mathsf{Recd}\ (f_1 : \tau_1) \ldots (f_n : \tau_n)$*. Before recursing, the fields are pushed onto the stack, extending the path to each field element:*

$$((recd, lval) :: t, str, \theta) \Rightarrow ([(\tau_1, lval.f_1), \cdots, (\tau_n, lval.f_n)]@t, str, \theta)$$

4. *The first element of the worklist is an array. The dimension expression is evaluated to get the width d, then d copies are pushed onto the stack, where each path is extended with the array index.*

$$((\mathsf{Array}\ \tau\ exp, lval) :: t, str, \theta) \Rightarrow ([(\tau, lval[0]), \cdots, (\tau, lval[d-1])]@t, str, \theta)$$

5. *The first element of the worklist is an* Alt*. If b evaluates to* true*,* τ_1 *is pushed on to the worklist; if it evaluates to* false*,* τ_2 *is pushed. Otherwise, fail.*

$$((\mathsf{Alt}\ b\ \tau_1\ \tau_2, lval) :: t, str, \theta) \Rightarrow ((\tau_i, lval) :: t, str, \theta)$$

The matcher function, match *begins with an initial state*

$$state_0 = ([(root, \tau)], str_0, \emptyset)$$

where the initial path is a default lval variable named root, the initial string is str_0*, and the initial assignment has no bindings.*

▶ **Theorem 5** (Matcher termination). *As mentioned, the state of the matcher is held in a* $(worklist, str, \theta)$ *tuple. The termination relation is a lexicographic combination, where either str gets shorter, or, str is unchanged and worklist gets smaller under the multiset order. (The multiset order is useful in this proof since the handling of the* Array *construct is a nice version of the Hercules-Hydra problem [3].)*

▶ **Definition 6** (Substitution application). *Correctness depends on an operation* $\theta\ lval\ \tau$ *applying substitution* θ *to contiguity type* τ*, starting at lval, in order to reconstruct the original string.*

$$\theta\ lval\ \tau = \mathsf{case}\ \tau \begin{cases} base \Rightarrow \theta(lval) \\ \mathsf{Recd}\ (f_1 : \tau_1) \ldots (f_n : \tau_n) \Rightarrow \theta\ (lval.f_1)\ \tau_1 \cdot \ldots \cdot \theta\ (lval.f_n)\ \tau_n \\ \mathsf{Array}\ \tau_1\ exp \Rightarrow \\ \quad \begin{cases} \theta\ (lval[0])\ \tau_1 \cdot \ldots \cdot \theta\ (lval[d-1])\ \tau_1, & \textit{if } d = \mathsf{evalExp}\ \theta\ exp \\ \emptyset & \textit{if evaluation fails} \end{cases} \\ \mathsf{Alt}\ bexp\ \tau_1\ \tau_2 \Rightarrow \\ \quad \begin{cases} \theta\ lval\ \tau_1 & \text{if } \mathsf{evalBexp}\ \theta\ bexp = \mathsf{true} \\ \theta\ lval\ \tau_2 & \text{if } \mathsf{evalBexp}\ \theta\ bexp = \mathsf{false} \\ \emptyset & \textit{if evaluation fails} \end{cases} \end{cases}$$

▶ **Theorem 7** (Correctness of substitution). *The correctness statement for the matcher is similar to those found in the term rewriting literature, namely that the computed substitution applied to the contiguity type yields the original string:*

$$\mathsf{match}\ state_0 = \mathsf{SOME}(\theta, s) \Rightarrow \theta\ root\ \tau \cdot s = str_0$$

Proof. By induction on the definition of match. ◀

▶ **Theorem 8** (Matcher soundness). *The connection to* $\mathcal{L}_\theta(\tau)$ *is formalized as*

$$str_0 = s_1 s_2 \wedge \mathsf{match}\ state_0 = \mathsf{SOME}(\theta, s_2) \Rightarrow s_1 \in \mathcal{L}_\theta(\tau)$$

Proof. By induction on the definition of match. ◀

In other words, a successful match provides a θ that will successfully evaluate all encountered expressions, and the matched string is indeed in the language of τ. A completeness theorem going in the other direction has not yet been tackled.

▶ **Example 9.** Given the contig

```
{A : Bool
 B : Char
 len : u16
 elts : i32 [len]
}
```

and an input string (listed in hex)

```
[0wx1, 0wx67, 0wx0, 0wx5, 0wx0, 0wx0, 0wx0, 0wx19, 0wx0, 0wx0,
 0wx9, 0wx34, 0wx0, 0wx0, 0wx30, 0wx39, 0wx0, 0wx0, 0wxD4,
 0wx31, 0wxFF, 0wxFF, 0wxFE, 0wxB3]
```

created by encoding: the boolean `true`, the letter `g`, the number 5 (MSB 2 byte unsigned), and the five MSB 4 byte signed twos complement integers 25, 2356, 12345, 54321, and -333, the matcher creates the following assignment of *lval*s to substrings of the input:

```
[(root.A,       (Bool, [0wx1])),
 (root.B,       (Char, [0wx67])),
 (root.len,     (u16,  [0wx0, 0wx5])),
 (root.elts[0], (i32,  [0wx0, 0wx0, 0wx0, 0wx19])),
 (root.elts[1], (i32,  [0wx0, 0wx0, 0wx9, 0wx34])),
 (root.elts[2], (i32,  [0wx0, 0wx0, 0wx30, 0wx39])),
 (root.elts[3], (i32,  [0wx0, 0wx0, 0wxD4, 0wx31])),
 (root.elts[4], (i32,  [0wxFF, 0wxFF, 0wxFE, 0wxB3]))
]
```

Note that each element of the list is of the form $(lval, (tag, bytes))$ where each slice is labelled with its corresponding base type, to support further translation.

Thus the matcher will break up the input string in accordance with the specification; the execution, in effect, generates a sequence of assignments that, if applied, would populate a data structure with the specified data in the specified places. Therefore it is not really necessary to generate parse trees to in order to decode messages: one merely needs a target data structure to write data into. (In fact, when filtering, no target data structure is needed at all.) The correctness property will ensure that *all* fields are written with the specified data. The assignments can be incrementally evaluated as the decoder runs, or can be stored and applied when the decoder terminates.

4 Extended contiguity types

In the discussion so far, contiguity types can only express bounded data: each base type has a fixed size and all Array types are given an explicit bound. Removing these two restrictions would greatly increase expressiveness. Of course, we look to the theory of formal languages to guide extensions to the formalism. We have thus explored the addition of the empty language $\emptyset$, Kleene star, and a lexer. The augmented syntax can be seen in Figure 3. The addition of $\emptyset$ (via Void) and Kleene star (via List) has been accomplished, along with the

$$
\begin{array}{rcl}
\tau & = & \mathsf{Base}\ (\mathit{regexp} \times \mathit{valFn}) \\
 & | & \mathsf{Void} \\
 & | & \mathsf{List}\ \tau \\
 & | & \mathsf{Recd}\ (f_1 : \tau_1) \ldots (f_n : \tau_n) \\
 & | & \mathsf{Array}\ \tau\ \mathit{exp} \\
 & | & \mathsf{Alt}\ \mathit{bexp}\ \tau_1\ \tau_2
\end{array}
$$

Figure 3 Extended contiguity types.

proofs verifying the upgraded matcher. We also discuss replacing the existing base types with a lexer for completeness, even though that discussion more appropriately belongs to future work.

▶ **Definition 10** (Semantics additions). *A base type element has the form* Base(*regexp*, *valFn*), *where valFn is a function that maps the string recognized by regexp to a data value. The semantics of a* Base *type is just the (formal language) semantics of its regular expression,* Void *denotes the empty set, and* List *is a "tagged" version of Kleene star (*NilTag *and* ConsTag *are described in Section 4.2).*

$$
\mathcal{L}_\theta(\tau) = \mathtt{case}\ \tau \begin{cases} \mathsf{Base}\ (\mathit{regexp}, \mathit{valFn}) \Rightarrow \mathcal{L}(\mathit{regexp}) \\ \mathsf{Void} \Rightarrow \emptyset \\ \mathsf{List}\ \tau \Rightarrow (\mathsf{ConsTag} \cdot \mathcal{L}_\theta(\tau))^* \cdot \mathsf{NilTag} \\ \ldots\mathit{(previous\ clauses)} \end{cases}
$$

4.1 Void and in-message assertions

The Alt constructor combined with Void supports an *in-message assertion* feature. The contiguity type Assert *bexp* is defined as follows:

▶ **Definition 11** (Assert).

$$\mathsf{Assert}\ \mathit{bexp} = \mathsf{Alt}\ \mathit{bexp}\ (\mathsf{Recd}\,[\,])\ \mathsf{Void}$$

The meaning of Assert b is obtained by simplification:

$$\mathcal{L}_\theta(\mathsf{Assert}\ \mathit{bexp}) = \mathtt{if}\ \mathsf{evalBexp}\ \theta\ \mathit{bexp}\ \mathtt{then}\ \varepsilon\ \mathtt{else}\ \emptyset$$

and match evaluates it by failing when evalBexp θ *bexp* is false, and otherwise continuing on without advancing in the input.

▶ **Example 12** ($A^nB^nC^n$). It is well known that $L = A^nB^nC^n$ is not a context-free language. We can use Assert to specify a language that is *nearly* L with the following contiguity type:

```
charA = {ch : char, isA : Assert (ch = 65)}   (* "A" = ASCII 65 *)
charB = {ch : char, isB : Assert (ch = 66)}
charC = {ch : char, isC : Assert (ch = 67)}

mesg = {len : u16
        A : charA [len]
        B : charB [len]
        C : charC [len]
      }
```

In fact $\mathcal{L}_\theta(\mathsf{mesg}) = u \cdot A^{\mathsf{toN}(u)} \cdot B^{\mathsf{toN}(u)} \cdot C^{\mathsf{toN}(u)}$.

Assert expressions have been used extensively when specifying wellformedness properties for messages in our application work. The applications include restrictions on array sizes and constraints on array elements, e.g., requiring that every element in an array of GPS coordinates is an acceptable GPS coordinate.

▶ **Example 13** (Array limits). In UxAS messages (see Section 5) the length of every array element is held in a separate *length* field which is two bytes in size. Thus the following contig, in the absence of any further constraint, supports arrays of length up to 65536 elements. A receiver system may well not be prepared for messages having collections of such potentially large components.

```
{ len : u16
  elts : i32 [len] }
```

In the meta-data for such messages, one can sometimes find information about the maximum expected size, usually a fairly small number. This can be directly expressed inside the contig with an Assert:

```
{ len : u16
  len-range : Assert (len <= 8)
  elts : i32 [len] }
```

Note that the expected array length should be specified before the array itself, otherwise the allocation attempt might be made before the check.

4.2 Kleene Star

Although the use of bounded Array types provides much expressiveness for representing sequences of data, ultimately some kinds of message can not be handled, i.e., those where there is no way to predict the number of nestings of structure: s-expressions, logical formulae, and programming language syntax trees are some typical examples. We address this shortcoming by adding a new contiguity type constructor – List – of unbounded lists. A message matching a List τ type will be subject to an encoding similar to implementations of lists in functional languages. The matching algorithm for contiguity types is extended to handle List objects by iteratively unrolling the recursive equation

$$L^* = \varepsilon \cup L \cdot L^*$$

Indeed the type List τ is represented by the following contiguity type, a recursive record:

$$\mathsf{List}\ \tau = \left\{ \begin{array}{lcrll} \mathsf{tag} & : & \mathsf{u8} & & \\ \mathsf{test} & : & \mathsf{Alt} & (\mathsf{tag} = \mathsf{NilTag}) & \longrightarrow\ \varepsilon \\ & & & (\mathsf{tag} = \mathsf{ConsTag}) & \longrightarrow\ \{\mathsf{hd} : \tau,\ \mathsf{tl} : \mathsf{List}\ \tau\} \\ & & & \text{otherwise} & \longrightarrow\ \mathsf{Void} \end{array} \right.$$

In words, a List τ matches a sequence of records where a single-byte tag (NilTag or ConsTag) is read, then tested to see whether to stop parsing the list (NilTag) or to continue on to parse a τ into the hd field and recurse in order to process the remainder of the list. An incorrect value for the tag results in failure. Thus, the list of integers

$$\mathsf{Cons}(1, \mathsf{Cons}(2, \mathsf{Cons}(3, \mathsf{Nil})))$$

can be represented in a message as (assume Code is an encoder for integers)

$$\mathsf{ConsTag} \cdot \mathsf{Code}(1) \cdot \mathsf{ConsTag} \cdot \mathsf{Code}(2) \cdot \mathsf{ConsTag} \cdot \mathsf{Code}(3) \cdot \mathsf{NilTag}$$

and match, given type $\mathsf{List\ int}$ (where int is a contiguity type for some flavor of integer) succeeds, returning the context

$$\begin{array}{rcl}
\mathsf{root.tag} & \mapsto & \mathsf{ConsTag} \\
\mathsf{root.hd} & \mapsto & \mathsf{Code}(1) \\
\mathsf{root.tl.tag} & \mapsto & \mathsf{ConsTag} \\
\mathsf{root.tl.hd} & \mapsto & \mathsf{Code}(2) \\
\mathsf{root.tl.tl.tag} & \mapsto & \mathsf{ConsTag} \\
\mathsf{root.tl.tl.hd} & \mapsto & \mathsf{Code}(3) \\
\mathsf{root.tl.tl.tl.tag} & \mapsto & \mathsf{NilTag}
\end{array}$$

This solution is compositional, in the sense that List types can be the arguments of other contiguity types, can be applied to themselves, e.g., $\mathsf{List}(\mathsf{List}\ \tau)$, etc. Thus quite general branching structures of arbitrary finite depth and width can be specified and parsed with this extension. The approach captures a certain class of *context-free-like* languages. However, it differs distinctly from the standard Chomsky hierarchy, mainly because sums are determined by *looking behind* when computing which choice to follow in an Alt type; for example, the list parser branches *after* it has seen the tag. The similarity with 'no-lookahead' parsing, such as LL(0) and LR(0), deserves further investigation.

▶ **Example 14** (First order term challenge)**.** Although lists of contig types can be straightforwardly constructed with the above encoding, there remains a problem when lists are part of a recursive construction. A classic example is *first order terms*, as described by the following ML-style datatype:

```
term = Var of string
     | App of string * term list
```

The following contiguity types capture a binary encoding of `term`, using tags to distinguish the two kinds of term:

$$\mathsf{string} = \{\mathsf{len} : \mathsf{u16}, \mathsf{elts} : \mathsf{char}\,[\mathsf{len}]\}$$

$$\mathsf{term} = \left\{\begin{array}{lcll}
\mathsf{tag} & : & \mathsf{u8} & \\
\mathsf{test} & : & \mathsf{Alt} & (\mathsf{tag} = \mathsf{VarTag}) \longrightarrow \{\mathsf{varName} : \mathsf{string}\} \\
 & & & (\mathsf{tag} = \mathsf{AppTag}) \longrightarrow \{\mathsf{fnName} : \mathsf{string},\ \mathsf{Args} : \mathsf{List\ term}\} \\
 & & & \text{otherwise} \longrightarrow \mathsf{Void}
\end{array}\right.$$

However, such *nested recursive* specifications demand more elaborate constructions, for example treating term and List as being mutually recursively defined, as is already done for nested recursive datatypes in theorem provers [6].

4.3 Lexing

Currently, the set of base contiguity types comprises the usual base types expected in most programming languages. Semantically, a base type denotes a set of strings of the specified width, but it is also coupled with an *interpretation function* for example, the contiguity type `u8` denotes the set of all one-byte strings, interpreted by the usual unsigned valuation function:

$$\mathsf{u8} = (\{s \mid \mathsf{length}(s) = 1\}, \mathsf{toN})$$

This approach cannot, however, capture base types such as string literals of arbitrary size, or bignums, or the situation in packed bit-level encodings where fields are of *ad hoc* sizes aimed at saving space. A common generalization is to express base types via regular expressions paired with interpretation functions. In that setting u8 can be defined as

$$\mathsf{u8} = (.\,, \mathsf{toN})$$

(where '.' is the standard regular expression denoting any character). Similarly,

$$\mathsf{Cstring} = ([\backslash 001 - \backslash 255]^{*}\backslash 000, \lambda x.\, x)$$

denotes the set of zero-terminated strings, as found in the C language. Its displayed interpretation is just the identity function, but could just as well be a function that drops the terminating \000 character.

Scott Owens has already formalized a HOL4 theory of regexp based, maximal munch, lexer generation, and it is future work to adapt contiguity types to use those lexemes instead of the current restricted set of base types.

5 Application

In the DARPA CASE project, we have been applying contiguity types to help create provably secure message filters and runtime monitors. One example we have been working with is *OpenUxAS*, which has been developed by the Air Force Research Laboratory.[4] UxAS is a collection of modular services that interact via a common message-passing architecture, aimed at unmanned autonomous systems. Each service subscribes to messages in the system and responds to queries. The content of each message conforms to the Light-weight Message Control Protocol (LMCP) format. In UxAS, software classes providing LMCP message creation, access, and serialization/deserialization are automatically generated from XML descriptions, which detail the exact data fields, units, and default values for each message. All UxAS services communicate with LMCP formatted messages.

An example LMCP message type is AirVehicleState. The following is its contiguity type:

```
AirVehicleState =
   {EntityState   : EntityState
    Airspeed      : float
    VerticalSpeed : float
    WindSpeed     : float
    WindDirection : float
   }
```

where an EntityState is quite an elaborate type:

```
{ ID : i64
  u  : float
  v  : float
  w  : float
  udot : float
  vdot : float
  wdot : float
  Heading : float
```

[4] See https://github.com/afrl-rq/OpenUxAS.

```
     Pitch    : float
     Roll     : float
     p : float,  q : float, r : float
     Course : float
     Groundspeed : float
     Location : mesgOption "LOCATION3D" location3D
     EnergyAvailable : float
     ActualEnergyRate : float
     PayloadStateList
        : uxasBoundedArray (mesgOption "PAYLOADSTATE" payloadState) 8
     CurrentWaypoint : i64
     CurrentCommand  : i64
     Mode : uxasNavigationMode
     AssociatedTasks : uxasBoundedArray i64 8
     Time : i64
     Info : uxasBoundedArray(mesgOption "KEYVALUEPAIR" keyValuePair) 32
   }
```

Most of the fields are simple base types, but the Location field and the PayloadStateList are complex. They are expressed with some derived syntax, which we will explain.

- The Location field, `mesgOption "LOCATION3D" location3D`, may or may not occur (signalled with a tag field), but if it does, it is a location3D, which is a GPS location, and its latitude, longitude, and altitude fields are checked with an Assert to make sure they lie within the expected numeric ranges, which are expressed as floating point numbers.

```
AltitudeType = AGL | MSL

location3D = {
 Latitude  : double,
 Longitude : double,
 Altitude  : float,
 AltitudeType : AltitudeType,
 Wellformed : Assert (
     -90.0 <= Latitude <= 90.0 and
    -180.0 <= Longitude <= 180.0 and
     0.0 <= Altitude <= 15000.0)
}
```

- The PayloadStateList is a variable-length array of optional records, with maximum length 8. Each record has, along with other fields, its own variable-length array of key-value pairs, and the key and value of each such pair is a variable-length string.

We have formalized most of the LMCP messages as contiguity types, and created filters and parsers by instantiating the match algorithm. In order to meet the demands of LMCP message modelling, the matcher algorithm has been upgraded to support a fuller expression and boolean expression language, but the core algorithm is the same as our verified core version. The filters and parsers have been added to an existing UxAS design and successfully tested with the UxAS simulator.

6 Extensions and future work

Various extensions have been easy to add to the contiguity type framework, and we also have more substantial ideas to pursue for future work.

Enumerations An *enumeration* declaration introduces a new base contiguity type, and also adds the specified elements to a map associating constant names to numbers. Suppose that enumerations are allowed to have up to 256 elements, allowing any enumerated element to fit in one byte. The following enumeration is taken from UxAS messages:

```
NavigationMode
  = Waypoint | Loiter | FlightDirector
  | TargetTrack | FollowLeader | LostComm
```

A field expecting a NavigationMode element will be one byte wide, and thus there are 250 byte patterns that should not be allowed in the field. Thus, the contig

```
{ A : NavigationMode }
```

should be replaced by

```
{ A : NavigationMode
  A-range : Assert (A <= 5)
}
```

Raw blocks A raw chunk of a string (byte array) of a size that can depend on the values of earlier fields is easy to specify:

$$\mathsf{Raw}\ \mathit{exp}$$

For example, a large Array form can lead to a large number of L-values being stored in θ; if none are ever accessed later, e.g., if the array is some image data, it can be preferable to simply declare a Raw block. Thus a 2D array can be blocked out in the following manner:

```
{ rows : i32
  cols : i32
  block : Raw (rows * cols)
}
```

Guest scanners It seemed useful to provide a general ability to host scanning functions. This is accomplished via the following constructor:

$$\mathsf{Scanner}\ (\mathit{scanfn} : \mathsf{string} \to (\mathsf{string} \times \mathsf{string})\mathsf{option})$$

When a custom scanner is encountered during the matching process, the scanner is invoked on the input and should either fail or provide an (s_1, s_2) pair representing a splitting of the input. Then s_1 is added to θ at the current *lval*, and matching continues on s_2.

Non-copying implementations In the discussion so far, we have assumed that the input string is being broken up into substrings that are placed into the *lval* map θ. However, very little is changed if, instead of a substring, an *lval* in θ maps to a pair of indices $(\mathit{pos}, \mathit{width})$ designating the location of the substring. The result is a matcher that never copies byte buffer data. This is necessary to synthesize efficient filters.

In making this representation change, there is a slight change to the semantics. In the original, $\theta(\mathit{lval})$ yields a string whereas in the non-copying version, $\theta(\mathit{lval})$ yields a pair of indices, which means that the original string str_0 needs to be included in applying the assignment.

Compilation Our current implementation of contiguity types is in an *interpreted* style: the evaluation of numeric Array bounds and boolean guards on Alt conditions is done with respect to the current context, which is explicitly accumulated as the message is processed. However, notice that the host programming language will no doubt already provide compilation for numeric and boolean expressions. This leads to the idea of *compiling contiguity types*: the current matching algorithm for contiguity types can be replaced by the generation of equivalent host-language code which is then compiled by the host-language compiler and evaluated. Although the current contiguity type matcher has proved to be fast enough to keep up with the real-time demands of the UxAS system, we expect that a compiled version of the code would be much faster.

We would like to formalize the compilation algorithm and prove it correct. Since the CakeML[9] formalization provides an operational semantics for CakeML programs, and a convenient translation from HOL4 expressions to CakeML ASTs, one should be able to prove a correctness theorem relating the matcher function of the present paper with a compiler that takes a contiguity type and generates CakeML. However, this is only speculative; there are many details to work out.

Relationship with grammars Contiguity types and match provide a type-directed and context-oriented parser generator that has some similarities with LL(0) or LR(0) languages wherein the parser can proceed with no lookahead. This is useful for binary-encoded datastructures. It would be very interesting to attempt to bridge the gap with conventional parsing technology based on grammars. A good beginning would be to understand the issues involved in attempting to translate context-free grammars into contiguity types and *vice versa*.

7 Related work

As mentioned in the introduction, domain specific languages for message formats have been around for a long time. Semantic definitions and verification for them is a much more recent phenomenon. The PADS framework [4] aimed at supporting a wide variety of formats, including text-based. Its core message description formalism was given semantics by translation into dependent type theory. An interesting integration of context-sensitivity into a conventional grammar framework has been done by Jim and Mandelbaum [7]. Everparse [11] is an impressive approach, based on parser combinators and having an emphasis on proving the invertibility of encode/decode pairs with automated proof (other properties are also established). Chlipala and colleagues [2] similarly emphasize encode/decode proofs, basing their work in Coq and using the power of dependent types to good effect. Formats based on dependent types can (and do) use the built-in expressive power of type theory to enforce semantic properties on data. A recent language in this vein is Parsely [10] which leverages the dependent records and predicate subtyping of PVS to provide a combination of PEG parsing and attribute grammars aimed at parsing complex language formats.

Many of these efforts obtain the semantics of the data description formalism by translation into features provided by a powerful host logic. In contrast, contiguity types use only very basic – and easily implemented – concepts. This means that contiguity type matchers, parsers, and extensions can be directly implemented in any convenient programming language. Contiguity types also provide a kind of dependency, without leveraging the type system of the theorem prover. We suspect that working at the representation level, and using L-values, allows one to get some of the benefits of type dependency. Another distinguishing aspect of our work is that our emphasis on filters means that we are primarily interested in the enforcement of semantic properties on message contents rather than encode/decode properties. In future work we expect to be able to leverage this in high performance filter implementations.

8 Conclusion

We have designed, formalized, proved correct, implemented, and applied a specification language for message formats, based on formal languages and the venerable notion of L- and R-values from imperative programming. The notion of contiguity type seems to give a lot of expressive power, sufficient to tackle difficult idioms in self-describing formats. Contiguity types integrate common structuring mechanisms from programming languages, such as arrays, records, and lists while keeping the foundation in sets of strings, which seems appropriate for message specifications.

References

1 Franz Baader and Tobias Nipkow. *Term Rewriting and all that.* Cambridge University Press, 1998.

2 Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. Narcissus: correct-by-construction derivation of decoders and encoders from binary formats. *Proc. ACM Program. Lang.*, 3(ICFP):82:1–82:29, 2019. doi:10.1145/3341686.

3 Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *CACM*, 22(8):465–476, 1979.

4 Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. The next 700 data description languages. *J. ACM*, 57(2), 2010. doi:10.1145/1667053.1667059.

5 David S. Hardin, Konrad Slind, Mark Bortz, James Potts, and Scott Owens. A high-assurance, high-performance hardware-based cross-domain system. In Amund Skavhaug, Jérémie Guiochet, and Friedemann Bitsch, editors, *Computer Safety, Reliability, and Security - 35th International Conference, SAFECOMP 2016, Trondheim, Norway, September 21-23, 2016, Proceedings*, volume 9922 of *Lecture Notes in Computer Science*, pages 102–113. Springer, 2016. doi:10.1007/978-3-319-45477-1_9.

6 John Harrison. Inductive definitions: automation and application. In E. Thomas Schubert, Phillip J. Windley, and James Alves-Foss, editors, *Proceedings of the 1995 International Workshop on Higher Order Logic theorem proving and its applications*, number 971 in LNCS, pages 200–213, Aspen Grove, Utah, 1995. Springer-Verlag.

7 Trevor Jim and Yitzhak Mandelbaum. A new method for dependent parsing. In Gilles Barthe, editor, *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6602 of *Lecture Notes in Computer Science*, pages 378–397. Springer, 2011. doi:10.1007/978-3-642-19718-5_20.

8 Donald E. Knuth. Semantics of context-free languages. In *In Mathematical Systems Theory*, pages 127–145, 1968.

9 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, 2014. doi:10.1145/2535838.2535841.

10 Prashanth Mundkur, Linda Briesemeister, Natarajan Shankar, Prashant Anantharaman, Sameed Ali, Zephyr Lucas, and Sean Smith. Research report: The parsley data format definition language. In *2020 IEEE Security and Privacy Workshops (SPW)*, pages 300–307, 2020. doi:10.1109/SPW50608.2020.00064.

11 Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. Everparse: Verified secure zero-copy parsers for authenticated message formats. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019,*

pages 1465–1482. USENIX Association, 2019. URL: `https://www.usenix.org/conference/usenixsecurity19/presentation/delignat-lavaud`.

12 Dennis Ritchie. The development of the C language. https://www.bell-labs.com/usr/dmr/www/chist.html.

13 Christopher Strachey. Fundamental concepts in programming languages. *High. Order Symb. Comput.*, 13(1/2):11–49, 2000. `doi:10.1023/A:1010000313106`.

Proof Pearl : Playing with the Tower of Hanoi Formally

Laurent Théry ✉
INRIA Sophia Antipolis - Université Côte d'Azur, France

Abstract

The Tower of Hanoi is a typical example that is used in computer science courses to illustrate all the power of recursion. In this paper, we show that it is also a very nice example for inductive proofs and formal verification. We present some non-trivial results that have been formalised in the Coq proof assistant.

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

Keywords and phrases Mathematical logic, Formal proof, Hanoi Tower

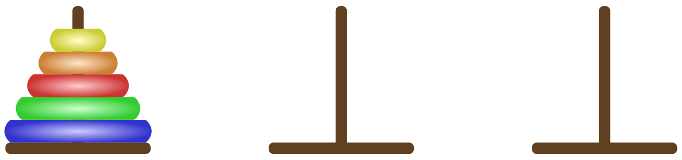
Digital Object Identifier 10.4230/LIPIcs.ITP.2021.31

Supplementary Material The code associated with this paper can be found at
Software (Code Repository): `https://github.com/thery/hanoi`
archived at `swh:1:dir:6f8e383f8827c6fd688847c23fab91a497bd60c7`

Acknowledgements We would like to thank Paul Zimmermann for pointing us to the star problem with 4 pegs and the anonymous reviewers for their careful reading and suggestions.

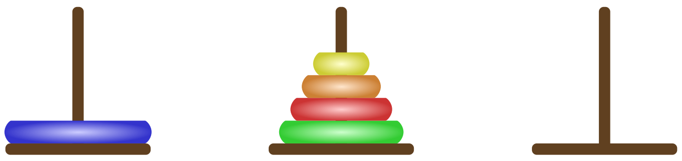
1 Introduction

The Tower of Hanoi is often used in computer science courses as example to teach recursion. The puzzle is composed of three pegs and some disks of different sizes. Here is a drawing of the initial configuration for five disks [1]:



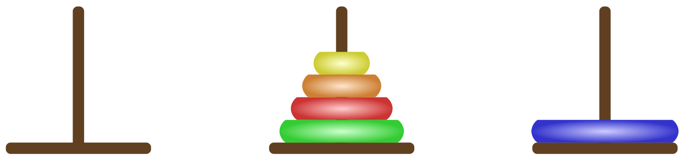
Initially, all the disks are stacked in increasing size on the left peg. The goal is to move them to the right peg using the middle peg as an auxiliary peg. There are two rules. First, only one disk can be moved at a time and this disk must be on the top of its peg. Second, a larger disk can never be put on top of a smaller one.

A program P^{3r} that solves this puzzle can easily be written using recursion : one builds the program P^{3r}_{n+1} that solves the puzzle for $n+1$ disks using the program P^{3r}_n that solves the puzzle for n disks. The algorithm proceeds as follows. We first call P^{3r}_n to move the top-n disks to the middle peg using the right peg as the auxiliary peg.

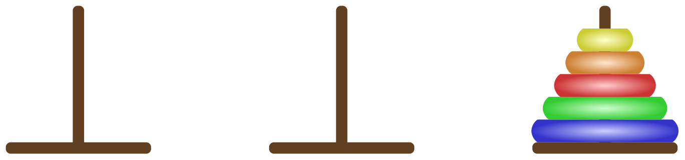


[1] We use macros designed by Martin Hofmann and Berteun Damman for our drawings.

Then we move the largest disk to its destination.



Finally, we use P_n^{3r} to move the n disks on the intermediate peg to their destination using the left peg as the auxiliary peg.



This simple recursive algorithm is also optimal: it produces the minimal numbers of moves.

In this paper, we consider some variants of this puzzle (with three or four pegs, with some constraints on the moves one can perform) and explain how these puzzles and their optimal solution have been formalised.

2 General setting

We present the general settings of our formalisation that has been done in Coq using the SSReflect extension [5]. Then, we explain more precisely the variants of the puzzle that we have taken into consideration and how they have been formalised. We have tried as much as possible to be precise and present exactly what has been formalised. For this, we adopt the syntax of the Mathematical Component library [11] that we have been using. We also use the following typesetting convention. Notions that are present in the library are written using a `typewriter font` while our own definitions are written using a **roman font**.

Syntax Summary

We first give of an overview of the syntax of the Mathematical Component library. For a more detailed presentation, we refer the reader to [5]. The basic data structures we are using are natural numbers and lists. For natural numbers, they are implemented by an inductive type with two constructors : a zero (0) and a successor (`S`). There is a special notation to hide the application of the successor. So, n`.+1`, n`.+2` and n`.+3` represent (`S` n), (`S` (`S` n)) and (`S` (`S` (`S` n)) respectively. For example, the addition of two natural numbers is defined as

`Fixpoint` $m + n$ `:=` `if` m `is` m_1`.+1` `then` $(m_1 + n)$`.+1` `else` n.

For lists, `[::]` is the empty list, h `::` t is the list whose head is the element h and whose tail is the list t and l_1 `++` l_2 is the concatenation of the two lists l_1 and l_2. We also use the function `last` that returns the last element of a list. It has an extra argument to handle the case of empty list, `last` c `[::]` $= c$ and `last` c $(h$ `::` $t) =$ `last` h t. Finally, `[seq f` i `|` i `<-` l `]` represents the list built by applying the function f to all the elements of l.

In Coq, there is a distinction between a logical proposition (the type `Prop`) and a boolean expression (the type `bool`) but the library facilitates the bridge between the two by using the so-called small scale reflection [7]. As everything is finite in our application, we state most of our definitions in the boolean world. For example, we use the syntax `[rel` a b `|` P`]` to define a relation between two arbitrary objects a and b with P being a boolean expression where the variables a and b can occur. The boolean equality is written a `==` b, the inequality a `!=` b. The syntax for boolean operators is `!`b, b_1 `&&` b_2, b_1 `||` b_2 and b_1 `==>` b_2 for `NEG`, `AND`, `OR` and `IMP` respectively. There is a cumulative notation for the `AND` and the `OR` : `[&&` b_1, b_2, `...` `&` b_n`]` and `[||` b_1, b_2, `...` `|` b_n`]`. Finally, the syntax for the two boolean quantifications is `[forall` x `:` T, P`]` and `[exists` x `:` T, P`]`.

Disks

A disk is represented by its size. We use the type $\mathtt{I}_n$ of natural numbers strictly smaller than n for this purpose.

```
Definition disk n := I_n.
```

In the following, we use the convention that a variable n will always represent a number of disks, and d a disk (an element of **disk** n).

As there is an implicit conversion from $\mathtt{I}_n$ to natural number, the comparison of the respective size of two disks is simply written as $d_1 < d_2$. A minimal element (`ord0`) and a maximal element (`ord_max`) are defined for $\mathtt{I}_n$ when n is not zero[2]. We use them to represent the smallest disk and the largest one.

```
Definition sdisk : disk n.+1 := ord0.
Definition ldisk : disk n.+1 := ord_max.
```

In particular, **ldisk** is the main actor of our proofs by simple induction on the number of disks, it represents the largest disk.

Pegs

We also use $\mathtt{I}_n$ for pegs.

```
Definition peg k := I_k.
```

In the following, we use the convention that a variable k will always represent a number of pegs, and p a peg (an element of **peg** k).

We mostly use elements of **peg** 3 or **peg** 4 but some generic properties hold for **peg** k. An operation associated to pegs is the one that picks a peg that differs from an initial peg p_i and a destination peg p_j when possible[3]. It is written as $p[p_i, p_j]$. Generic and specific properties are derived from it. For example, we have:

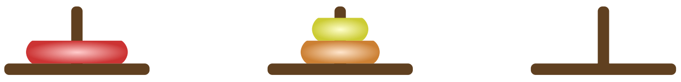
[2] $\mathtt{I}_0$ is an empty type.
[3] If it is not possible (when working with **peg** 1 or **peg** 2), it returns `ord0`

```
Lemma opeg_sym (p₁ p₂ : peg k) : p[p₁, p₂] = p[p₂, p₁].
Lemma opegDl (p₁ p₂ : peg k.+3) : p[p₁, p₂] != p₁.
Lemma opeg3Kl (p₁ p₂ : peg 3) : p₁ != p₂ → p[p[p₁, p₂], p₁] = p₂.
```

The symmetry is valid for every number of pegs. The property of being distinct is only fulfilled when we have more than two pegs. Finally, there is a version of the pigeon-hole principle for three pegs.

Configuration

If we start from the initial position and follow the rules, all the configurations we can encounter are such that on each peg, the disks are always ordered from the largest to the smallest. This means that just recording which disk is on which peg is enough to encode a configuration. To give an example, if we consider the following configuration with three disks and three pegs :



knowing that the small disk is on the second peg, the medium disk is on the second peg and the large disk is on the first peg is enough to recover the information that the small disk is on top of the medium one.

Configurations are then simply represented by finite functions from disks to pegs.

```
Definition configuration k n := {ffun disk n → peg k}.
```

From a technical point of view, using finite functions gives for free functional extensionality (which is not valid for usual functions in Coq). As a consequence, we can use the boolean equality == to test equality between two configurations.

A *perfect configuration* is a configuration where all the disks are on the same peg. So, it is encoded as the constant function:

```
   Definition perfect p := [ffun d ⇒ p].
```

It is written as $c[p]$ in the following, or $c[p, n]$ when the number of disks n is given explicitly in order to help typechecking.

Note that our encoding of configurations has the merit of covering exactly valid configurations. The price to pay is that we have to recover some natural notions. One of these notions is the predicate **on_top** d c that indicates that the disk d is on top of its peg in the configuration c. It is defined as follows:

```
Definition on_top (d : disk n) (c : configuration k n) :=
    [forall d₁ : disk n, c d == c d₁ ==> d ≤ d₁].
```

It simply states that d is on top if every disk on the same peg as d has a size larger than d.

Most of the main results of the library are proved by some kind of inductive argument. In order to apply the inductive hypothesis formally, it is then central to be able to see a subset of a configuration composed of some disks and/or some pegs as a proper configuration with lesser disks and/or lesser pegs. As configurations are functions, it consists in restricting the domain and/or the codomain of the function. We call these operations transformations and usually also define their inverse. Here, we are only going to give an overview of the transformations we have been using : showing their type and what they are used for but not their definition. We refer to the library for the actual implementation.

The most common transformation that is used in proofs by simple induction is to see the configuration without the largest disk as a configuration with one disk less or conversely to extend a configuration with a new large disk that is put at the bottom on an arbitrary peg p to get a configuration with one disk more[4].

```
Definition cunliftr (c : configuration k n.+1) : configuration k n.
Definition cliftr (c : configuration k n) (p : peg k) : configuration k n.+1.
```

A dedicated notation ↓[c] is associated with **cunlift** c and a corresponding one, ↑[c]$_p$, for **clift** c p. A set of basic properties is derived for these operations. For example, we have:

```
Lemma cunliftrK (c : configuration k n.+1) : ↑[ ↓[c]]_(c ldisk) = c.
Lemma perfect_unliftr (p : peg k) : ↓[c[p, n.+1]] = c[p, n].
```

If we remove the largest disk then add it to the same peg, we get the same configuration. If we remove the last disk of a perfect configuration on peg p, we obtain a perfect configuration.

Similarly, in proofs by strong induction, one may need to take bigger piece of configuration. One way to do this is to be directed by the type (here the addition).

```
Definition clshift (c : configuration k (m + n)) : configuration k m.
Definition crshift (c : configuration k (m + n)) : configuration k n.
Definition cmerge (c1 : configuration k m) (c2 : configuration k n) :
    configuration k (m + n).
```

clshift builds a configuration with m disks taking the m largest disks of c while **crshift** builds a configuration with n disks taking the n smallest disks of c. As a matter of fact, ↓[c] and ↑[c]$_p$ are just defined as a special case of these operators for $m = 1$.

Other kinds of transformations that have been defined but not used frequently are:

```
Definition ccut (C : c ≤ n) (c : configuration k n) : configuration k c.
Definition ctuc (C : c ≤ n) (c : configuration k n) : configuration k (n - c).
Definition cset (s : {set (disk n)}) (c : configuration k n) : configuration k #|s|.
```

[4] Following, the `lift` and `unlift` operations for `I`$_n$ of the Mathematical Component Library, we take the convention that lifting a configuration adds a disk.

```
Definition cset2 (sp : {set (peg k)}) (sd : {set (disk n)})
                  (c : configuration k n) : configuration #|sp| #|sd|.
```

ccut and **ctuc** are the equivalent of **crshift** and **clshift** but directed by the proof C rather than the type. **cset** considers a subset of disks but with the same number of peg (#$|s|$ is the cardinal of s). **cset2** is the most general and considers both a subset of disks and a subset of pegs.

Move

A move is defined as a relation between configurations. As we want to possibly add constraints on moves, it is parameterised by a relation r on pegs: $r\ p_1\ p_2$ indicates that it is possible to go from peg p_1 to peg p_2. Assumptions are usually added on the relation r (such as irreflexivity or symmetry) in order to prove basic properties of moves.

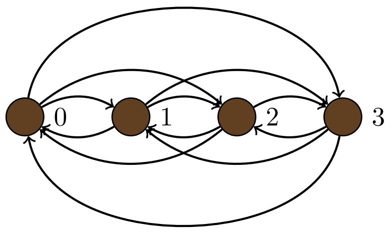
Here is the formal definition of a move:

```
Definition move : rel (configuration k n) :=
   [rel c1 c2 | [exists d1 : disk n,
     [&& r (c1 d1) (c2 d1), on_top d1 c1, on_top d1 c2 &
        [forall d2, d1 != d2 ==> c1 d2 == c2 d2]]]].
```

It simply states that there is a disk d_1 that fulfills 4 conditions:

- the move of d_1 from its peg in c_1 to its peg in c_2 is compatible with r;
- the disk d_1 is on top of its peg in c_1;
- the disk d_1 is on top of its peg in in c_2;
- it is the unique disk that has possibly moved.

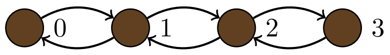
The standard puzzle has no restriction on the moves between pegs as long as we don't put a large disk on top of a small one. If we draw the possible moves as arrows between pegs, the picture for four pegs gives the following complete graph:



We call this version *regular*. It is denoted with the r exponent. For example, $P_5^{4^{\mathtt{r}}}$ corresponds to the puzzle with four pegs and five disks with no restriction on the moves. Its associated relation **rrel** only enforces irreflexivity:

```
Definition rrel : rel (peg k) := [rel x y | x != y].
Definition rmove : rel (configuration k n) := move rrel.
```

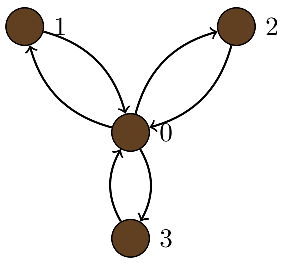
The first variant we consider is where one can only move from one peg to its neighbour. The picture for four pegs is the following:



This version is called *linear* and its associated infix is `l`. For example, the puzzle with four pegs and five disks with linear moves is written $P_5^{4\mathtt{l}}$. The corresponding relation **lrel** uses simple arithmetic to check neighbourhood :

```
Definition lrel : rel (peg k) := [rel x y | (x.+1 == y) || (y.+1 == x)].
Definition lmove : rel (configuration k n) := move lrel.
```

Finally the last variant we consider is the one where one central peg is the only one that can communicate with its outer pegs. A picture for four pegs gives



This version is called *star* and its associated exponent is `s`. For example, the puzzle with four pegs and five disks with star moves is written $P_5^{4\mathtt{s}}$. The corresponding relation **srel** uses multiplication to put the peg 0 in the center :

```
Definition srel : rel (peg k) := [rel x y | (x != y) && (x * y == 0)].
Definition smove : rel (configuration k n) := move srel.
```

Note that these categories may overlap when there are few pegs. For example, $P_n^{3\mathtt{l}}$ and $P_n^{3\mathtt{s}}$ correspond to the same puzzle.

2.1 Path and distance

Moves are defined as relations over configurations. So, we can see sequences of moves as paths on a graph whose nodes are the configurations. As configurations belong to a finite type, we can benefit from the elements of graph theory that are present in the Mathematical Component library. For example, (`rgraph` **move** c) returns the set of all the configurations that are reachable from c in one move, (`connect` **move** c_1 c_2) indicates that c_1 and c_2 can be connected through moves, or (`path` **move** c cs) gives that the sequence cs of configurations is a path that connects c with the last element of cs.

Now, all the transformations on configurations need to be lifted to paths. As distinct configurations may become identical when taking sub-parts, we first need to define an operation on sequences that removes repetitions.

```
Fixpoint rm_rep (A : eqType) (a : A) (s : seq A) :=
  if n is b :: s1 then
    if a == b then rm_rep b s1 else b :: rm_rep b s1
  else [::]
```

It is then possible to derive properties on paths. For example, we have:

```
Lemma path_clshift (c : configuration (m + n)) cs :
   path move c cs →
   path move (clshift c) (rm_rep (clshift c) [seq (clshift i) | i <- cs]).
```

As we want to show that some algorithms are optimal, the last ingredient we need is a notion of distance between configurations. Unfortunately, there is no built-in notion of distance in the Mathematical Component library, so we have to define one. For this, we first build recursively the function **connectn** that computes the set of elements that are connected with exactly n moves. Then, we can define the distance between two points x and y as the smallest n such as (**connectn** r n x y) holds. It is defined as (**gdist** r x y) and written as `d[`x`,` y`]`$_r$ in the following. From this definition, the triangle inequality is derived as :

```
Lemma gdist_triangular r x y z : d[x, y]_r ≤ d[x, z]_r + d[z, y]_r.
```

Finally, we introduce the notion of geodesic path: a path that realises the distance.

```
Definition gpath r x y p :=
   [&& path r x p, last x p == y & d[x, y]_r == size p].
```

Companion theorems are derived for these basic notions. For example, the following lemma shows that concatenation behaves well with respect to distances.

```
Lemma gdist_cat r x y p1 p2 :
   gpath r x y (p1 ++ p2) → d[x, y]_r = d[x, last x p1]_r + d[last x p1, y]_r.
```

3 Puzzles with three pegs

The proofs associated with the puzzles with three pegs are straightforward. They are done by induction on the number of disks inspecting the moves of the largest disk. What makes the simple induction work so well with three pegs is that when, from a configuration c, the largest disk moves from p_i to p_j, all the smaller disks in c are necessarily on the peg $p[p_i, p_j]$. So, they make a perfect configuration on which one can apply the inductive hypothesis using ↓`[`c`]`.

3.1 Regular puzzle

It is easy to translate in Coq the algorithm described in the introduction. We write it as a recursive function that works on n disks and generates the sequence of configurations that goes from the configuration $c[p_1]$ to the configuration $c[p_2]$:

```
Fixpoint ppeg $n$ $p_1$ $p_2$ :=
   if $n$ is $n_1$.+1 then
     let $p_3$ := $p[p_1, p_2]$ in
     [seq ↑ $[i]_{p_1}$ | $i \leftarrow$ ppeg $n_1$ $p_1$ $p_3$] ++ [seq ↑ $[i]_{p_2}$ | $i \leftarrow c[p_3]$ :: ppeg $n_1$ $p_3$ $p_2$]
   else [::].
```

We pick an auxiliary peg p_3, appropriately lift the results of the two recursive calls and concatenate them to get the resulting path. It is easy to prove the basic properties of this function

```
Lemma size_ppeg $n$ $p_1$ $p_2$ : size (ppeg $n$ $p_1$ $p_2$) $= 2^n - 1$
Lemma last_ppeg $n$ $p_1$ $p_2$ $c$ : last $c$ (ppeg $n$ $p_1$ $p_2$) $= c[p_2]$.
Lemma path_ppeg $n$ $p_1$ $p_2$ : $p_1$ != $p_2$ $\rightarrow$ path rmove $c[p_1]$ (ppeg $n$ $p_1$ $p_2$).
```

Note that even for such simple theorems, there are some little subtleties. The **last_ppeg** does hold unconditionally even when the function returns the empty list. It is because this only happens when the configuration has no disk, so the theorem is an equality between elements of an empty type. Also, the **path_ppeg** needs the condition p_1 != p_2 otherwise it will try to move the largest peg from p_1 to p_1 that is not valid since the relation **rmove** is irreflexive.

The key property of the **ppeg** function is that it builds a path of minimal size. As a matter of fact, we have proved something slightly stronger : it is the unique minimal path. In order to state this property, we make use of the extended comparison ($e_1 \leq e_2$?= iff C) that is available in the library. It tells not only that e_1 is smaller than e_2 but also that the condition C indicates exactly when the comparison between e_1 and e_2 is an equality. This comparison comes with some algebraic rules. For example, the transitivity gives that if ($e_1 \leq e_2$?= iff C_1) and ($e_2 \leq e_3$?= iff C_2) hold, we have ($e_1 \leq e_3$?= iff C_1 && C_2). With this comparison, the uniqueness and minimality are stated as :

```
Lemma ppeg_min $n$ $p_1$ $p_2$ $cs$ :
   $p_1$ != $p_2$ $\rightarrow$ path rmove $c[p_1]$ $cs$ $\rightarrow$ last $c[p_1]$ $cs$ $= c[p_2]$ $\rightarrow$
   $2^n - 1 \leq$ size $cs$ ?= iff ($cs$ == ppeg $n$ $p_1$ $p_2$).
```

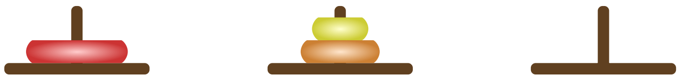
The proof is simply done by double induction (one on the size of cs and one on n) inspecting the moves of the largest disk in the sequence cs. Since p_1 differs from p_2, we know that the largest disk must move at least one time in cs. If it moves exactly once, the inductive hypothesis on n let us conclude directly. If it moves at least twice, the equality never holds. If the first two moves are on different pegs, adding the inductive hypothesis on $n - 1$ twice plus the two moves of the largest disk gives us a path of size at least $2 \times (2^{n-1} - 1) + 2 = 2^n$. If the largest disk moves on one peg and then returns to the peg p_1, the path has some repetition, so the inductive hypothesis on the size of cs let us conclude.

From this theorem, we easily derive the corollary on the distance.

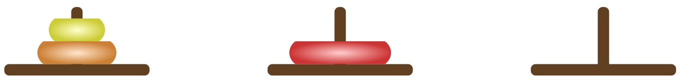
```
Lemma gdist_rhanoi3p $n$ ($p_1$ $p_2$ : peg 3) :
   $d[c[p_1, n], c[p_2, n]]_{\mathbf{rmove}} = (2^n - 1) \times$ ($p_1$ != $p_2$)
```

Note that we have used the automatic conversion from boolean to integer (`true` is 1 and `false` is 0) to include the case where p_1 and p_2 are the same peg.

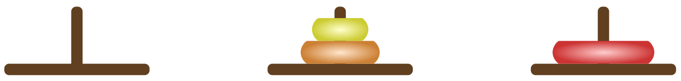
The last theorem only talks about going from a perfect configuration to another one. What about the distance between two arbitrary configurations c_1 and c_2? It seems natural to apply the same greedy strategy always trying to move the largest disk to its destination. The greedy algorithm would proceed as follows : If c_1 **ldisk** is equal to c_2 **ldisk**, we just perform the recursive call for smaller disks. If they are different, we perform a recursive call to move the smaller disks in c_1 to an intermediate peg, $p[c_1\ \mathbf{ldisk}, c_2\ \mathbf{ldisk}]$, then move the largest disk to its position in c_2, and finally perform another recursive call to move the smaller disks to their position in c_2. Unfortunately, this natural strategy is not optimal anymore. We can illustrate this with an example with 3 disks. Let us suppose that we try to go from the initial configuration:



to the final position:



The greedy strategy would move the largest disk only once and would have size seven, one for the largest disk, plus two times moving the two small disks. The optimal strategy instead moves the largest disks twice. It moves it first to the right peg:



and now it follows the greedy strategy and makes four extra moves to reach the target configuration. So, we have a solution of size five.

We have formalised exactly what the optimal solutions are :

- between an arbitrary configuration and a perfect configuration, the greedy strategy is always optimal;
- between two arbitrary configurations c_1 and c_2, the optimal strategy just needs to compare the one-jump solution with the two-jump solution only for the largest disk d such that c_1 d differs from c_2 d.

3.2 Linear puzzle

Implementing the greedy strategy for the linear puzzle is slightly more complicated. As we can move only between pegs that are neighbours, the largest disk may need to jump twice to reach its destination. But from the optimality point of view the situation is much simpler, the greedy strategy is always optimal. This is formally proved and we get the expected theorem about distance between perfect configurations:

```
Lemma gdist_lhanoi3p $n$ ($p_1$ $p_2$ : peg 3) :
    $d[c[p_1, n], c[p_2, n]]_{\mathbf{lmove}}$ =
    if lrel $p_1$ $p_2$ then $(3^n - 1)/2$ else $(3^n - 1) \times (p_1$ != $p_2)$
```

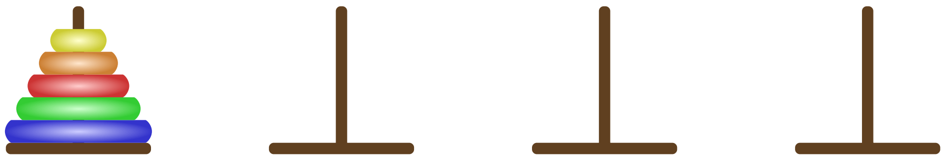
Note that as there are n disks and 3 pegs, there are 3^n possible configurations. This last theorem tells us that the solution that goes from the perfect configuration where all the disks are on the left peg to the perfect configuration where they are on the right peg visits all the configurations!

4 Puzzles with four pegs

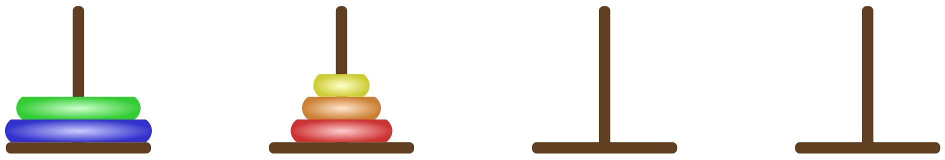
Adding a peg changes completely the situation. If the previous simple recursive algorithm still works, it does not give anymore an optimal solution. The new strategy is implemented by the so-called Frame-Stewart algorithm. We explain it using the regular puzzle with four pegs. Then, we explain how the proofs about the distances for $P_n^{\mathtt{4r}}$ and $P_n^{\mathtt{4s}}$ have been formalised in Coq.

4.1 The Frame-Stewart Algorithm

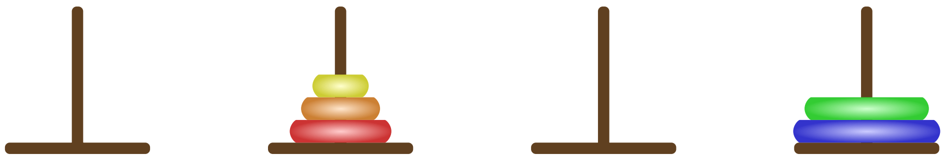
Let us build, $P_n^{\mathtt{4r}}$, an algorithm that moves the disks from the leftmost peg to the rightmost one for the regular puzzles with four pegs.



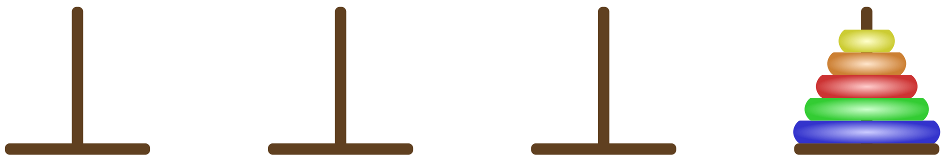
We choose an arbitrary m smaller than n and use $P_m^{\mathtt{4r}}$ to move the top-m disks to an intermediate peg.



The remaining $n - m$ disks can now freely move except on this intermediate peg, so we can use P_{n-m}^{3r} to move them to their destination.



and reuse $P_{4\mathtt{r}k}$ to move the top-m disks to their destination.



Now, we can choose the parameter m as to minimise the number of moves. This means that we have $|P_n^{\mathtt{4r}}| = \min_{m<n} 2|P_m^{\mathtt{4r}}| + (2^{n-m} - 1)$. This strategy can be generalised to an arbitrary

number k of pegs, leading to the recurrence relation:

$$|P_n^{k\mathtt{r}}| = \min_{m<n} 2|P_m^{k\mathtt{r}}| + |P_{n-m}^{k-1\mathtt{r}}|.$$

Knowing if this general program is optimal is an open question but it has been shown optimal for $P_n^{4\mathtt{r}}$ and $P_n^{4\mathtt{s}}$. It is what we have formalised. Note that, from the proving point of view, the new strategy just seems to move from simple induction to strong induction with this new parameter m. As a matter of fact, if we look closely, this new strategy is just a generalisation of the previous one. If we apply it to three pegs, taking the minimum is trivial : there is only one way we can move $n-m$ pegs for the $P_{n-m}^{2\mathtt{r}}$ puzzle; it is by taking $m = n-1$.

4.2 Regular puzzle

The proof given in [2] that shows that the Frame-Stewart algorithm is optimal for the regular puzzle with four pegs is rather technical. As a matter of fact, this technicality was a motivation of our formalisation. The proof is very well written and very convincing but contains several cases which makes it difficult to assess its correctness. A formalisation ensures that no detail has been overlooked. As an anecdote, the Wikipedia page [13] about the Tower of Hanoi in March 2020 was indicating that there was actually a journal paper [6] with a proof of the optimality of the Frame-Stewart algorithm for the regular puzzle with k pegs. When we started formalising this proof, we quickly got stuck on the formalisation of Corollary 3. We contacted the author that told us that it was a known flaw in the proof as documented in [4].

In what follows, we are only going to highlight the overall structure of the proof of optimality of $P_n^{4\mathtt{r}}$. We refer to the paper proof given in [2] for the details. The first step is to relate $|P_n^{4\mathtt{r}}|$ with triangular numbers. Following [2], we write $|P_n^{4\mathtt{r}}|$ as $\Phi(n)$. We introduce the notation Δn for the sum of the first n natural numbers :

$$\Delta n = \sum_{i \le n} i = \frac{n(n+1)}{2}.$$

A number n is triangular if it is a Δi for some i. By analogy to the square root, we introduce the triangular root ∇n :

$$\Delta(\nabla n) \le n < \Delta(\nabla(n+1)).$$

Now, we can give explicit formula for $\Phi(n)$:

$$\Phi(n) = \sum_{i<n} 2^{\Delta i}$$

It is relatively easy to show that the function Φ verifies the recurrence relation of the Frame-Stewart algorithm: $\Phi(n) = \min_{m<n} 2\Phi(m) + (2^{n-m} - 1)$. Then, what is left to be proved is that it is the optimal solution. The key ingredient of the proof is of course to find the right inductive invariant. This is done thanks to a valuation function Ψ that takes a finite set over the natural numbers and returns a natural number:

$$\Psi E = \max_{L \in \mathbb{N}}((1-L)2^L - 1 + \sum_{n \in E} 2^{\min(\nabla n, L)})$$

The idea is that E will contain the disks we are interested in. If we consider the set $[n]$ of all the natural numbers smaller than n (i.e. we are interested by all the disks)

$$[n] = \{\mathbf{set}\ i \mid i < n\}$$

we get back the Φ function we are aiming at:

$$\Psi[n] = \frac{\Phi(n+1)-1}{2} = \frac{\sum_{i<n} 2^{\nabla(i+1)}}{2}$$

Now we can present the central theorem. We consider two configurations u and v of n disks and the four pegs p_0, p_1, p_2 and p_3. If v is such that there is no disk on pegs p_0 and p_1 and E is defined as

$$E = \{\textbf{set } i \mid \textbf{the disk } i \textbf{ is on the peg } p_0 \textbf{ in } u\}$$

the invariant is:

$$d[u,v]_{\textbf{rmove}} \geq \Psi E$$

The proof proceeds by strong induction on the number of disks. It examines a geodesic path p from u to v. If p_2 is the peg where the disk **ldisk** is in v (it cannot be p_0 nor p_1), it considers T the largest disk that was initially on the peg p_0 and visits at least one time the peg p_3. If such a disk does not exist, the inequality easily holds. Then, it considers inside the path p the configuration x_0 before which the disk T leaves the peg p_0 for the first time and the configuration x_3 in which the disk T reaches the peg p_3 for the first time. Similarly, it considers the configuration z_0 before which the disk n leaves the peg p_0 for the first time and the configuration z_2 before which the disk n reaches the peg p_2 for the last time. Examining the respective positions of x_0, x_3, z_0 and z_2 in p and applying some surgery on the configurations of the path p in order to fit the inductive hypothesis it concludes that the inequality holds in every cases.

The six-page long proof of the main theorem 2.9 in [2] translates to a 1000-line long Coq proof.

```
Lemma gdist_le_psi (u v : configuration 4 n) (p0 p2 p3 : peg 4) :
   [∧ p3 != p2, p3 != p0 & p2 != p0] → (codom v) \subset [:: p2 ; p3] →
   Ψ [set i | u i == p0] ≤ d[u, v]rmove.
```

From which, we easily derive the expected theorem:

```
Lemma gdist_rhanoi4 (n : nat) (p1 p2 : peg 4) :
   p1 != p2 → d[c[p1, n], c[p2, n]]rmove = Φ n.
```

5 Star puzzle

We first recall how to apply the Frame-Stewart algorithm to the star puzzle. Let us build the program P_n^{4s} that generates the moves between two perfect configurations: one on an outer peg p_i ($p_i \neq 0$) and the other on another outer peg p_j ($p_j \neq p_i \neq 0$). We first choose a parameter m and use P_m^{4s} to move the top-m disk to the third outer peg p_k ($p_k \neq p_j \neq p_i \neq 0$). Now, we use P_{n-m}^{3s} (which is identical to P_{n-m}^{31}) to move the $n-m$ from peg p_i to peg p_j and avoiding p_k. Finally, we use P_k^{4s} to move the top-m disk from peg p_k to peg p_j. This leads to the recurrence relation:

$$|P_n^{4s}| = \min_{m<n} 2|P_m^{4s}| + (3^{n-m} - 1).$$

Now, we have to find a mathematical object that verifies this recurrence relation. This time it is not the triangular numbers but the increasing sequence α_1 of the elements $2^i 3^j$. The first elements of this sequence are 1, 2, 3, 4, 6, 8, 9. If we define $S_1(n) = \sum_{i<n} \alpha_1(i)$, it is relatively easy to prove that

$$S_1(n) = \min_{m<n} 2S_1(m) + (3^{n-m} - 1)/2.$$

It follows that $2S_1$ verifies the recurrence relation.

Here, the proof of optimality is even more intricate, we just give an idea of the inductive invariant and how the proof proceeds. We refer to [3] for a detailed and very clear exposition of the proof. The first generalisation is to consider distance not only between two configurations but between $l+1$ configurations (i.e. a distance between u_0 and u_l passing through $u_1, \ldots,$ u_{l-1}): $\sum_{i<l} d[u_i, u_{i+1}]$. These intermediate configurations ($0 < i < l$) are alternating. If p_1, p_2 and p_3 are the outer pegs, the configuration u_i is supposed to have its disks on pegs p_2 and $a[p_1, p_3](i)$ where alternation is defined as

$$a[p_i, p_j](0) = p_i \qquad a[p_i, p_j](n+1) = a[p_j, p_i](n)$$

Taking into account this new parameter l, we need to lift S_1 to a parametrised function S_l.

$$S_l(n) = \min_{m<n} 2S_1(m) + l(3^{n-m} - 1)/2$$

and α_1 to $\alpha_l(n) = S_l(n+1) - S_l(n)$. Finally, we introduce the penality function β defined as

$$\beta_{n,l}(k) = \texttt{if } 1 < l \texttt{ and } k+1 = n \texttt{ then } \alpha_l(k) \texttt{ else } 2\alpha_1(k)$$

Given these definitions, the inductive invariant looks like:

```
Lemma main (p₁ p₂ p₃ : peg 4) n l (u : {ffun I_l.+1 → configuration 4 n}) :
   p₁ != p₂ → p₁ != p₃ → p₂ != p₃
   p₁ != p₀ → p₂ != p₀ → p₃ != p₀ →
   (∀k, 0 < k < l → codom (u k) subset [:: p₂; a[p₁, p₃] k]) →
   (S_[l] n).*2 ≤ sum_(i < l) d[u i, u i.+1]_smove +
                  sum_(k < n) (u ord0 k != p₁) * β_[n, l] k +
                  sum_(k < n) (u ord_max k != a[p₁, p₃] l) * β_[n, l] k.
```

The proof is done by simple induction on n. It is then split in several cases depending on the number of elements i such that $u_i(\mathbf{ldisk}) = a[p_1, p_3](i)$. Furthermore, the inductive invariant has this alternating assumption. So, often, one needs to split the path in a bunch of alternating sub-paths in order to apply the inductive hypothesis on each of these sub-paths. This leads to a lower-bound where various values of $S_i(m)$ appear. Key properties of $S_l(n)$ (i.e. its convexity[5] in n and concavity[6] in l) are then used to derive simpler lower-bounds. For example, the combination of these two theorems

[5] A function on natural numbers is convex if f and $n \mapsto f(n+1) - f(n)$ are both increasing.

[6] A function on natural numbers is concave if f is increasing and $n \mapsto f(n+1) - f(n)$ is decreasing.

```
Lemma concaveEk f i k : concave f → f (i + k) + f (i − k) ≤ 2 × (f i)
Lemma concave_dsum_alphaL_l n : concave (fun l → S_[l] n).
```

are often used to simplify inequalities dealing with the S_l function.

The paper proof of the **main** lemma is 15-page long and translates into 3500 lines of Coq proof script. As it contains several crossed references between cases, the formal proof of the **main** lemma is composed of 3 separate sub-lemmas plus one use of the "without loss of generality" tactic [8].

Finally, the Frame-Stewart algorithm gives an upper-bound to the distance and the **main** lemma applied with $l = 1$ gives a lower-bound. Altogether we get the expected theorem:

```
Lemma gdist_shanoi4 (n : nat) (p1 p2 : peg 4) :
   p1 != p2 → p1 != p0 → p1 != p0 → d[c[p1, n], c[p2, n]]_smove = (S_[1] n).*2.
```

6 Conclusion

We have presented a formalisation of the Tower of Hanoi with three and four pegs. Of course, we have only scratched the surface of what can be proved. We refer the reader to [9, 10] for an account of all the mathematical objects and programming techniques this simple puzzle can be linked to.

We started this formalisation as a mere exercise. Then, we got addicted and tried to prove more difficult results. This development relies heavily on the graph library for the modeling part. In that respect, this work bears some similarity with the formalisation of another puzzle, the mini-Rubik, where it is proved that the Rubik cube 2x2x2 can be solved in a maximum of 11 moves by looking at the diameter of the Cayley graph of all its configurations [12]. An attractive aspect of this formalisation is also that it uses very elementary mathematics. So, it is heavily testing the capability to do various flavour of inductive proofs and to manipulate combinations of big operators [1] such as $\max_{i \le n} F$, $\min_{i \le n} F$ and $\sum_{i \le n} F$. To give only one example, in order to prove the concavity of the function S_l we had to revisit the merge sort algorithm. It is usually given as a beginner exercise as a way to merge two sorted lists. Here, it is used to "merge" two increasing functions f and g in an increasing function $\mathbf{fmerge}(f, g)$[7] and we had to prove that

$$\mathbf{fmerge}(f, g)(n) = \max_{i \le n} \min(f(i), g(n - i))$$

When doing it formally, it is very easy to get lost in this kind of proof.

The main contribution of this work is the formal proofs about the distance between two perfect configurations for the 4 pegs versions. These results are relatively recent. We believe that our formal proofs are a natural companion to the paper proofs. These paper proofs are very technical. We have mechanically checked all the details. As a matter of fact, we have been using very little automation, so our formal proofs follow very closely the paper proofs. The main difference is that our formalisation used natural numbers only. So, we have tried

[7] this is used for example to build the increasing sequence of $2^i 3^j$

to avoid as much as possible subtraction. In our formal setting, $(m - n) + n = m$ is a valid theorem only if we add the assumption that $n \leq m$. So an expression such as $a - b \leq c - d$ in the paper proof is translated into $a + d \leq b + c$ in the formal development.

Our formal proofs can, of course, be largely improved. To cite only one example, in the proof of the star puzzle with 4 pegs, lots of subcases are proved by simple symbolic manipulations with applications of some concavity theorems like **concaveEk**. These are currently done by hand. There is clearly room for automating all these steps.

References

1 Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Paşca. Canonical Big Operators. In *TPHOLs*, volume 5170 of *LNCS*, Montreal, Canada, 2008.

2 Thierry Bousch. La quatrième tour de Hanoï. *Bull. Belg. Math. Soc. Simon Stevin*, 21(5):895–912, 2014.

3 Thierry Bousch. La Tour de Stockmeyer. *Séminaire Lotharingien de Combinatoire*, 77(B77d), 2017.

4 Thierry Bousch, Andreas M. Hinz, Sandi Klavžar, Daniele Parisse, Ciril Petr, and Paul K. Stockmeyer. A note on the Frame-Stewart conjecture. *Discrete Math., Alg. and Appl.*, 11(4), 2019.

5 Mathematical Component. The SSReflect Proof Language. Section of the Coq refence manual, available at `https://coq.inria.fr/refman/proof-engine/ssreflect-proof-language.html`.

6 Roberto Demontis. What is the least number of moves needed to solve the k-peg Towers of Hanoi problem? *Discrete Math., Alg. and Appl.*, 11(1), 2019.

7 Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, Inria, 2016. URL: `https://hal.inria.fr/inria-00258384`.

8 John Harrison. Without loss of generality. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 43–59, 2009.

9 Andreas M. Hinz, Sandi Klavžar, Uroš Milutinović, and Ciril Petr. *The Tower of Hanoi – Myths and Maths*. Birkhaüser, 2013.

10 Ralf Hinze. Functional pearl: la tour d'Hanoï. In *ICFP*, pages 3–10. ACM, 2009.

11 Assia Mahboubi and Enrico Tassi. Mathematical components. available at `https://math-comp.github.io/mcb/book.pdf`.

12 Laurent Théry. Proof Pearl: Revisiting the Mini-Rubik in Coq. In *TPHOLs*, volume 5170 of *LNCS*, pages 310–319. Springer, 2008.

13 Wikipedia. Tower of Hanoi, March 2020. available at `https://en.wikipedia.org/w/index.php?title=Tower_of_Hanoi&oldid=943067350`.

Verifying an HTTP Key-Value Server with Interaction Trees and VST

Hengchu Zhang
University of Pennsylvania,
Philadelphia, PA, USA

Wolf Honoré
Yale University, New Haven, CT, USA

Nicolas Koh
Princeton University, NJ, USA

Yao Li
University of Pennsylvania,
Philadelphia, PA, USA

Yishuai Li
University of Pennsylvania,
Philadelphia, PA, USA

Li-Yao Xia
University of Pennsylvania,
Philadelphia, PA, USA

Lennart Beringer
Princeton University, New Haven, NJ, USA

William Mansky
University of Illinois at Chicago, IL, USA

Benjamin Pierce
University of Pennsylvania,
Philadelphia, PA, USA

Steve Zdancewic
University of Pennsylvania,
Philadelphia, PA, USA

Abstract

We present a networked key-value server, implemented in C and formally verified in Coq. The server interacts with clients using a subset of the HTTP/1.1 protocol and is specified and verified using interaction trees and the Verified Software Toolchain. The codebase includes a reusable and fully verified C string library that provides 17 standard POSIX string functions and 17 general purpose non-POSIX string functions. For the KVServer socket system calls, we establish a refinement relation between specifications at user-space level and at CertiKOS kernel-space level.

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

Keywords and phrases formal verification, Coq, HTTP, deep specification

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.32

Supplementary Material The webserver code, proofs, and a step-by-step guide can be found at:
Software (Source Code): `https://zenodo.org/record/4697379`

Funding This work is supported by the National Science Foundation's Expedition in Computing The Science of Deep Specification under the awards 1521602 (Appel), 1521539 (Weirich, Zdancewic, Pierce), and 1521523 (Shao). Additional support is provided by the NSF projects Verified High Performance Data Structure Implementations, award 1005849 (Beringer, Mansky), Random Testing for Language Design, award 1421243 (Pierce), and by the ONR projects N00014-18-2618 and N00014-17-1-2930.

Acknowledgements We thank all members of the DeepSpec project for their collaboration and feedback, and we greatly appreciate the reviewers' comments and suggestions.

1 Introduction

The Science of Deep Specification launched a series of experiments in formal verification of real-world systems [3]. Among these, Koh et al. [25] demonstrated how to verify a simple networked server (called a "swap server"), written in C, against an implementation model written with interaction trees [42], using the Verified Software Toolchain (VST) [4]. The main result was a guarantee that any trace of observed external communications with the server is included in an interaction tree describing intended server behavior.

In the work described here, we scale these verification techniques to a more realistic example: a Key-Value server (KVServer) running over a minimal but practical subset of the HTTP/1.1 protocol. The KVServer provides clients with a `Get(key)` and `Put(key, value)` interface that uses HTTP/1.1 features including GET requests, POST requests, and Content-Length encoding. It runs on the verified operating system CertiKOS [18] or any other OS with a POSIX-compatible socket interface.

Besides significantly scaling up server features and protocol complexity, the present work reduces the set of trusted axioms compared to that of Koh et al. [25]. The network interface of the earlier swap server is described by a set of Hoare triples for the socket system calls, which are *assumed* to be satisfied by the host operating system. In this work, we apply recently developed techniques for proving refinement relations between CertiKOS' kernel-level system call specifications and user-level VST system call specifications, and prove that network interface assumptions from [25] are consistent with CertiKOS' system call specifications [32]. Such proofs are necessary because the specification styles of VST and CertiKOS are different enough that it is not obvious that two specifications describe the same behaviors. One significant difference stems from the logics used by VST and CertiKOS. Another distinction is in their representations of the system state and the external effects of socket operations. The user-level VST socket specifications use interaction trees to describe external effects as observed "on the wire". The CertiKOS specifications, on the other hand, capture these external effects in a logical log of events, while also describing the internal effects on the kernel state, which are invisible to the user-level code.

By proving refinement between the VST and CertiKOS models of socket system calls, we demonstrate that the kernel- and user-level specifications agree on the behavior of sockets. The kernel implementation of the socket system calls in CertiKOS is currently unverified with respect to CertiKOS socket specifications. Our work does *not* attempt to fill this gap (which requires modeling and verifying a TCP/IP stack), but instead proves a refinement between the CertiKOS and VST socket specifications. This guarantees at least that the operating system and the server agree on how sockets are expected to behave, thus removing this interface from the trusted computing base and leaving only the kernel's implementation.

The main challenges in developing the new KVServer stem from the significant increase in feature complexity across all levels of the server. At the connection management level, the KVServer needs verified data structures to maintain incoming and outgoing buffers for multiple concurrent connections. At the protocol level, the KVServer requires a verified parser to deserialize HTTP/1.1 requests and a verified printer to serialize HTTP/1.1 responses. The parser and printer depend on a verified C string library. At the application level, the KVServer needs a verified in-memory map for storing key-value pairs.

This blow-up in feature complexity also calls for a modular approach that can (1) contain the implementation and verification complexity within each module and (2) reduce total proof checking time through parallelization. We achieve this by dividing the entire KVServer into eight independent verified modules. Each module comes with VST specifications for all exposed functions. A module that depends on a lower-level module only needs the lower-level module's C API and VST specifications, while implementation and proof complexity remain hidden away. This modular separation of the KVServer also produces general purpose and reusable low-level modules. Furthermore, we keep function specifications separate from code and avoid intermingling of code and proof information (e.g., loop invariants), as the latter is typically specification-dependent. This organization sets our development apart from the methodologies of verification tools such as Frama-C, VeriFast, KeY, and Dafny [23, 22, 28, 1] and enables us to distribute the verification of individual function bodies into different files that can be processed in parallel. We discuss related verification projects in more detail in Section 6.

Our main contributions are:

1. We demonstrate a verified network server, implemented in C and communicating using a subset of HTTP/1.1. The termination-insensitive specification and proofs are formulated using VST and interaction trees (Section 3).
2. We prove that the network operations in KVServer interaction trees correspond directly to I/O operations performed by the operating system. We use the verification methods of Mansky et al. [32] to demonstrate a refinement between two disparate specifications of the socket interface written in two different specification languages, by abstracting the lower kernel-level CCAL specification on kernel socket states and the logical log of socket operations into a higher user-level VST specification on externally observable network effects. Our work is the first to formally bridge the gap between user- and kernel-level specifications of POSIX network operations (Section 4).
3. We present a reusable and fully verified C string module offering 34 verified C string functions. 17 of these belong to the POSIX string library; the rest are general-purpose string functions used by the KVServer (Section 5).

2 Background

2.1 Interaction Trees

Interaction trees (ITrees) are a data structure and an accompanying Coq library for representing and reasoning about effectful and potentially non-terminating programs [42]. A significant part of the verification work for KVServer involves proving that the server performs various socket system calls in an expected way. We streamlined this effort by using ITree programs to specify the socket-level behavior of KVServer.

To write an ITree program, one must first define the set of visible effects the program can perform. In KVServer, the socket-level network effects are modeled by the following Coq datatype.

```
Variant networkE : Type -> Type :=
| Listen : endpoint_id -> networkE unit
| Accept : endpoint_id -> networkE connection_id
| Shutdown : connection_id -> networkE unit
| RecvByte : connection_id -> networkE byte
| SendByte : connection_id -> byte -> networkE unit.
```

Each constructor describes a network effect parameterized by its argument types (the parameters to the constructor) and its result type (the type argument to `networkE` at the end of each line). For example, the `Listen` constructor represents the server beginning to listen for incoming client connections on an `endpoint_id` (a network address identifier). This operation does not return any meaningful data so its return type is `unit`. We specify how this effect corresponds to the POSIX `listen` system call in the C program in Section 2.3.

ITree programs may involve multiple sets of effects. For example, runtime errors can be expressed in ITree programs through the `exceptE` effect type from the ITree library...

```
Variant exceptE (Err : Type) : Type -> Type :=
| Throw : Err -> exceptE Err void.
```

... and we can use the binary operator `+'` to create a larger effect type: `networkE +' (exceptE string)` is a composition of two kinds of effects, which together allow an ITree program to perform socket effects and throw string-valued errors.

The ITree library also provides a mechanism for expressing effect subsumption relations between effect types, leveraging Coq's typeclass mechanism to automatically resolve subsumption constraints. This mechanism can be used to automatically "lift" a concrete, monomorphic effect type into a polymorphic one. For example, we can use it to define a helper function that generalizes the `Listen` effect:

```
Definition listen `{networkE -< E} : endpoint_id -> itree E unit := embed Listen.
```

Here, the effect type `E` represents some possibly larger set of effects satisfying the subsumption relation `networkE -< E`, and `embed` is an ITree library function that performs the lifting from `networkE` into the subsuming type `E`. The signature of `listen` says that it is a function that receives a network identifier and produces an ITree of type `itree E unit` that, intuitively, calls the "listen" kernel function and returns the unit value once the effect completes.

ITrees satisfy the interface of monads [42], a standard mechanism for composing effectful programs in a pure functional programming context. The monad interface consists of two combinators:

```
Definition bind `{Monad m} {a b : Type} : m a -> (a -> m b) -> m b.
Definition ret `{Monad m} {a : Type} : a -> m a.
```

Intuitively, the `bind` combinator builds a computation that runs the effectful computation in its first argument (with type `m a`) to produce a result of type `a`, passes the result to the continuation in the second argument (with type `a -> m b`), and returns an effectful computation with type `m b`. The `ret` combinator injects an effectless value of type `a` into a computation that may have effects. ITree programmers can use these two combinators to compose ITree values; for convenience, the ITree library provides the notation `a <- m;; k` for the expression `bind m (fun a => k)`.

2.2 Verified Software Toolchain

The Verified Software Toolchain [4] is a Coq framework for verifying C programs using concurrent separation logic [37, 35]. To verify a piece of code, a user employs Coq's programming features to define assertions, connects partial-correctness specifications to function definitions in CompCert's Clight program representation, and finally applies forward symbolic execution tactics to verify the corresponding function bodies. For readability, specifications in this paper are presented in informal notation rather than in VST's Coq-based syntax.

As an example, consider this string library function, which allocates space for a string of a given length.

```
unsigned char* new_string(uint32_t len);
```

Drawing upon predicates Mem and Mtok from VST's verified malloc/free library [5], this function's specification

$$\left\{ \begin{array}{l} !!(l < \texttt{max\_unsigned}) \\ \&\&\ \mathsf{Mem_M\ gv} \end{array} \right\} \texttt{new\_string}(l) \left\{ \begin{array}{l} p.\,\mathsf{if}\ p = \mathsf{null\ then\ Mem_M\ gv} \\ \quad \mathsf{else}\ \ \mathsf{CUStringN}(\mathsf{Ews}, [\,], l+1, p) * \\ \qquad\quad \mathsf{Mtok}(\mathsf{Ews}, \mathsf{uchar}_{l+1}, p) * \mathsf{Mem_M\ gv} \end{array} \right\}$$

asserts that the result p of a call to `new_string` (with a suitable argument l), is either null or is a pointer to some fresh region of memory that satisfies the predicate CUStringN. The "deallocation token" Mtok represents the fact that the client not only gains read/write access (represented by the exclusive-write-share Ews) to the freshly allocated region but may also free it. In addition to these predicates – whose precise definitions we elide – the specification makes use of VST's operators for separating ($*$) and ordinary ($\&\&$) conjunction, and an operator !! that injects a pure Coq proposition into VST's category of assertions. Finally, $\texttt{uchar}_n$ is a shorthand for a length-annotated specialization of CompCert's representation of the function's return type when interpreted as an array. Note that the malloc/free library assertion $\mathsf{Mem}_\mathsf{M}\ \mathsf{gv}$ must be present but is not modified by the call, and that no fresh memory is allocated if the return value is null.

2.3 Specifying Effects with VST and ITrees

We not only use VST for verifying memory safety and application logic, but also rely on a combination of VST and ITrees to verify externally observable effects of C code.

Consider the `listen` system call from the POSIX socket API:

```
int listen(int sockfd, int backlog);
```

Recalling the lifted `Listen` network effect from Section 2.1, we specify the `listen()` system call using two abstract predicates. The predicate ITREE t states that effects described by the interaction tree t are included in the overall effects exhibited by the host OS. The predicate SOCKAPI st states that the host OS has sockets in states corresponding to st, a map from socket identifiers to states (bound, open, listening, etc.).

The specification of `listen()` quantifies over two ITrees, t and k, that respectively describe the sequence of effects performed by the KVServer before and after running this `listen()` network call.

$$\begin{gathered}\{!!((\texttt{listen}\ \mathit{addr};;k) \sqsubseteq t)\ \&\&\ \mathsf{ITREE}\ t * \mathsf{SOCKAPI}\ \mathit{st}\}\\ \texttt{listen}(\mathit{fd}, \mathit{backlog})\\ \left\{\begin{array}{c} r.\ \mathsf{EX}\ t'\ \mathit{st}'.\ !!(-1 \le r \le 0 \land \texttt{post\_listen}\ t\ k\ \mathit{st}\ \mathit{fd}\ \mathit{addr}\ r\ t'\ \mathit{st}')\\ \&\&\ \mathsf{ITREE}\ t' * \mathsf{SOCKAPI}\ \mathit{st}'\end{array}\right\}\end{gathered}$$

Informally, the precondition says that the observed effects in `t` must first be a `listen` effect, followed by effects observable in `k`, and that st is the current internal state of the sockets being managed by the kernel. We think of the tree t as *permission* to perform certain sequences of external operations, from the perspective of an observer that checks off operations one by one as they are performed by the program. The ITree value k is a *continuation* that models effects following this `listen` effect. The relation `post_listen` specifies how t and st evolve to the ITree value t' modeling the remaining observable effects and the updated socket state st', depending on whether the `listen()` system call succeeds. Specifically, if the `listen()` system call fails ($r = -1$), then `post_listen` states that t' is the same as t. (The specification implies that the actual side effect of `listen()` does not occur when the system call fails, and this design leads to a more straightforward connection to the CertiKOS socket specification compared to some alternative designs. We discuss this detail in Section 4.) Note that `post_listen` is purely propositional: it is independent of the memory.

Formally, the predicates ITREE and SOCKAPI are defined as assertions on the *external ghost state* of VST [32], which is kept in sync with a piece of external state in the OS. In this case, the external state is the log of socket communications and the set of currently active sockets; Section 4 will detail how this ghost state is related to CertiKOS' kernel state.

2.4 HTTP/1.1

HTTP/1.1 is a standard network protocol that allows a client (e.g., a web browser) to access and modify resources (e.g., HTML files, databases) stored on a remote server. A client initiates the communication with a request formatted as: (1) a request line consisting of a method (e.g., GET, PUT, POST) indicating the desired action, and the resource on which to perform it; (2) a sequence of header fields that specify extra options; (3) a blank line; and (4) an optional body whose meaning depends on the request line. After handling the request, the server responds with a message comprising (1) a status line that includes the numeric status code (e.g., 404) and a textual message (e.g., "Not Found"); (2) a sequence of header fields with additional information; (3) a blank line; and (4) an optional body [16].

The full HTTP/1.1 specification includes nine methods and many possible headers whose effects range from setting the acceptable language of the response to specifying compression and caching behaviors. In practice, though, only a relatively small subset is necessary for common operations such as retrieving or updating resources: in particular, the only methods the KVServer needs to implement are GET for looking up keys and POST for updating or inserting key-value pairs, and the only header that the server needs to recognize is Content-Length, which indicates the size of the message body. The following is a sample request-response pair for a successful retrieval of the key-value pair `foo` $\mapsto$ `bar`. The ↩ symbol represents an ASCII carriage return and line feed (CRLF) sequence.

```
GET /foo HTTP/1.1↩
↩
HTTP/1.1 200 OK↩
Content-Length: 3↩
↩
bar
```

Though lean, this subset is sufficient to build a non-trivial application and demonstrate the effectiveness of our methodology. Two real-world webservers[1] also implement just this lean subset of HTTP with only GET and POST support.

3 Components

The implementation of our HTTP server is divided into eight verified modules. We verify memory safety and (termination-insensitive) functional correctness of each.

3.1 Infrastructure Modules

StringAPI. The KVServer presents a string-based key-value store over HTTP, and its implementation uses C strings throughout. Our lowest-level infrastructure module is therefore a reusable verified string library with implementations and specifications of many common string functions, plus some useful variants. Due to the details of C memory semantics, idiomatic C string programming introduces proof obligations for side conditions that programmers typically gloss over. We therefore provide alternative implementations and specifications for commonly used string functions to hide these proof obligations from dependent modules. Section 5 describes the string library in more detail.

BufferAPI. This module provides a dynamically allocated resizable byte-buffer data structure. These byte buffers are used to both accumulate data received from clients and to construct data to be sent to clients. We provide verified functions to create, resize, append to, and deallocate byte buffers. Our specifications assert that BufferAPI operations allocate and deallocate memory correctly and do not perform any invalid memory reads or writes.

[1] `http://tinyserver.sourceforge.net/` and `https://sourceforge.net/projects/miniweb/`

SocketAPI. This module provides the VST specifications of POSIX system calls for creating sockets, binding sockets to network addresses, accepting connections, writing and reading data on sockets, and learning which sockets are ready to read or write. This module bridges the CertiKOS kernel and the rest of the KVServer by establishing refinement proofs of the CertiKOS socket specifications against the VST socket specifications. Since these socket operations have network effects that are observable from outside the KVServer, their VST specifications describe the effects they may trigger using the technique introduced in Section 2.3.

The earlier "swap server" described by Koh et al. [25] had similar specifications for the network operations used by our KVServer. However, the specifications of network operations in that work did not have a refinement relation with the CertiKOS socket specifications. In our work, we push this verification boundary lower into CertiKOS with the SocketAPI refinement proofs. We discuss this improvement in Section 4.

3.2 Application Modules

ParseAPI. This module provides functions that parse the subset of HTTP/1.1 requests accepted by the KVServer and functions that serialize standard HTTP responses for KVServer clients. The top-level specification for the parser is a relation between the input string, the parsed request, and the remaining unused input string. This parser specification describes how the parsed request and the unused portion of the input string can be reassembled to recover the original input string. Specifically, the subset of HTTP/1.1 requests that the KVServer accepts are GET and POST requests, and POST requests must have a string as payload in the Content-Length encoding (the length of the string payload must be equal to the value in the Content-Length header).

We also develop an executable parser in Gallina, Coq's internal functional programming language, for the subset of requests KVServer supports, plus refinement proofs between the VST specification of the C parser and the Gallina parser, showing that the C parser is a refinement of the pure Gallina parser.

KeyValueAPI. This module is a thin wrapper around the ParseAPI module that interprets HTTP requests to the KVServer as read/write operations on the KeyValue storage. An HTTP GET request at some `url` is translated into a read request to the KeyValue storage, with the key being the specified `url`. Similarly, an HTTP POST request at some `url` with some `payload` is a write request that puts the value of `payload` under the key `url`. The KeyValueAPI specifications assert that GET and POST requests are correctly translated into key-value read and write requests.

HashtableAPI. This module implements a hash table for string-valued keys and values, which acts as the in-memory KeyValue storage. This module uses a pure Gallina implementation of a string-valued hash table as its specification; we establish the refinement between the Gallina implementation and the C implementation using VST. The hash table uses a verified hash function that computes the hash value of a string by arithmetic manipulations on the ASCII values of the characters in the string. The HashtableAPI specifications assert that the C hash table implementation strictly follows the Gallina implementation model.

3.3 Server I/O Modules

ConnectionAPI. This module pulls in both the application modules and the infrastructure modules and provides an interface for managing and communicating with logical client connections.

The KVServer is a single-threaded event-driven server. The event-driven I/O model allows a single-threaded server to manage multiple concurrent connections. Under this scheme, the server repeatedly tries to receive network data in a loop, buffers available network data from all clients and checks the client buffers for pending requests in each loop iteration.

ConnectionAPI uses the BufferAPI module to maintain both incoming and outgoing byte buffers for each connection. It also relies on the ParseAPI module to abstract over the underlying byte stream by repeatedly trying to parse the accumulated bytes in the receiving buffer until a complete HTTP request has been parsed. This request is then interpreted as a KeyValue request by the KeyValueAPI module, and the corresponding KeyValue operations are executed on the hash table storage.

ConnectionAPI uses a linked-list data structure to manage a collection of connections. Furthermore, ConnectionAPI abstracts over `select` and provides an interface for focusing on connections ready for network I/O.

ServerAPI. This module implements the main event-loop of the entire KVServer. The main loop relies on operations from the ConnectionAPI module to focus on connections that have pending requests to be processed, performs the operations encoded in these pending requests, appends the serialized responses to the outgoing buffers for relevant connections, and flushes outgoing buffers for connections that are ready for outgoing communication. The ServerAPI specification asserts that the main loop does not cause the connection data structures to become invalid and that the hash table storage's content is updated correctly upon client requests.

The ServerAPI module exposes a top-level VST specification for the `main()` function of the entire server and relates the C implementation of the server's main loop to its interaction-tree specification.

$$\left\{\begin{array}{c} !!(\texttt{consistent\_world}\ st)\ \&\& \\ (\mathsf{ITREE}\ \texttt{ITree.iter}\ (\texttt{run\_server})\ ([], \texttt{empty\_table});;k\ \texttt{tt})\ * \\ \mathsf{SOCKAPI}\ st\ *\ \mathsf{Mem_M}\ \mathsf{gv} \end{array}\right\}$$
$$\texttt{main}()$$
$$\{st'.\,\mathsf{ITREE}\ k\ \texttt{tt}\ *\ \mathsf{SOCKAPI}\ st'\ *\ \mathsf{Mem_M}\ \mathsf{gv}\}$$

This states that, starting from a valid state of OS sockets modeled by *st*, the effect of running the `main()` function is reflected by the interaction tree that iterates the server specification `run_server` in a loop, and that the loop starts with the initial empty state ([], `empty_table`). The empty list is the initial empty list of client connections, and the empty table is an initial empty key-value storage table.

The ITree function `run_server` is the specification of a single event-loop iteration of the main server. This iteration is repeated using an ITree combinator `ITree.iter`. The composed specification `ITree.iter run_server` is a program that runs forever, unless the server encounters an irrecoverable error (e.g. memory allocation failure). The proof of this VST triple uses data structure invariants for the hash table and connection list and relies on lemmas that prove each server operation preserves these data structures' invariants. However, these proof details need not be exposed in the top-level VST specification.

Name	Spec	Proof	Impl	Description
String				
POSIX functions	369	1890	143	POSIX-compliant string functions
Non-POSIX functions	563	2393	212	Other string functions
Buffer				
`buffer_append`	49	225	27	Append bytes to a byte buffer
Socket				
`accept`	36	37	N/A	Accept a connection
`listen`	27	16	N/A	Open port to listen for connections
`send`	40	33	N/A	Send bytes
`recv`	44	158	N/A	Receive bytes
`socket`	15	20	N/A	Create a socket
Parse				
`parse_request`	50	921	73	Parse an HTTP request
`serialize_http_response`	34	257	35	Serialize an HTTP response
Hashtable				
`hashtable_get`	27	300	13	Read a key
`hashtable_update`	30	162	5	Set the string at a key
KeyValue				
`message_process`	39	91	12	Run a KeyValue request
Connection				
`connection_set_next`	31	41	3	Sets the tail of a connections list
`connection_get_next`	26	92	3	Gets the tail of a connections list
`process_and_register`	60	305	44	Process an HTTP request
`master_process_and_register`	58	1162	30	Process the entire connections list

Figure 1 Critical verified C functions from each module.

3.4 Summary

To give an idea of the scale of the KVServer and its verification, Figure 1 lists some of the important functions from each module along with the number of lines of C code and Coq specification and proof each required. Figure 2 compares the total number of lines to the earlier Swap Server [25] to highlight the significant increase in scale.

4 Socket API

4.1 Connecting VST Specifications to CertiKOS Socket Calls

The KVServer communicates with clients using POSIX socket system calls: `bind`, `accept`, `send`, `recv`, and so on. These system calls are provided by the verified operating system CertiKOS, which includes functional specifications (either verified or axiomatized) for each system call. Using a technique due to Mansky et al. [32], we connect the VST specifications (separation logic pre- and postconditions) for the socket calls to the Certified Concurrent Abstraction Layers (CCAL) specifications of socket calls provided by CertiKOS, strengthening our confidence in the correctness of our server's network communication.

The basic approach of Mansky et al. [32] is shown in Figure 3. We connect VST specifications of each call to CertiKOS by means of an intermediate-level first-order predicate on CompCert memories (maps from memory locations to values) and external state. Mansky

Lines of Code	KVServer	Swap
Total specification lines	7033	1373
Total proof lines	28998	8545
Total implementation lines	3097	469

Figure 2 Total lines of code for KVServer and Swap Server.

$$\begin{tikzcd}
\text{VST} & \{buf \mapsto vals * \mathsf{EXT}(t)\} \arrow[d] & \texttt{syscall}(buf);\ & \{buf \mapsto vals' * \mathsf{EXT}(t')\} \\
\text{dry} & \mathsf{load}(buf) = vals \land ext = t \arrow[dr] & & \mathsf{load}(buf) = vals' \land ext = t' \arrow[u] \\
\text{CertiKOS} & & \mathsf{syscall}(buf, OS\_state) = OS\_state' \arrow[ur] &
\end{tikzcd}$$

Figure 3 Connecting VST to CertiKOS.

et al. refer to this intermediate-level specification as a “dry specification”. The dry specification serves as a translation layer between the corresponding VST specification and CertiKOS specification; this allows us to prove a round-trip theorem stating that the VST specification follows from the guarantees provided by CertiKOS. For instance, recall the VST specification of the `listen` system call from Section 2.3. The CertiKOS specification for `listen` is:

```
Definition listen_spec (abd : RData) (fd : Z) : option (SysRet Z) :=
  if negb (kern_init abd) then None else
  let socks := ZMap.get (curid abd) abd.(sockets) in
  let log := ZMap.get (curid abd) abd.(socket_log) in
  match is_bound (ZMap.get fd socks) with
  | Some port =>
    let socks' := ZMap.set fd (ListeningSocket port) socks in
    let log' := SysSockListen port :: log in
    Some (abd {sockets: ZMap.set (curid abd) socks' abd.(sockets)}
              {socket_log: ZMap.set (curid abd) log' abd.(socket_log)},
          OK)
  | _ => Some (abd, ERR EBADF) (* Invalid socket state *)
  end.
```

This specification mentions various pieces of OS state that are invisible to the C programmer, including a record of the state of the sockets (i.e., `abd.(sockets)`) that is modified during the call and a log of socket operations performed (i.e., `abd.(socket_log)`). The OS socket states should correspond to the SOCKAPI in the VST specification, while operations appended to the log should be reflected in events removed from the ITREE. We connect the two layers by writing a dry specification for `listen`, in which the assertions of the VST specification are converted to first-order predicates on memory and external state. The dry pre- and postcondition take the parameters of the VST specification as arguments, along with the memory m, external state z, and – in the case of the postcondition – the initial memory m_0 and return value r. They capture the requirements on the external state and reflect the fact that `listen` has no effect on user memory:

$$\begin{aligned}
&\mathrm{Pre}((t, k, st, addr, fd, backlog), m, z) \triangleq (\mathsf{listen}\ addr;; k) \sqsubseteq t \land z = (t, st) \land \\
&\qquad st\ fd = \mathsf{BoundSocket}\ addr \\
&\mathrm{Post}((t, k, st, addr, fd, backlog), m_0, m, z, r) \triangleq m = m_0 \land \exists t'\, st'.\ z = (t', st') \land \\
&\qquad \mathsf{consistent\_world}\ st' \land -1 \le r \le 0 \land \mathsf{post\_listen}\ t\ k\ st\ fd\ addr\ r\ t'\ st'
\end{aligned}$$

In particular, note that the ITREE and SOCKAPI predicates are no longer present, and their contents are translated into assertions on the external state z. We then complete the refinement by showing that the VST precondition implies Pre, that Post implies the VST postcondition, and that if Pre is true of the user memory and external state given to the CertiKOS `listen_spec`, then Post is true of the corresponding parts of the output – thereby translating the assertions on z to effects on `abd.(sockets)` and `abd.(socket_log)`, the OS-level representation of sockets and communication.

The server implementation uses the system calls `socket`, `bind`, `listen`, `accept`, `send`, `recv`, `close`, `shutdown`, and `htons`. For each of these, we prove a connection between the VST proof rule used in the verification of the server and the CertiKOS axiom for the call. The proofs follow the pattern of prior work. Most of the calls only affect external state (socket state and/or interaction tree), while `send` and `recv` also use or change the contents of a single buffer in memory; both of these patterns were illustrated by Mansky et al. [32]. These refinement proofs significantly strengthen our confidence in the correctness of the server by removing the VST socket specifications from the trusted computing base and replacing them with the abstractions provided by the operating system. Indeed, while carrying out the refinement proof, we discovered some correctness conditions that were missing from the original VST specifications. For instance, the `bind` call is only guaranteed to return a valid result when the provided port number is between 0 and 65535; the original VST specification omitted this range requirement.

4.2 Granularity of ITree Events

The VST specification we used for the `listen` call looks like this:

$$\{!!((\texttt{listen}\ addr;;k) \sqsubseteq t)\ \&\&\ \mathsf{ITREE}\ t * \mathsf{SOCKAPI}\ st\}$$
$$\texttt{listen}(fd, backlog)$$
$$\left\{\begin{array}{c} r.\,\mathsf{EX}\,t'\ st'.\ !!(-1 <= r <= 0 \land \texttt{post\_listen}\ t\ k\ st\ fd\ addr\ r\ t'\ st') \\ \&\&\ \mathsf{ITREE}\ t' * \mathsf{SOCKAPI}\ st' \end{array}\right\}$$

Most of the details of what `listen` actually does are hidden in `post_listen`, which says that either the call succeeds and t' is the continuation k (i.e., t minus the `listen` event), or the call fails and $t' = t$. In other words, the `listen` event in the ITREE represents a *successful* call to `listen`, and on failure the server must retry the call before moving on to k. This is only one possible approach to representing communication events with an ITREE: we could imagine using the `listen` event to represent an invocation of the `listen` system call, successful or not, or even a permission guaranteeing that, if the server calls `listen` at this point, then the call will succeed. In the former case, the event would be removed from the ITREE regardless of the result; in the latter case, the error result (and the return value of -1) would not appear in the specification at all.

In the swap server of Koh et al. [25], it was (somewhat arbitrarily) decided that events should represent successful communications, leading to the current style of specification. Now that we have connected the ITREE to the operating system's log, this choice is no longer arbitrary: each ITREE event corresponds to exactly one `socket_log` event, and the OS does not add an event to its log when the call fails. We could build and validate specifications in the other styles (by writing a wrapper function that calls `listen` until it succeeds, or by providing a token from the OS that somehow guarantees that the next `listen` will not fail), but our specification style leads to the most direct translation of the CertiKOS specification into VST. If a user writes a program that assumes `listen` will always succeed and tries to verify it using VST, the gap between their assumptions and the guarantees of the OS will show up in the verification.

5 C Strings in VST

A C string is a contiguous array of non-zero unsigned bytes (1-255), terminated by a null (0) byte. To avoid confusion between a C string value and a Coq string value, we write C strings here with array notation. For example, `['K', 'V', 'S', 'e', 'r', 'v', 'e', 'r', '\0']` is a C string that can be modeled by the Coq string `"KVServer"`.

Programs manipulate C strings through pointers to these byte arrays. For example, our implementation of the standard function `strstr()` takes two pointers of type `const unsigned char *`, representing a "haystack" and a "needle," and searches for the needle in the haystack.

```
const unsigned char* strstr(const unsigned char* hstk, const unsigned char* ndl);
```

Two key properties of a C string are its contiguous memory layout and its terminating null byte; violating these can cause unexpected behaviors and subtle memory bugs [43, 14]. For example, if `hstk` is not null-terminated, `strstr` may read beyond the allocated memory region, possibly leaking secret information or crashing the program.

We use two VST predicates $\mathsf{CUStringN}(\mathtt{sh}, \mathtt{s}, \mathtt{n}, \mathtt{p})$ and $\mathsf{CUString}(\mathtt{sh}, \mathtt{s}, \mathtt{p})$ to model C strings. The predicate CUStringN is defined in terms of an access-permission share, the list of bytes in the string `s`, and a pointer `p` to an array of size `n`. $\mathsf{CUStringN}(\mathtt{sh}, \mathtt{s}, \mathtt{n}, \mathtt{p})$ states that:

1. the list of bytes `s` does not contain a null byte;
2. the length of `s` is strictly smaller than the array's size (`n`);
3. the pointer `p` points to a contiguous memory region that starts with the contents of `s`;
4. the pointer `p` points to a value with the type $\mathtt{uchar}_n$ (i.e., an array of `n` unsigned bytes); and
5. the contents of `s` are immediately followed by a null byte.

The leftover space may hold arbitrary data. The share parameter `sh` controls whether read or write accesses are allowed on the memory where the string is located. In KVServer, we use the share values `Ews` and `Tsh`, which respectively mark heap-allocated read-write memory and stack-allocated read-write memory.

We then further refine CUStringN with CUString by requiring that the length of the array at the pointer `p` is exactly the length of `s` plus 1 (the extra byte is for the terminating null). For example, the C string `['K', 'V', 'S', 'e', 'r', 'v', 'e', 'r', '\0']` allocated at pointer `p` with both read and write permissions can be specified, in Coq, as $\mathsf{CUString}(\mathtt{Ews}, \text{“}\mathtt{KVServer}\text{”}, \mathtt{p})$.

5.1 Specifying strstr

To specify a string function like `strstr`, we need to formally describe two parts:

1. Memory safety: the memory-layout assumptions that the function makes about its inputs. In this case, `strstr` only requires both inputs to be valid C strings.
2. Functional correctness. In this case, if `hstk` contains `ndl`, then `strstr` (either diverges or) returns a pointer to a substring of `hstk` whose prefix is `ndl`. Otherwise, `strstr` returns `NULL`.

For functions that return a substring of one of their arguments, the convention in the standard C string library is to return a pointer at some offset from the input. For example, when `strstr` succeeds, it returns a pointer at some offset `i` into the haystack C string. However, in many instances (especially when working with constant strings), we are primarily interested in the *index* at which the substring begins, rather than the substring itself.

In principle, the returned pointer from `strstr` implicitly encodes a non-negative offset into the haystack string where the needle string can be found: if the haystack string is at pointer `p` and the returned pointer is `n`, then `offset = n - p`. Although this offset is trivial to compute, it adds proof obligations to convince VST that the arithmetic uses only well-defined operations according to the CompCert memory model. While programmers usually think of memory in C programs as a big array of data indexed by memory addresses, and while memory addresses can obviously be subtracted from one another to compute the offset between them, the C standard as reflected in the CompCert memory model is more structured. In VST, memory regions allocated by different calls to `malloc` are considered disjoint, and it is undefined behavior to take a pointer p_A that points to region A and add an offset x to p_A such that $p_A + x$ points to a separate memory region B starting at some pointer p_B, even if arithmetically $p_A + x = p_B$ [29].

Thus, a pointer subtraction like $p_1 - p_2$ induces an extra proof obligation that p_1 and p_2 point to memory addresses within the same memory region. Users of `strstr`'s specification must deal with such proof obligations if what they really want is the offset. Since computing the offset is indeed a common pattern throughout KVServer, we provide an alternative "indexed" version of `strstr` called `strstr_idx` that packages up the pointer subtraction proof and directly returns the offset.

```c
int strstr_idx(const unsigned char* hstk, const unsigned char* ndl) {
  const unsigned char* s = strstr(hstk, ndl);
  if (s == NULL) { return -1; }
  int i = s - hstk;
  return i;
}
```

The specification of `strstr_idx` also reflects some logical simplifications that come with working with offsets instead of pointers:

$$\begin{gathered}\{\mathsf{CUString}(\mathtt{sh1},\mathtt{hstk},\mathtt{hstk_{ptr}}) * \mathsf{CUString}(\mathtt{sh2},\mathtt{ndl},\mathtt{ndl_{ptr}})\}\\ \mathtt{strstr\_idx}(\mathtt{hstk_{ptr}},\mathtt{ndl_{ptr}})\\ \left\{\begin{gathered} i.\,!!(-1 \le i < \mathtt{length}(\mathtt{hstk}))\ \&\&\\ !!(\mathtt{post\_strstr\_idx}\ \mathtt{hstk_{ptr}}\ \mathtt{hstk}\ \mathtt{ndl}\ i)\ \&\&\\ \mathsf{CUString}(\mathtt{sh1},\mathtt{hstk},\mathtt{hstk_{ptr}}) * \mathsf{CUString}(\mathtt{sh2},\mathtt{ndl},\mathtt{ndl_{ptr}})\end{gathered}\right\}\end{gathered}$$

The proposition `post_strstr_idx` used in the postcondition is defined as follows

```coq
Variant post_strstr_idx (ptr1 : val) (s1 s2 : list byte) : Z -> Prop :=
| StrStr_Idx_Not_Found:
     ~ (is_sublist s2 s1)
  -> post_strstr_idx ptr1 s1 s2 (-1)
| StrStr_Idx_Empty:
     s2 = nil
  -> post_strstr_idx ptr1 s1 s2 0
| StrStr_Idx_Found (r : Z):
     0 <= r < Zlength s1
  -> s2 = firstn (List.length s2) (skipn (Z.to_nat r) s1)
  -> ~ (is_sublist s2 (firstn (Z.to_nat r + List.length s2 - 1) s1))
  -> post_strstr_idx ptr1 s1 s2 r.
```

The preconditions state the inputs are valid C strings stored in readable memory, and the postconditions state that the returned value i is an integer between -1 (inclusive) and the length of the "haystack" (exclusive) and that the model haystack string, needle string, and returned value `i` satisfy a relation `post_strstr_idx`. This relation is split into three cases:

1. The needle is not in the haystack and `i = -1`.
2. The needle is empty and `i = 0`.
3. The needle appears at offset `i` in the haystack, and `i` is in the range $[0, \texttt{length}(\texttt{hstk}))$.

The extra proof obligation induced by pointer subtraction is then handled once and for all in the verification of `strstr_idx`.

We applied this technique for three functions in our string library: `strstr`, `strchr`, and `strcasestr`. Each of these has a `_idx` version with a simplified specification, and higher-level modules that depend on the C string module all use the indexed versions instead of the raw pointer versions.

6 Related Work

Verifying networked servers. There are many papers on verifying networked servers, including HTTP servers [11], distributed systems [20, 41], and mail servers [12]. Koh et al. provide a detailed discussion of this previous work [25].

The goals and techniques of our work have much in common with those of Koh et al. [25]. The primary methodology in both projects is to refine a C program against an interaction tree specification using separation logic and VST based on the Clight semantics of CompCert. The KVServer extends the scale of this earlier effort in several dimensions. One is the complexity of the server's state and behavior: The swap server by Koh et al. [25] simply remembers the last integer it received, where our KVServer manages arbitrarily many mappings, which requires operations such as growing and shrinking buffers and string hashing. Another difference is the protocols used for server-client communications. The swap server assumes requests are always 4-byte integers, while the KVServer understands a subset of HTTP/1.1, a ubiquitous industry standard. Handling HTTP requires verified parsing and C string libraries, most of which are generic and could be reused in other verified projects.

Additionally, our work significantly strengthens the connection to CertiKOS. Although Koh et al. [25] discussed connecting user- and kernel-level socket specifications, at the time there was only a work-in-progress proof for `recv`, whereas we provide complete proofs for a significant portion of the POSIX socket interface. The relation between user and kernel state in the swap server also ignored some important details, such as the translation between virtual and physical memory addresses, which are handled correctly in our work.

One limitation of our work compared to the swap server [25] is that we do not provide a full refinement proof connecting the implementation to a high-level "linear specification" that models the server's behavior at the level of whole HTTP requests, hiding the low-level details of parsing and buffering. We believe that the "network refinement" relation between the low-level implementation and such a top-level linear specification can be formulated in terms of linearizability [21]. The additional complexity in the KVServer compared to the swap server arises from the fact that requests and responses are not atomic, since they may be split by the network. As future work, we plan to formalize the connection to linearizability and prove network refinement.

There is a great deal of previous work on verified parsers for network protocols. For example, TRX [26] is a parser interpreter that can be used to extract HTTP parsers with total correctness guarantees, and EverParse [36] is a framework for generating secure parsers and has been used to implement a parser for TLS. Instead of *generating* a verified parser, our work focuses on verifying hand-written C programs that use the standard, low-level C string library to implement HTTP parsing.

Perennial [13] is a new framework for verifying concurrent, crash-safe systems that has been used to implement a mail server. Whereas Perennial focuses on reasoning about crash-safety of concurrent programs, our work focuses on building a networked server whose specification is connected to the host operating system. A potential next step of our work is to incorporate Perennial's crash safety reasoning methodology.

There are also many prior efforts to specify the POSIX socket interface [8, 9, 10, 38]. Since KVServer only requires a subset of the POSIX socket interface, our specification is not as complete as these. However, the KVServer socket specification is formally verified against the specification provided by the host CertiKOS (Section 4), while [8], [9], [10] and [38] all considered the verification against their specifications out of scope.

Modular verification. Beringer and Appel [7, 6] extended VST to support data representation abstraction and separation logic specification subsumption. These improvements made it possible to modularly specify and verify abstract data structures that hide their internal layouts and their operations with support from VST. KVServer uses an earlier release of VST that does not have these features yet, and all internal data structures used by KVServer in fact expose their implementation layouts – all C `structs` are defined in C header files. Although we manually ensure higher-level modules only access lower-level data structures through their verified C APIs to avoid breaking abstractions or introducing spurious dependencies, this manual discipline could be mechanically checked by leveraging these new features in the latest VST.

CCAL [17, 19] is a modular verification technique used in the CertiKOS project. CCAL is a formal calculus that enforces clear separation between the interface of a verified module and its implementation. CCAL gives a formal semantics of horizontal module composition within abstraction layers and vertical composition between abstraction layers. The composition of KVServer modules is similar to the horizontal composition of CCAL modules. However, KVServer does not employ vertical composition or abstraction layers, since these features do not exist yet in VST.

Verifying C strings. The VeriSoft project [2] verified a custom string library [39] based on the C0 semantics, a restricted version of ANSI C99 [27]. Moy and Marché [34] verified 22 functions from the standard C string library of MINIX 3 (`https://www.minix3.org/`); however, they only checked basic safety properties of these functions (e.g., absence of memory access errors), not functional correctness. Efremov et al. [15] verified the functional correctness of 26 string functions from the Linux kernel via deductive verification in Frama-C. Our work includes the functional correctness specifications and proofs of 34 functions (see Figure 1) based on the Clight semantics of CompCert [29]. 17 out of the 34 functions are POSIX compliant.

Interaction trees. Our specifications are phrased in terms of interaction trees, a general-purpose data structure for representing the behaviors of recursive programs that interact with their environments [42]. They are a coinductive variant of "freer monads" [24]; similar data structures include the program monads of Letan et al. [30], the general monads of McBride [33], and the action trees of Swamy et al. [40]. Interaction trees were also used in the specification of the swap server [25].

Connecting user-space and kernel-space specifications. Mansky et al. [32] demonstrated how to connect higher-order specifications with external effects written in VST with first-order specifications written in CCAL. This technique removes a verification gap between user-space programs and the host kernel. We apply Mansky et al.'s verification methods, and prove a refinement between the KVServer and CertiKOS socket specifications.

7 Conclusions and Future Work

We have verified a networked key-value server based on a subset of the HTTP/1.1 protocol, using VST and interaction trees to verify memory safety and functional correctness of the C implementation for each module. We also deepened the connection between KVServer and CertiKOS by proving that the user-level socket specifications agree with kernel-level socket specifications. The resulting proof guarantees the termination-insensitive correctness of the KVServer down to the kernel level, reducing the trusted computing base to the unverified POSIX socket system calls provided by CertiKOS.

As discussed in Section 6, an important future project is to define a high-level specification similar to the "linear specification" of Koh et al. [25] and prove the associated refinement, which can be viewed as a form of linearizability [21].

Specifying servers with interaction trees allows us to test server implementations against the specification [31]. We have written a top-level linear specification for testing purposes, whose relationship with the VST specification is still to be proven. From the testable specification, we have automatically derived a "testing client" that interacts with servers and checks whether they violate the specification. When developing the verified KVServer, we ran it against the derived tester, which has helped shake out a liveness-related bug – when a client pipelines more than one request in a single `send()`, the client connection may hang without immediately processing the latter requests. This liveness bug was out of scope for the verification of the KVServer due to the partial-correctness nature of VST specifications, but we have patched the server implementation and related proofs to correctly handle pipelined requests.

References

1 Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H Schmitt, and Mattias Ulbrich. *Deductive Software Verification–The KeY Book*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.

2 Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert Schirmer, and Artem Starostin. The verisoft approach to systems verification. In Natarajan Shankar and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008, Toronto, Canada, October 6-9, 2008. Proceedings*, volume 5295 of *Lecture Notes in Computer Science*, pages 209–224. Springer, 2008. `doi:10.1007/978-3-540-87873-5_18`.

3 Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. Position paper: the science of deep specification. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 375(2104), 2017. `doi:10.1098/rsta.2016.0331`.

4 Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, USA, 2014.

5 Andrew W. Appel and David A. Naumann. Verified sequential malloc/free. In Chen Ding and Martin Maas, editors, *ISMM'20: 2020 ACM SIGPLAN International Symposium on Memory Management*, pages 48–59. ACM, 2020. `doi:10.1145/3381898.3397211`.

6 Lennart Beringer. Verified software units. In Nobuko Yoshida, editor, *Programming Languages and Systems*, pages 118–147, Cham, 2021. Springer International Publishing.

7 Lennart Beringer and Andrew W. Appel. Abstraction and subsumption in modular verification of C programs. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*, volume 11800 of *Lecture Notes in Computer Science*, pages 573–590. Springer, 2019. `doi:10.1007/978-3-030-30942-8_34`.

8 Steve Bishop, Matthew Fairbairn, Hannes Mehnert, Michael Norrish, Tom Ridge, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: Rigorous test-oracle specification and validation for TCP/IP and the Sockets API. *J. ACM*, 66(1), 2018. `doi:10.1145/3243650`.

9 Steven Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. TCP, UDP, and Sockets: rigorous and experimentally-validated behavioural specification. Volume 1: Overview. Technical Report UCAM-CL-TR-624, Computer Laboratory, University of Cambridge, 2005. 88pp. URL: `http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-624.html`.

10 Steven Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. TCP, UDP, and Sockets: rigorous and experimentally-validated behavioural specification. Volume 2: The specification. Technical Report UCAM-CL-TR-625, Computer Laboratory, University of Cambridge, March 2005. 386pp. URL: `http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-625.html`.

11 Paul E. Black. *Axiomatic Semantics Verification of a Secure Web Server*. PhD thesis, Brigham Young University, Provo, UT, USA, 1998. AAI9820483.

12 Tej Chajed, M. Frans Kaashoek, Butler W. Lampson, and Nickolai Zeldovich. Verifying concurrent software using movers in CSPEC. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 306–322. USENIX Association, 2018. URL: `https://www.usenix.org/conference/osdi18/presentation/chajed`.

13 Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with Perennial. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 243–258. ACM, 2019. `doi:10.1145/3341301.3359632`.

14 Cristina Cifuentes and Bernhard Scholz. Parfait - designing a scalable bug checker. In Florian Martin, Hanne Riis Nielson, Claudio Riva, and Markus Schordan, editors, *Scalable Program Analysis, 13.04. - 18.04.2008*, volume 08161 of *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2008. URL: `http://drops.dagstuhl.de/opus/volltexte/2008/1573/`.

15 Denis Efremov, Mikhail U. Mandrykin, and Alexey V. Khoroshilov. Deductive verification of unmodified Linux kernel library functions. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Verification - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part II*, volume 11245 of *Lecture Notes in Computer Science*, pages 216–234. Springer, 2018. `doi:10.1007/978-3-030-03421-4_15`.

16 Roy T. Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230, 2014. `doi:10.17487/RFC7230`.

17 Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 595–608. ACM, 2015. `doi:10.1145/2676726.2676975`.

18 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 Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 653–669. USENIX Association, 2016. URL: `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu`.

19 Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified concurrent abstraction layers. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 646–661. ACM, 2018. `doi:10.1145/3192366.3192381`.

20 Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 1–17. ACM, 2015. `doi:10.1145/2815400.2815428`.

21 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. `doi:10.1145/78969.78972`.

22 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods Symposium*, pages 41–55. Springer, 2011.

23 Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Aspects Comput.*, 27(3):573–609, 2015. `doi:10.1007/s00165-014-0326-7`.

24 Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 94–105, 2015. `doi:10.1145/2804302.2804319`.

25 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, page 234–248, New York, NY, USA, 2019. Association for Computing Machinery. `doi:10.1145/3293880.3294106`.

26 Adam Koprowski and Henri Binsztok. TRX: A formally verified parser interpreter. *Log. Methods Comput. Sci.*, 7(2), 2011. `doi:10.2168/LMCS-7(2:18)2011`.

27 Dirk Leinenbach, Wolfgang J. Paul, and Elena Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctnes. In Bernhard K. Aichernig and Bernhard Beckert, editors, *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005), 7-9 September 2005, Koblenz, Germany*, pages 2–12. IEEE Computer Society, 2005. `doi:10.1109/SEFM.2005.51`.

28 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010. `doi:10.1007/978-3-642-17511-4_20`.

29 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. `doi:10.1145/1538788.1538814`.

30 Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. Modular verification of programs with effects and effect handlers in Coq. In *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*, pages 338–354, 2018. `doi:10.1007/978-3-319-95582-7_20`.

31 Yishuai Li, Benjamin C. Pierce, and Steve Zdancewic. Model-based testing of networked applications. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

32 William Mansky, Wolf Honoré, and Andrew W. Appel. Connecting higher-order separation logic to a first-order outside world. In Peter Müller, editor, *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12075 of *Lecture Notes in Computer Science*, pages 428–455. Springer, 2020. `doi:10.1007/978-3-030-44914-8_16`.

33 Conor McBride. Turing-completeness totally free. In *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*, pages 257–275, 2015. `doi:10.1007/978-3-319-19797-5_13`.

34 Yannick Moy and Claude Marché. Modular inference of subprogram contracts for safety checking. *J. Symb. Comput.*, 45(11):1184–1211, 2010. `doi:10.1016/j.jsc.2010.06.004`.

35 Peter W. O'Hearn. Separation logic. *Commun. ACM*, 62(2):86–95, 2019. `doi:10.1145/3211968`.

36 Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. Everparse: Verified secure zero-copy parsers for authenticated message formats. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1465–1482. USENIX Association, 2019. URL: `https://www.usenix.org/conference/usenixsecurity19/presentation/delignat-lavaud`.

37 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002. `doi:10.1109/LICS.2002.1029817`.

38 Thomas Ridge, Michael Norrish, and Peter Sewell. TCP, UDP, and Sockets: Volume 3: The Service-level Specification. Technical Report UCAM-CL-TR-742, University of Cambridge, Computer Laboratory, 2009. 305pp.

39 Artem Starostin. Formal verification of a C-library for strings. Master's thesis, Saarland University, Saarbrücken, Germany, 2006.

40 Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. Steelcore: an extensible concurrent separation logic for effectful dependently typed programs. *Proc. ACM Program. Lang.*, 4(ICFP):121:1–121:30, 2020. `doi:10.1145/3409003`.

41 Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for change in a formal verification of the Raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, page 154–165, New York, NY, USA, 2016. Association for Computing Machinery. `doi:10.1145/2854065.2854081`.

42 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`.

43 Fang Yu, Tevfik Bultan, and Ben Hardekopf. String abstractions for string verification. In Alex Groce and Madanlal Musuvathi, editors, *Model Checking Software - 18th International SPIN Workshop, Snowbird, UT, USA, July 14-15, 2011. Proceedings*, volume 6823 of *Lecture Notes in Computer Science*, pages 20–37. Springer, 2011. `doi:10.1007/978-3-642-22306-8_3`.